<https://www.facebook.com/groups/phattrienlaptrinh>

**Clean Code**

<https://www.facebook.com/groups/phattrienlaptrinh>

**Robert C. Martin Series**

The mission of this series is to improve the state of the art of software craftsmanship. The books in this series are technical, pragmatic, and substantial. The authors are highly experienced craftsmen and professionals dedicated to writing about what actually works in practice, as opposed to what might work in theory. You will read about what the author has done, not what he thinks you should do. If the book is about programming, there will be lots of code. If the book is about managing, there will be lots of case studies from real projects.

These are the books that all serious practitioners will have on their bookshelves. These are the books that will be remembered for making a difference and for guiding professionals to become true craftsman.

*Managing Agile Projects* Sanjiv Augustine

*Agile Estimating and Planning* Mike Cohn

*Working Effectively with Legacy Code* Michael C. Feathers

*Agile Java™: Crafting Code with Test-Driven Development* Jeff Langr

*Agile Principles, Patterns, and Practices in C#* Robert C. Martin and Micah Martin

*Agile Software Development: Principles, Patterns, and Practices* Robert C. Martin

*Clean Code: A Handbook of Agile Software Craftsmanship* Robert C. Martin

*UML For Java™ Programmers* Robert C. Martin

*Fit for Developing Software: Framework for Integrated Tests* Rick Mugridge and Ward Cunningham

*Agile Software Development with SCRUM* Ken Schwaber and Mike Beedle

*Extreme Software Engineering: A Hands on Approach* Daniel H. Steinberg and Daniel W. Palmer

**For more information, visit informit.com/martinseries**

<https://www.facebook.com/groups/phattrienlaptrinh>

**Clean Code *A Handbook of Agile***

***Software Craftsmanship***

The Object Mentors: Robert C. Martin

Michael C. Feathers Timothy R. Ottinger Jeffrey J. Langr Brett L. Schuchert James W. Grenning Kevin Dean Wampler Object Mentor Inc.

*Writing clean code is what you must do in order to call yourself a professional. There is no reasonable excuse for doing anything less than your best.*

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco New York • Toronto • Montreal • London • Munich • Paris • Madrid Capetown • Sydney • Tokyo • Singapore • Mexico City



<https://www.facebook.com/groups/phattrienlaptrinh>

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales (800) 382-3419 corpsales@pearsontechgroup.com

For sales outside the United States please contact:

International Sales international@pearsoned.com

Includes bibliographical references and index. ISBN 0-13-235088-2 (pbk. : alk. paper)

1. Agile software development. 2. Computer software—Reliability. I. Title. QA76.76.D47M3652 2008

005.1—dc22 2008024750

Copyright © 2009 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, write to:

Pearson Education, Inc

Rights and Contracts Department 501 Boylston Street, Suite 900 Boston, MA 02116

Fax: (617) 671-3447

ISBN-13: 978-0-13-235088-4 ISBN-10: 0-13-235088-2

Text printed in the United States on recycled paper at Courier in Stoughton, Massachusetts. First printing July, 2008

<https://www.facebook.com/groups/phattrienlaptrinh>

***For Ann Marie:The ever enduring love of my life.***

<https://www.facebook.com/groups/phattrienlaptrinh>

*This page intentionally left blank*

<https://www.facebook.com/groups/phattrienlaptrinh>

**Contents**

[Foreword................................................................................................ xix](#_page_799_0)

[Introduction..........................................................................................xxv](#_page_819_0)

[On the Cover....................................................................................... xxix](#_page_836_0)

[**Chapter 1: Clean Code**........................................................................1](#_page_846_0)

[**There Will Be Code**...............................................................................2](#_page_853_0)

[**Bad Code**................................................................................................3](#_page_859_0)

[**The Total Cost of Owning a Mess**........................................................4](#_page_866_0)

[The Grand Redesign in the Sky..........................................................5](#_page_873_0)

[Attitude...............................................................................................5](#_page_873_0)

[The Primal Conundrum......................................................................6](#_page_881_0)

[The Art of Clean Code?......................................................................6](#_page_881_0)

[What Is Clean Code?..........................................................................7](#_page_889_0)

[**Schools of Thought**..............................................................................12](#_page_916_0)

[**WeAre Authors**....................................................................................13](#_page_923_0)

[**The Boy Scout Rule**.............................................................................14](#_page_929_0)

[**Prequel and Principles**........................................................................15](#_page_935_0)

[**Conclusion**............................................................................................15](#_page_935_0)

[**Bibliography**.........................................................................................15](#_page_935_0)

[**Chapter 2: Meaningful Names**.......................................................17](#_page_947_0)

[**Introduction**.........................................................................................17](#_page_947_0)

[**Use Intention-Revealing Names**.........................................................18](#_page_956_0)

[**Avoid Disinformation**..........................................................................19](#_page_962_0)

[**Make Meaningful Distinctions**...........................................................20](#_page_968_0)

[**Use Pronounceable Names**..................................................................21](#_page_975_0)

[**Use Searchable Names**........................................................................22](#_page_981_0)

vii

<https://www.facebook.com/groups/phattrienlaptrinh>

viii **Contents**

[**Avoid Encodings**..................................................................................23](#_page_987_0)

[Hungarian Notation..........................................................................23](#_page_987_0)

[Member Preﬁxes...............................................................................24](#_page_995_0)

[Interfaces and Implementations .......................................................24](#_page_995_0)

[**Avoid Mental Mapping**.......................................................................25](#_page_1003_0)

[**Class Names**.........................................................................................25](#_page_1003_0)

[**Method Names**.....................................................................................25](#_page_1003_0)

[**Don’t Be Cute** ......................................................................................26](#_page_1015_0)

[**Pick One Word per Concept**...............................................................26](#_page_1015_0)

[**Don’t Pun**.............................................................................................26](#_page_1015_0)

[**Use Solution Domain Names**..............................................................27](#_page_1026_0)

[**Use Problem Domain Names**..............................................................27](#_page_1026_0)

[**Add Meaningful Context**....................................................................27](#_page_1026_0)

[**Don’t Add Gratuitous Context**...........................................................29](#_page_1039_0)

[**Final Words**..........................................................................................30](#_page_1045_0)

[**Chapter 3: Functions**.........................................................................31](#_page_1051_0)

[**Small!**....................................................................................................34](#_page_1067_0)

[Blocks and Indenting........................................................................35](#_page_1075_0)

[**Do One Thing**.......................................................................................35](#_page_1075_0)

[Sections within Functions ................................................................36](#_page_1084_0)

[**One Level of Abstraction per Function**.............................................36](#_page_1084_0)

[Reading Code from Top to Bottom: *The Stepdown Rule*..................37](#_page_1092_0)

[**Switch Statements**...............................................................................37](#_page_1092_0)

[**Use Descriptive Names**........................................................................39](#_page_1112_0)

[**Function Arguments**............................................................................40](#_page_1118_0)

[Common Monadic Forms.................................................................41](#_page_1125_0)

[Flag Arguments ................................................................................41](#_page_1125_0)

[Dyadic Functions..............................................................................42](#_page_1133_0)

[Triads................................................................................................42](#_page_1133_0)

[Argument Objects.............................................................................43](#_page_1141_0)

[Argument Lists.................................................................................43](#_page_1141_0)

[Verbs and Keywords.........................................................................43](#_page_1141_0)

[**Have No Side Effects**...........................................................................44](#_page_1151_0)

[Output Arguments ............................................................................45](#_page_1157_0)

[**Command Query Separation**.............................................................45](#_page_1157_0)

<https://www.facebook.com/groups/phattrienlaptrinh>

**Contents** ix

[**Prefer Exceptions to Returning Error Codes**...................................46](#_page_1165_0)

[Extract Try/Catch Blocks.................................................................46](#_page_1165_0)

[Error Handling Is One Thing............................................................47](#_page_1173_0)

[The Error.java Dependency Magnet .............................................47](#_page_1173_0)

[**Don’t Repeat Yourself** .........................................................................48](#_page_1181_0)

[**Structured Programming** ...................................................................48](#_page_1181_0)

[**How Do You Write Functions Like This?**..........................................49](#_page_1190_0)

[**Conclusion**............................................................................................49](#_page_1190_0)

[**SetupTeardownIncluder**.....................................................................50](#_page_1198_0)

[**Bibliography**.........................................................................................52](#_page_1207_0)

[**Chapter 4: Comments**.......................................................................53](#_page_1213_0)

[**Comments Do Not Make Up for Bad Code**.......................................55](#_page_1223_0)

[**Explain Yourself in Code** ....................................................................55](#_page_1223_0)

[**Good Comments**..................................................................................55](#_page_1223_0)

[Legal Comments...............................................................................55](#_page_1223_0)

[Informative Comments.....................................................................56](#_page_1235_0)

[Explanation of Intent........................................................................56](#_page_1235_0)

[Clariﬁcation......................................................................................57](#_page_1243_0)

[Warning of Consequences................................................................58](#_page_1249_0)

[TODO Comments.............................................................................58](#_page_1249_0)

[Ampliﬁcation....................................................................................59](#_page_1258_0)

[Javadocs in Public APIs....................................................................59](#_page_1258_0)

[**Bad Comments** ....................................................................................59](#_page_1258_0)

[Mumbling.........................................................................................59](#_page_1258_0)

[Redundant Comments......................................................................60](#_page_1270_0)

[Misleading Comments......................................................................63](#_page_1282_0)

[Mandated Comments........................................................................63](#_page_1282_0)

[Journal Comments............................................................................63](#_page_1282_0)

[Noise Comments..............................................................................64](#_page_1292_0)

[Scary Noise ......................................................................................66](#_page_1301_0)

[Don’t Use a Comment When You Can Use a](#_page_1307_0)

[Function or a Variable.......................................................................67](#_page_1307_0)

[Position Markers...............................................................................67](#_page_1307_0)

[Closing Brace Comments.................................................................67](#_page_1307_0)

[Attributions and Bylines...................................................................68](#_page_1317_0)

<https://www.facebook.com/groups/phattrienlaptrinh>

x **Contents**

[Commented-Out Code......................................................................68](#_page_1317_0)

[HTML Comments............................................................................69](#_page_1325_0)

[Nonlocal Information.......................................................................69](#_page_1325_0)

[Too Much Information.....................................................................70](#_page_1333_0)

[Inobvious Connection.......................................................................70](#_page_1333_0)

[Function Headers..............................................................................70](#_page_1333_0)

[Javadocs in Nonpublic Code ............................................................71](#_page_1343_0)

[Example............................................................................................71](#_page_1343_0)

[**Bibliography**.........................................................................................74](#_page_1357_0)

[**Chapter 5: Formatting**......................................................................75](#_page_1363_0)

[**The Purpose of Formatting** ................................................................76](#_page_1370_0)

[**Vertical Formatting**.............................................................................76](#_page_1370_0)

[The Newspaper Metaphor................................................................77](#_page_1378_0)

[Vertical Openness Between Concepts..............................................78](#_page_1385_0)

[Vertical Density................................................................................79](#_page_1391_0)

[Vertical Distance ..............................................................................80](#_page_1397_0)

[Vertical Ordering..............................................................................84](#_page_1412_0)

[**Horizontal Formatting**........................................................................85](#_page_1419_0)

[Horizontal Openness and Density....................................................86](#_page_1426_0)

[Horizontal Alignment.......................................................................87](#_page_1432_0)

[Indentation........................................................................................88](#_page_1438_0)

[Dummy Scopes.................................................................................90](#_page_1447_0)

[**Team Rules**...........................................................................................90](#_page_1447_0)

[**Uncle Bob’s Formatting Rules**............................................................90](#_page_1447_0)

[**Chapter 6: Objects and Data Structures**....................................93](#_page_1464_0)

[**Data Abstraction**..................................................................................93](#_page_1464_0)

[**Data/Object Anti-Symmetry** ..............................................................95](#_page_1476_0)

[**The Law of Demeter**............................................................................97](#_page_1485_0)

[Train Wrecks....................................................................................98](#_page_1493_0)

[Hybrids.............................................................................................99](#_page_1500_0)

[Hiding Structure...............................................................................99](#_page_1500_0)

[**Data Transfer Objects**.......................................................................100](#_page_1508_0)

[Active Record.................................................................................101](#_page_1514_0)

[**Conclusion**..........................................................................................101](#_page_1514_0)

[**Bibliography**.......................................................................................101](#_page_1514_0)

<https://www.facebook.com/groups/phattrienlaptrinh>

**Contents** xi

[**Chapter 7: Error Handling**...........................................................103](#_page_1526_0)

[**Use Exceptions Rather Than Return Codes**...................................104](#_page_1533_0)

[**Write Your Try-Catch-Finally Statement First**.......................105](#_page_1539_0)

[**Use Unchecked Exceptions**...............................................................106](#_page_1545_0)

[**Provide Context with Exceptions**.....................................................107](#_page_1551_0)

[**Deﬁne Exception Classes in Terms of a Caller’s Needs**..................107](#_page_1551_0)

[**Deﬁne the Normal Flow**....................................................................109](#_page_1566_0)

[**Don’t Return Null**..............................................................................110](#_page_1573_0)

[**Don’t Pass Null**..................................................................................111](#_page_1579_0)

[**Conclusion**..........................................................................................112](#_page_1585_0)

[**Bibliography**.......................................................................................112](#_page_1585_0)

[**Chapter 8: Boundaries**....................................................................113](#_page_1593_0)

[**Using Third-Party Code**....................................................................114](#_page_1600_0)

[**Exploring and Learning Boundaries**...............................................116](#_page_1609_0)

[**Learning log4j**.................................................................................116](#_page_1609_0)

[**Learning Tests Are Better Than Free**...............................................118](#_page_1620_0)

[**Using Code That Does Not Yet Exist**................................................118](#_page_1620_0)

[**Clean Boundaries**..............................................................................120](#_page_1632_0)

[**Bibliography**.......................................................................................120](#_page_1632_0)

[**Chapter 9: Unit Tests**.......................................................................121](#_page_1640_0)

[**The Three Laws of TDD**...................................................................122](#_page_1647_0)

[**Keeping Tests Clean**..........................................................................123](#_page_1655_0)

[Tests Enable the -ilities...................................................................124](#_page_1661_0)

[**Clean Tests** .........................................................................................124](#_page_1661_0)

[Domain-Speciﬁc Testing Language................................................127](#_page_1675_0)

[A Dual Standard.............................................................................127](#_page_1675_0)

[**One Assert per Test** ...........................................................................130](#_page_1691_0)

[Single Concept per Test..................................................................131](#_page_1699_0)

[**F.I.R.S.T.**.............................................................................................132](#_page_1705_0)

[**Conclusion**..........................................................................................133](#_page_1711_0)

[**Bibliography**.......................................................................................133](#_page_1711_0)

[**Chapter 10: Classes**..........................................................................135](#_page_1721_0)

[**Class Organization**............................................................................136](#_page_1728_0)

[Encapsulation.................................................................................136](#_page_1728_0)

<https://www.facebook.com/groups/phattrienlaptrinh>

xii **Contents**

[**Classes Should Be Small!**..................................................................136](#_page_1728_0)

[The Single Responsibility Principle...............................................138](#_page_1741_0)

[Cohesion.........................................................................................140](#_page_1750_0)

[Maintaining Cohesion Results in Many Small Classes..................141](#_page_1756_0)

[**Organizing for Change** .....................................................................147](#_page_1777_0)

[Isolating from Change....................................................................149](#_page_1786_0)

[**Bibliography**.......................................................................................151](#_page_1795_0)

[**Chapter 11: Systems**........................................................................153](#_page_1803_0)

[**How Would You Build a City?**..........................................................154](#_page_1810_0)

[**Separate Constructing a System from Using It**..............................154](#_page_1810_0)

[Separation of Main.........................................................................155](#_page_1818_0)

[Factories .........................................................................................155](#_page_1818_0)

[Dependency Injection.....................................................................157](#_page_1831_0)

[**Scaling Up**..........................................................................................157](#_page_1831_0)

[Cross-Cutting Concerns.................................................................160](#_page_1845_0)

[**Java Proxies**........................................................................................161](#_page_1851_0)

[**Pure JavaAOP Frameworks**.............................................................163](#_page_1860_0)

[**AspectJ Aspects** .................................................................................166](#_page_1878_0)

[**Test Drive the System Architecture**..................................................166](#_page_1878_0)

[**Optimize Decision Making**...............................................................167](#_page_1886_0)

[**Use Standards Wisely, When They Add *Demonstrable*Value**.........168](#_page_1892_0)

[**Systems Need Domain-Speciﬁc Languages**.....................................168](#_page_1892_0)

[**Conclusion**..........................................................................................169](#_page_1900_0)

[**Bibliography**.......................................................................................169](#_page_1900_0)

[**Chapter 12: Emergence**..................................................................171](#_page_1936_0)

[**Getting Clean via Emergent Design** ................................................171](#_page_1936_0)

[**Simple Design Rule 1: Runs All the Tests**........................................172](#_page_1945_0)

[**Simple Design Rules 2–4: Refactoring** ............................................172](#_page_1945_0)

[**No Duplication**...................................................................................173](#_page_1953_0)

[**Expressive**...........................................................................................175](#_page_1962_0)

[**Minimal Classes and Methods**.........................................................176](#_page_1968_0)

[**Conclusion**..........................................................................................176](#_page_1968_0)

[**Bibliography**.......................................................................................176](#_page_1968_0)

[**Chapter 13: Concurrency**..............................................................177](#_page_1978_0)

[**Why Concurrency?** ...........................................................................178](#_page_1985_0)

[Myths and Misconceptions.............................................................179](#_page_1991_0)

<https://www.facebook.com/groups/phattrienlaptrinh>

**Contents** xiii

[**Challenges**..........................................................................................180](#_page_1997_0)

[**Concurrency Defense Principles**......................................................180](#_page_1997_0)

[Single Responsibility Principle......................................................181](#_page_2005_0)

[Corollary: Limit the Scope of Data................................................181](#_page_2005_0)

[Corollary: Use Copies of Data.......................................................181](#_page_2005_0)

[Corollary: Threads Should Be as Independent as Possible ............182](#_page_2015_0)

[**Know Your Library**...........................................................................182](#_page_2015_0)

[Thread-Safe Collections.................................................................182](#_page_2015_0)

[**Know Your Execution Models**..........................................................183](#_page_2025_0)

[Producer-Consumer........................................................................184](#_page_2031_0)

[Readers-Writers..............................................................................184](#_page_2031_0)

[Dining Philosophers.......................................................................184](#_page_2031_0)

[**Beware Dependencies Between Synchronized Methods**................185](#_page_2047_0)

[**Keep Synchronized Sections Small**..................................................185](#_page_2047_0)

[**Writing Correct Shut-Down Code Is Hard**.....................................186](#_page_2055_0)

[**Testing Threaded Code**.....................................................................186](#_page_2055_0)

[Treat Spurious Failures as Candidate Threading Issues.................187](#_page_2063_0)

[Get Your Nonthreaded Code Working First....................................187](#_page_2063_0)

[Make Your Threaded Code Pluggable ............................................187](#_page_2063_0)

[Make Your Threaded Code Tunable................................................187](#_page_2063_0)

[Run with More Threads Than Processors.......................................188](#_page_2075_0)

[Run on Different Platforms............................................................188](#_page_2075_0)

[Instrument Your Code to Try and Force Failures............................188](#_page_2075_0)

[Hand-Coded ...................................................................................189](#_page_2085_0)

[Automated......................................................................................189](#_page_2085_0)

[**Conclusion**..........................................................................................190](#_page_2093_0)

[**Bibliography**.......................................................................................191](#_page_2101_0)

[**Chapter 14: Successive Reﬁnement**............................................193](#_page_2109_0)

[**Args Implementation**........................................................................194](#_page_2116_0)

[How Did I Do This? .......................................................................200](#_page_2137_0)

[**Args:The Rough Draft**.....................................................................201](#_page_2143_0)

[So I Stopped...................................................................................212](#_page_2179_0)

[On Incrementalism.........................................................................212](#_page_2179_0)

[**String Arguments**..............................................................................214](#_page_2190_0)

[**Conclusion**.........................................................................................250](#_page_2301_0)

<https://www.facebook.com/groups/phattrienlaptrinh>

xiv **Contents**

[**Chapter 15: JUnit Internals**..........................................................251](#_page_2307_0)

[**The JUnit Framework**.......................................................................252](#_page_2315_0)

[**Conclusion**..........................................................................................265](#_page_2357_0)

[**Chapter 16: Refactoring SerialDate**.........................................267](#_page_2365_0)

[**First, Make It Work**...........................................................................268](#_page_2374_0)

[**Then Make It Right**...........................................................................270](#_page_2383_0)

[**Conclusion**..........................................................................................284](#_page_2430_0)

[**Bibliography**.......................................................................................284](#_page_2430_0)

[**Chapter 17: Smells and Heuristics**.............................................285](#_page_2438_0)

[**Comments**..........................................................................................286](#_page_2445_0)

[C1: *Inappropriate Information*.......................................................286](#_page_2445_0)

[C2: *Obsolete Comment*...................................................................286](#_page_2445_0)

[C3: *Redundant Comment* ...............................................................286](#_page_2445_0)

[C4: *Poorly Written Comment*..........................................................287](#_page_2457_0)

[C5: *Commented-Out Code*.............................................................287](#_page_2457_0)

[**Environment** ......................................................................................287](#_page_2457_0)

[E1: *Build Requires More Than One Step*........................................287](#_page_2457_0)

[E2:*Tests Require More Than One Step*..........................................287](#_page_2457_0)

[**Functions**............................................................................................288](#_page_2471_0)

[F1:*Too Many Arguments*................................................................288](#_page_2471_0)

[F2: *Output Arguments* ....................................................................288](#_page_2471_0)

[F3: *Flag Arguments*........................................................................288](#_page_2471_0)

[F4: *Dead Function* .........................................................................288](#_page_2471_0)

[**General**...............................................................................................288](#_page_2471_0)

[G1: *Multiple Languages in One Source File*..................................288](#_page_2471_0)

[G2: *Obvious Behavior Is Unimplemented*......................................288](#_page_2471_0)

[G3: *Incorrect Behavior at the Boundaries*.....................................289](#_page_2495_0)

[G4: *Overridden Safeties*.................................................................289](#_page_2495_0)

[G5: *Duplication*..............................................................................289](#_page_2495_0)

[G6: *Code at Wrong Level of Abstraction*........................................290](#_page_2505_0)

[G7: *Base Classes Depending on Their Derivatives* .......................291](#_page_2511_0)

[G8:*Too Much Information*.............................................................291](#_page_2511_0)

[G9: *Dead Code*...............................................................................292](#_page_2519_0)

[G10:*Vertical Separation* ................................................................292](#_page_2519_0)

[G11: *Inconsistency*.........................................................................292](#_page_2519_0)

[G12: *Clutter*....................................................................................293](#_page_2529_0)

<https://www.facebook.com/groups/phattrienlaptrinh>

**Contents** xv

[G13:*Artiﬁcial Coupling*.................................................................293](#_page_2529_0)

[G14: *Feature Envy*..........................................................................293](#_page_2529_0)

[G15: *Selector Arguments*................................................................294](#_page_2539_0)

[G16: *Obscured Intent*.....................................................................295](#_page_2545_0)

[G17: *Misplaced Responsibility*.......................................................295](#_page_2545_0)

[G18: *Inappropriate Static*...............................................................296](#_page_2553_0)

[G19: *Use Explanatory Variables* ....................................................296](#_page_2553_0)

[G20: *Function Names Should Say What They Do*..........................297](#_page_2561_0)

[G21: *Understand the Algorithm*.....................................................297](#_page_2561_0)

[G22: *Make Logical Dependencies Physical*...................................298](#_page_2569_0)

[G23: *Prefer Polymorphism to If/Else or Switch/Case*....................299](#_page_2575_0)

[G24: *Follow Standard Conventions*................................................299](#_page_2575_0)

[G25: *Replace Magic Numbers with Named Constants*..................300](#_page_2583_0)

[G26: *Be Precise*..............................................................................301](#_page_2593_0)

[G27: *Structure over Convention*.....................................................301](#_page_2593_0)

[G28: *Encapsulate Conditionals* .....................................................301](#_page_2593_0)

[G29:*Avoid Negative Conditionals*.................................................302](#_page_2603_0)

[G30: *Functions Should Do One Thing*...........................................302](#_page_2603_0)

[G31: *Hidden Temporal Couplings*..................................................302](#_page_2603_0)

[G32: *Don’t Be Arbitrary*.................................................................303](#_page_2613_0)

[G33: *Encapsulate Boundary Conditions*........................................304](#_page_2619_0)

[G34: *Functions Should Descend Only*](#_page_2619_0)

[*One Level of Abstraction*................................................................304](#_page_2619_0)

[G35: *Keep Conﬁgurable Data at High Levels*................................306](#_page_2630_0)

[G36:*Avoid Transitive Navigation*...................................................306](#_page_2630_0)

[**Java**.....................................................................................................307](#_page_2638_0)

[J1:*Avoid Long Import Lists by Using Wildcards*............................307](#_page_2638_0)

[J2: *Don’t Inherit Constants*............................................................307](#_page_2638_0)

[J3: *Constants versus Enums*...........................................................308](#_page_2648_0)

[**Names**.................................................................................................309](#_page_2654_0)

[N1: *Choose Descriptive Names*......................................................309](#_page_2654_0)

[N2: *Choose Names at the Appropriate Level of Abstraction*..........311](#_page_2665_0)

[N3: *Use Standard Nomenclature Where Possible*...........................311](#_page_2665_0)

[N4: *Unambiguous Names*...............................................................312](#_page_2673_0)

[N5: *Use Long Names for Long Scopes*...........................................312](#_page_2673_0)

[N6:*Avoid Encodings*......................................................................312](#_page_2673_0)

[N7: *Names Should Describe Side-Effects.* .....................................313](#_page_2683_0)

<https://www.facebook.com/groups/phattrienlaptrinh>

xvi **Contents**

[**Tests**....................................................................................................313](#_page_2683_0)

[T1: *Insufﬁcient Tests*.......................................................................313](#_page_2683_0)

[T2: *Use a Coverage Tool!*...............................................................313](#_page_2683_0)

[T3: *Don’t Skip Trivial Tests* ............................................................313](#_page_2683_0)

[T4:*An Ignored Test Is a Question about an Ambiguity*..................313](#_page_2683_0)

[T5:*Test Boundary Conditions*........................................................314](#_page_2699_0)

[T6: *Exhaustively Test Near Bugs*....................................................314](#_page_2699_0)

[T7: *Patterns of Failure Are Revealing*............................................314](#_page_2699_0)

[T8:*Test Coverage Patterns Can Be Revealing*...............................314](#_page_2699_0)

[T9:*Tests Should Be Fast*.................................................................314](#_page_2699_0)

[**Conclusion**..........................................................................................314](#_page_2699_0)

[**Bibliography**.......................................................................................315](#_page_2715_0)

[**Appendix A: Concurrency II**.........................................................317](#_page_2723_0)

[**Client/Server Example**......................................................................317](#_page_2723_0)

[The Server......................................................................................317](#_page_2723_0)

[Adding Threading...........................................................................319](#_page_2736_0)

[Server Observations .......................................................................319](#_page_2736_0)

[Conclusion......................................................................................321](#_page_2747_0)

[**Possible Paths of Execution** ..............................................................321](#_page_2747_0)

[Number of Paths.............................................................................322](#_page_2755_0)

[Digging Deeper..............................................................................323](#_page_2761_0)

[Conclusion......................................................................................326](#_page_2773_0)

[**Knowing Your Library**......................................................................326](#_page_2773_0)

[Executor Framework ......................................................................326](#_page_2773_0)

[Nonblocking Solutions...................................................................327](#_page_2783_0)

[Nonthread-Safe Classes..................................................................328](#_page_2789_0)

[**Dependencies Between Methods**](#_page_2795_0)

[**Can Break Concurrent Code** ...........................................................329](#_page_2795_0)

[Tolerate the Failure.........................................................................330](#_page_2801_0)

[Client-Based Locking.....................................................................330](#_page_2801_0)

[Server-Based Locking....................................................................332](#_page_2812_0)

[**Increasing Throughput**.....................................................................333](#_page_2818_0)

[Single-Thread Calculation of Throughput......................................334](#_page_2824_0)

[Multithread Calculation of Throughput..........................................335](#_page_2831_0)

[**Deadlock**.............................................................................................335](#_page_2831_0)

[Mutual Exclusion ...........................................................................336](#_page_2840_0)

[Lock & Wait...................................................................................337](#_page_2846_0)

<https://www.facebook.com/groups/phattrienlaptrinh>

**Contents** xvii

[No Preemption................................................................................337](#_page_2846_0)

[Circular Wait ..................................................................................337](#_page_2846_0)

[Breaking Mutual Exclusion............................................................337](#_page_2846_0)

[Breaking Lock & Wait....................................................................338](#_page_2859_0)

[Breaking Preemption......................................................................338](#_page_2859_0)

[Breaking Circular Wait...................................................................338](#_page_2859_0)

[**Testing Multithreaded Code**.............................................................339](#_page_2869_0)

[**Tool Support for Testing Thread-Based Code** ................................342](#_page_2881_0)

[**Conclusion**..........................................................................................342](#_page_2881_0)

[**Tutorial: Full Code Examples**..........................................................343](#_page_2891_0)

[Client/Server Nonthreaded.............................................................343](#_page_2891_0)

[Client/Server Using Threads ..........................................................346](#_page_2905_0)

[**Appendix B: org.jfree.date.SerialDate**......................................349](#_page_2916_0)

[**Appendix C: Cross References of Heuristics**...........................409](#_page_3098_0)

[**Epilogue**................................................................................................411](#_page_3107_0)

[**Index**......................................................................................................413](#_page_3118_0)

<https://www.facebook.com/groups/phattrienlaptrinh>

*This page intentionally left blank*

<https://www.facebook.com/groups/phattrienlaptrinh>

[**Foreword**](#_page_37_0)

One of our favorite candies here in Denmark is Ga-Jol, whose strong licorice vapors are a perfect complement to our damp and often chilly weather. Part of the charm of Ga-Jol to us Danes is the wise or witty sayings printed on the ﬂap of every box top. I bought a two-pack of the delicacy this morning and found that it bore this old Danish saw:  
Một trong những loại kẹo yêu thích của chúng tôi ở đây tại Đan Mạch là Ga-Jol, có hơi cam thảo mạnh là sự bổ sung hoàn hảo cho thời tiết ẩm ướt và thường se lạnh của chúng tôi. Một phần sức hấp dẫn của Ga-Jol đối với người Đan Mạch chúng tôi là những câu nói khôn ngoan hoặc hóm hỉnh được in trên nắp của mỗi hộp. Tôi đã mua một gói hai món ngon sáng nay và phát hiện ra rằng nó có hình cái cưa cũ của Đan Mạch:

*Ærlighed i små ting er ikke nogen lille ting*.  
Tôi cảm thấy nhẹ nhõm vì nó không giống như tôi.

“Honesty in small things is not a small thing.” It was a good omen consistent with what I already wanted to say here. Small things matter. This is a book about humble concerns whose value is nonetheless far from small.  
“Thành thật trong những điều nhỏ nhặt không phải là một điều nhỏ nhặt.” Đó là một điềm tốt phù hợp với những gì tôi đã muốn nói ở đây. Những điều nhỏ nhặt quan trọng. Đây là một cuốn sách về những mối quan tâm khiêm tốn mà giá trị của nó không hề nhỏ.

*God is in the details*, said the architect Ludwig mies van der Rohe. This quote recalls contemporary arguments about the role of architecture in software development, and par-ticularly in the Agile world. Bob and I occasionally ﬁnd ourselves passionately engaged in this dialogue. And yes, mies van der Rohe was attentive to utility and to the timeless forms of building that underlie great architecture. On the other hand, he also personally selected every doorknob for every house he designed. Why? Because small things matter.  
Chúa ở trong từng chi tiết, kiến trúc sư Ludwig mies van der Rohe nói. Trích dẫn này nhắc lại những lập luận đương đại về vai trò của kiến trúc trong phát triển phần mềm và đặc biệt là trong thế giới Agile. Bob và tôi thỉnh thoảng thấy mình say mê tham gia vào cuộc đối thoại này. Và vâng, mies van der Rohe chú ý đến tiện ích và các hình thức xây dựng vượt thời gian làm nền tảng cho kiến trúc vĩ đại. Mặt khác, anh còn tự tay lựa chọn từng chiếc tay nắm cửa cho từng ngôi nhà do mình thiết kế. Tại sao? Bởi vì những điều nhỏ bé quan trọng.

In our ongoing “debate” on TDD, Bob and I have discovered that we agree that soft-ware architecture has an important place in development, though we likely have different visions of exactly what that means. Such quibbles are relatively unimportant, however, because we can accept for granted that responsible professionals give *some* time to think-ing and planning at the outset of a project. The late-1990s notions of design driven *only* by the tests and the code are long gone. Yet attentiveness to detail is an even more critical foundation of professionalism than is any grand vision. First, it is through practice in the small that professionals gain proﬁciency and trust for practice in the large. Second, the smallest bit of sloppy construction, of the door that does not close tightly or the slightly crooked tile on the ﬂoor, or even the messy desk, completely dispels the charm of the larger whole. That is what clean code is about.  
Trong “cuộc tranh luận” đang diễn ra của chúng tôi về TDD, Bob và tôi đã phát hiện ra rằng chúng tôi đồng ý rằng kiến trúc phần mềm có một vị trí quan trọng trong quá trình phát triển, mặc dù chúng tôi có thể có những tầm nhìn khác nhau về ý nghĩa chính xác của điều đó. Tuy nhiên, những lời ngụy biện như vậy tương đối không quan trọng, bởi vì chúng ta có thể chấp nhận rằng các chuyên gia có trách nhiệm dành thời gian để suy nghĩ và lập kế hoạch ngay từ đầu của một dự án. Những quan niệm cuối những năm 1990 về thiết kế chỉ được thúc đẩy bởi các bài kiểm tra và mã đã biến mất từ ​​lâu. Tuy nhiên, sự chú ý đến từng chi tiết thậm chí còn là nền tảng quan trọng của tính chuyên nghiệp hơn bất kỳ tầm nhìn vĩ đại nào. Thứ nhất, thông qua thực hành trong quy mô nhỏ mà các chuyên gia đạt được sự thành thạo và tin tưởng để thực hành trong quy mô lớn. Thứ hai, một chi tiết nhỏ nhất của việc xây dựng cẩu thả, của cánh cửa không đóng chặt hay viên gạch hơi cong trên sàn nhà, hay thậm chí là chiếc bàn lộn xộn, cũng hoàn toàn xua tan sự quyến rũ của tổng thể lớn hơn. Đó chính là nội dung của mã sạch.

Still, architecture is just one metaphor for software development, and in particular for that part of software that delivers the initial *product* in the same sense that an architect delivers a pristine building. In these days of Scrum and Agile, the focus is on quickly bringing *product* to market. We want the factory running at top speed to produce software. These are human factories: thinking, feeling coders who are working from a product back-log or user story to create *product*. The manufacturing metaphor looms ever strong in such thinking. The production aspects of Japanese auto manufacturing, of an assembly-line world, inspire much of Scrum.  
Tuy nhiên, kiến trúc chỉ là một phép ẩn dụ cho sự phát triển phần mềm, và đặc biệt là phần phần mềm cung cấp sản phẩm ban đầu theo cùng nghĩa mà một kiến trúc sư cung cấp một tòa nhà nguyên sơ. Trong thời đại của Scrum và Agile, trọng tâm là nhanh chóng đưa sản phẩm ra thị trường. Chúng tôi muốn nhà máy chạy với tốc độ cao nhất để sản xuất phần mềm. Đây là những nhà máy của con người: những lập trình viên suy nghĩ, cảm nhận đang làm việc từ back-log của sản phẩm hoặc câu chuyện của người dùng để tạo ra sản phẩm. Phép ẩn dụ sản xuất hiện ra rõ ràng hơn bao giờ hết trong suy nghĩ như vậy. Các khía cạnh sản xuất của ngành sản xuất ô tô Nhật Bản, của một thế giới dây chuyền lắp ráp, truyền cảm hứng cho nhiều Scrum.

xix  
xix

<https://www.facebook.com/groups/phattrienlaptrinh>

xx **Foreword**  
xx Lời nói đầu

Yet even in the auto industry, the bulk of the work lies not in manufacturing but in maintenance—or its avoidance. In software, 80% or more of what we do is quaintly called “maintenance”: the act of repair. Rather than embracing the typical Western focus on *pro-ducing* good software, we should be thinking more like home repairmen in the building industry, or auto mechanics in the automotive ﬁeld. What does Japanese management have to say about *that*?  
Tuy nhiên, ngay cả trong ngành công nghiệp ô tô, phần lớn công việc không nằm ở khâu sản xuất mà nằm ở khâu bảo trì – hoặc phòng tránh nó. Trong phần mềm, 80% hoặc hơn những gì chúng ta làm được gọi một cách kỳ lạ là “bảo trì”: hành động sửa chữa. Thay vì chấp nhận sự tập trung điển hình của phương Tây vào việc sản xuất phần mềm tốt, chúng ta nên suy nghĩ giống như những người thợ sửa chữa nhà trong ngành xây dựng, hoặc thợ sửa xe trong lĩnh vực ô tô. Quản lý Nhật Bản nói gì về điều đó?

In about 1951, a quality approach called Total Productive Maintenance (TPM) came on the Japanese scene. Its focus is on maintenance rather than on production. One of the major pillars of TPM is the set of so-called 5S principles. 5S is a set of disciplines—and here I use the term “discipline” instructively. These 5S principles are in fact at the founda-tions of Lean—another buzzword on the Western scene, and an increasingly prominent buzzword in software circles. These principles are not an option. As Uncle Bob relates in his front matter, good software practice requires such discipline: focus, presence of mind, and thinking. It is not always just about doing, about pushing the factory equipment to pro-duce at the optimal velocity. The 5S philosophy comprises these concepts:  
Vào khoảng năm 1951, một phương pháp tiếp cận chất lượng được gọi là Bảo trì Năng suất Toàn diện (TPM) xuất hiện ở Nhật Bản. Trọng tâm của nó là bảo trì hơn là sản xuất. Một trong những trụ cột chính của TPM là tập hợp các nguyên tắc 5S. 5S là một tập hợp các nguyên tắc—và ở đây tôi sử dụng thuật ngữ “kỷ luật” một cách có tính hướng dẫn. Trên thực tế, các nguyên tắc 5S này là nền tảng của Lean - một từ thông dụng khác ở phương Tây và một từ thông dụng ngày càng nổi bật trong giới phần mềm. Những nguyên tắc này không phải là một lựa chọn. Như chú Bob kể lại trong vấn đề chính của mình, việc thực hành phần mềm tốt đòi hỏi kỷ luật như vậy: tập trung, tỉnh táo và suy nghĩ. Nó không phải lúc nào cũng chỉ là làm, là thúc đẩy thiết bị của nhà máy sản xuất với vận tốc tối ưu. Triết lý 5S bao gồm các khái niệm sau:

**•** *Seiri*, or organization (think “sort” in English). Knowing where things are—using approaches such as suitable naming—is crucial. You think naming identiﬁers isn’t important? Read on in the following chapters.  
• Seiri, hay tổ chức (nghĩ là “sắp xếp” trong tiếng Anh). Biết mọi thứ đang ở đâu - sử dụng các phương pháp như đặt tên phù hợp - là rất quan trọng. Bạn nghĩ rằng việc đặt tên định danh là không quan trọng? Hãy đọc tiếp trong các chương sau.

**•** *Seiton*, or tidiness (think “systematize” in English). There is an old American saying: *A place for everything, and everything in its place*. A piece of code should be where you expect to ﬁnd it—and, if not, you should re-factor to get it there.  
• Seiton, hay ngăn nắp (nghĩ là “systematize” trong tiếng Anh). Có một câu nói cổ của người Mỹ: Một nơi dành cho mọi thứ, và mọi thứ ở đúng vị trí của nó. Một đoạn mã nên ở nơi mà bạn mong muốn tìm thấy nó—và, nếu không, bạn nên tính toán lại để tìm thấy nó ở đó.

**•** *Seiso*, or cleaning (think “shine” in English): Keep the workplace free of hanging wires, grease, scraps, and waste. What do the authors here say about littering your code with comments and commented-out code lines that capture history or wishes for the future? Get rid of them.  
• Seiso, hay làm sạch (trong tiếng Anh nghĩ là “shine”): Giữ cho nơi làm việc không có dây treo, dầu mỡ, phế liệu và chất thải. Các tác giả ở đây nói gì về việc xả rác mã của bạn bằng các nhận xét và dòng mã nhận xét ghi lại lịch sử hoặc mong muốn cho tương lai? Loại bỏ chúng.

**•** *Seiketsu*, or standardization: The group agrees about how to keep the workplace clean. Do you think this book says anything about having a consistent coding style and set of practices within the group? Where do those standards come from? Read on.  
• Seiketsu, hay tiêu chuẩn hóa: Nhóm đồng ý về cách giữ cho nơi làm việc sạch sẽ. Bạn có nghĩ rằng cuốn sách này nói bất cứ điều gì về việc có một phong cách viết mã nhất quán và tập hợp các thực hành trong nhóm không? Những tiêu chuẩn đó đến từ đâu? Đọc tiếp.

**•** *Shutsuke*, or discipline (*self*-discipline). This means having the discipline to follow the practices and to frequently reﬂect on one’s work and be willing to change.  
• Shutsuke, hay kỷ luật (tự kỷ luật). Điều này có nghĩa là có kỷ luật để tuân theo các thực hành và thường xuyên suy nghĩ về công việc của mình và sẵn sàng thay đổi.

If you take up the challenge—yes, the challenge—of reading and applying this book, you’ll come to understand and appreciate the last point. Here, we are ﬁnally driving to the roots of responsible professionalism in a profession that should be concerned with the life cycle of a product. As we maintain automobiles and other machines under TPM, break-down maintenance—waiting for bugs to surface—is the exception. Instead, we go up a level: inspect the machines every day and ﬁx wearing parts before they break, or do the equivalent of the proverbial 10,000-mile oil change to forestall wear and tear. In code, refactor mercilessly. You can improve yet one level further, as the TPM movement inno-vated over 50 years ago: build machines that are more maintainable in the ﬁrst place. Mak-ing your code readable is as important as making it executable. The ultimate practice, introduced in TPM circles around 1960, is to focus on introducing entire new machines or  
Nếu bạn chấp nhận thử thách—vâng, thử thách—đọc và áp dụng cuốn sách này, bạn sẽ hiểu và đánh giá cao điểm cuối cùng. Ở đây, cuối cùng chúng ta cũng đang hướng đến gốc rễ của tính chuyên nghiệp có trách nhiệm trong một nghề liên quan đến vòng đời của một sản phẩm. Khi chúng tôi bảo trì ô tô và các máy móc khác theo TPM, việc bảo trì hỏng hóc—chờ các lỗi xuất hiện—là ngoại lệ. Thay vào đó, chúng tôi tiến lên một cấp độ: kiểm tra máy móc hàng ngày và sửa chữa các bộ phận bị mòn trước khi chúng bị hỏng, hoặc thực hiện tương đương với việc thay dầu 10.000 dặm để ngăn ngừa hao mòn. Trong mã, tái cấu trúc không thương tiếc. Bạn có thể cải thiện thêm một cấp độ nữa, như phong trào TPM đã đổi mới hơn 50 năm trước: chế tạo những cỗ máy dễ bảo trì hơn ngay từ đầu. Làm cho mã của bạn có thể đọc được cũng quan trọng như làm cho nó có thể thực thi được. Thực tiễn cuối cùng, được giới thiệu trong vòng TPM vào khoảng năm 1960, là tập trung vào việc giới thiệu toàn bộ máy mới hoặc

<https://www.facebook.com/groups/phattrienlaptrinh>

**Foreword** xxi  
Lời nói đầu xxi

replacing old ones. As Fred Brooks admonishes us, we should probably re-do major soft-ware chunks from scratch every seven years or so to sweep away creeping cruft. Perhaps we should update Brooks’ time constant to an order of weeks, days or hours instead of years. That’s where detail lies.  
thay thế những cái cũ. Như Fred Brooks đã khuyên chúng ta, có lẽ chúng ta nên làm lại từ đầu những phần phần mềm lớn cứ sau bảy năm hoặc lâu hơn để quét sạch những thứ vụn vặt. Có lẽ chúng ta nên cập nhật hằng số thời gian của Brooks theo thứ tự tuần, ngày hoặc giờ thay vì năm. Đó là nơi chi tiết nằm.

Chi tiết có sức mạnh to lớn, nhưng có điều gì đó khiêm tốn và sâu sắc về cách tiếp cận cuộc sống này, như chúng ta có thể mong đợi một cách rập khuôn từ bất kỳ cách tiếp cận nào có nguồn gốc từ Nhật Bản. Nhưng đây không chỉ là cách nhìn của người phương Đông về cuộc sống; Trí tuệ dân gian Anh và Mỹ có đầy những lời khuyên như vậy. Câu trích dẫn Seiton ở trên được rút ra từ ngòi bút của một bộ trưởng Ohio, người đã coi sự gọn gàng “như một phương thuốc cho mọi mức độ xấu xa”. Seiso thì sao? Nhà sạch thì mát bát sạch ngon cơm. Một ngôi nhà đẹp như thế nào, nhưng một chiếc bàn bừa bộn sẽ cướp đi vẻ đẹp lộng lẫy của nó. Còn Shutsuke trong những vấn đề nhỏ này thì sao? Ai trung tín trong việc nhỏ cũng trung tín trong việc lớn. Còn về việc háo hức tái cấu trúc vào thời điểm có trách nhiệm, củng cố vị trí của một người cho các quyết định “lớn” tiếp theo, thay vì trì hoãn nó? Một khâu trong thời gian tiết kiệm chín. Con chim đầu bắt sâu. Đừng để đến ngày mai những gì bạn có thể làm hôm nay. (Đó là ý nghĩa ban đầu của cụm từ “thời điểm chịu trách nhiệm cuối cùng” trong Lean cho đến khi nó rơi vào tay các chuyên gia tư vấn phần mềm.) Còn việc hiệu chỉnh vị trí của những nỗ lực nhỏ, cá nhân trong một tổng thể lớn thì sao? Những cây sồi hùng mạnh từ những quả sồi nhỏ mọc lên. Hoặc làm thế nào về việc tích hợp công việc phòng ngừa đơn giản vào cuộc sống hàng ngày? Phòng ngừa đáng giá một cân chữa bệnh. Mỗi ngày một quả táo, bác sĩ không tới nhà. Mã sạch tôn vinh nguồn gốc sâu xa của trí tuệ bên dưới nền văn hóa rộng lớn hơn của chúng ta, hoặc nền văn hóa của chúng ta như nó đã từng hoặc nên có, và có thể chú ý đến từng chi tiết.

Even in the grand architectural literature we ﬁnd saws that hark back to these sup-posed details. Think of mies van der Rohe’s doorknobs. That’s *seiri*. That’s being attentive to every variable name. You should name a variable using the same care with which you name a ﬁrst-born child.  
Ngay cả trong các tài liệu kiến trúc vĩ đại, chúng ta cũng tìm thấy những vết cưa quay trở lại những chi tiết được cho là này. Hãy nghĩ về tay nắm cửa của mies van der Rohe. Đó là Seiri. Đó là chú ý đến mọi tên biến. Bạn nên đặt tên cho một biến giống như cách bạn đặt tên cho đứa con đầu lòng.

As every homeowner knows, such care and ongoing reﬁnement never come to an end. The architect Christopher Alexander—father of patterns and pattern languages—views every act of design itself as a small, local act of repair. And he views the craftsmanship of ﬁne structure to be the sole purview of the architect; the larger forms can be left to patterns and their application by the inhabitants. Design is ever ongoing not only as we add a new room to a house, but as we are attentive to repainting, replacing worn carpets, or upgrad-ing the kitchen sink. Most arts echo analogous sentiments. In our search for others who ascribe God’s home as being in the details, we ﬁnd ourselves in the good company of the 19th century French author Gustav Flaubert. The French poet Paul Valery advises us that a poem is never done and bears continual rework, and to stop working on it is abandonment. Such preoccupation with detail is common to all endeavors of excellence. So maybe there is little new here, but in reading this book you will be challenged to take up good disci-plines that you long ago surrendered to apathy or a desire for spontaneity and just “responding to change.”  
Như mọi chủ nhà đều biết, việc chăm sóc và hoàn thiện liên tục như vậy không bao giờ kết thúc. Kiến trúc sư Christopher Alexander - cha đẻ của các mẫu và ngôn ngữ mẫu - xem mọi hành động thiết kế như một hành động sửa chữa nhỏ, cục bộ. Và ông coi sự khéo léo của cấu trúc tinh xảo là mục đích duy nhất của kiến trúc sư; các hình thức lớn hơn có thể được cư dân để lại cho các mẫu và ứng dụng của chúng. Thiết kế không ngừng diễn ra không chỉ khi chúng ta thêm một căn phòng mới vào ngôi nhà, mà còn khi chúng ta chú ý đến việc sơn lại, thay thế những tấm thảm đã mòn hoặc nâng cấp bồn rửa trong nhà bếp. Hầu hết các nghệ thuật lặp lại tình cảm tương tự. Trong khi tìm kiếm những người khác coi nhà của Đức Chúa Trời là chi tiết, chúng tôi thấy mình có mối quan hệ tốt với tác giả người Pháp thế kỷ 19, Gustav Flaubert. Nhà thơ Pháp Paul Valery khuyên chúng ta rằng một bài thơ không bao giờ được hoàn thành và phải liên tục làm lại, và ngừng làm nó là bỏ rơi. Mối bận tâm với chi tiết như vậy là phổ biến đối với tất cả các nỗ lực xuất sắc. Vì vậy, có thể có một chút điều mới mẻ ở đây, nhưng khi đọc cuốn sách này, bạn sẽ được thử thách để tiếp thu những nguyên tắc tốt mà từ lâu bạn đã đầu hàng trước sự thờ ơ hoặc khao khát tự phát và chỉ “đáp ứng với sự thay đổi”.

Unfortunately, we usually don’t view such concerns as key cornerstones of the art of programming. We abandon our code early, not because it is done, but because our value system focuses more on outward appearance than on the substance of what we deliver.  
Thật không may, chúng ta thường không coi những mối quan tâm đó là nền tảng chính của nghệ thuật lập trình. Chúng tôi từ bỏ mã của mình sớm, không phải vì nó đã xong, mà vì hệ thống giá trị của chúng tôi tập trung nhiều hơn vào hình thức bên ngoài hơn là bản chất của những gì chúng tôi cung cấp.

<https://www.facebook.com/groups/phattrienlaptrinh>

xxii **Foreword**  
xxii Lời nói đầu

This inattentiveness costs us in the end: *A bad penny always shows up*. Research, neither in industry nor in academia, humbles itself to the lowly station of keeping code clean. Back in my days working in the Bell Labs Software Production Research organization (*Produc-tion*, indeed!) we had some back-of-the-envelope ﬁndings that suggested that consistent indentation style was one of the most statistically signiﬁcant indicators of low bug density. We want it to be that architecture or programming language or some other high notion should be the cause of quality; as people whose supposed professionalism owes to the mastery of tools and lofty design methods, we feel insulted by the value that those factory-ﬂoor machines, the coders, add through the simple consistent application of an indentation style. To quote my own book of 17 years ago, such style distinguishes excellence from mere competence. The Japanese worldview understands the crucial value of the everyday worker and, more so, of the systems of development that owe to the simple, everyday actions of those workers. Quality is the result of a million selﬂess acts of care—not just of any great method that descends from the heavens. That these acts are simple doesn’t mean that they are simplistic, and it hardly means that they are easy. They are nonetheless the fabric of greatness and, more so, of beauty, in any human endeavor. To ignore them is not yet to be fully human.  
Sự thiếu chú ý này cuối cùng sẽ khiến chúng ta phải trả giá: Một đồng xu xấu luôn xuất hiện. Nghiên cứu, không phải trong ngành công nghiệp cũng như trong học viện, tự hạ mình xuống vị trí thấp kém trong việc giữ sạch mã. Quay trở lại những ngày tôi làm việc trong tổ chức Nghiên cứu Sản xuất Phần mềm Bell Labs (Thực sự là Sản xuất!), chúng tôi đã có một số phát hiện bí mật gợi ý rằng kiểu thụt đầu dòng nhất quán là một trong những chỉ báo có ý nghĩa thống kê nhất về mật độ lỗi thấp . Chúng tôi muốn kiến trúc hoặc ngôn ngữ lập trình hoặc một số khái niệm cao cấp khác phải là nguyên nhân của chất lượng; với tư cách là những người được cho là có tính chuyên nghiệp nhờ vào việc làm chủ các công cụ và phương pháp thiết kế cao cả, chúng tôi cảm thấy bị xúc phạm bởi giá trị mà những cỗ máy sản xuất tại nhà máy đó, những người lập trình, thêm vào thông qua ứng dụng nhất quán đơn giản của kiểu thụt đầu dòng. Để trích dẫn cuốn sách của riêng tôi cách đây 17 năm, phong cách như vậy phân biệt sự xuất sắc với năng lực đơn thuần. Thế giới quan của Nhật Bản hiểu được giá trị quan trọng của người lao động hàng ngày và hơn thế nữa, của các hệ thống phát triển nhờ vào những hành động đơn giản, hàng ngày của những người lao động đó. Chất lượng là kết quả của hàng triệu hành động quan tâm quên mình—chứ không phải của bất kỳ phương pháp tuyệt vời nào từ trên trời rơi xuống. Rằng những hành động này đơn giản không có nghĩa là chúng đơn giản, và hầu như không có nghĩa là chúng dễ dàng. Dù sao thì chúng cũng là kết cấu của sự vĩ đại và hơn thế nữa là của vẻ đẹp trong bất kỳ nỗ lực nào của con người. Bỏ qua chúng vẫn chưa phải là con người hoàn toàn.

Of course, I am still an advocate of thinking at broader scope, and particularly of the value of architectural approaches rooted in deep domain knowledge and software usability. The book isn’t about that—or, at least, it isn’t obviously about that. This book has a subtler message whose profoundness should not be underappreciated. It ﬁts with the current saw of the really code-based people like Peter Sommerlad, Kevlin Henney and Giovanni Asproni. “The code is the design” and “Simple code” are their mantras. While we must take care to remember that the interface is the program, and that its structures have much to say about our program structure, it is crucial to continuously adopt the humble stance that the design lives in the code. And while rework in the manufacturing metaphor leads to cost, rework in design leads to value. We should view our code as the beautiful articulation of noble efforts of design—design as a process, not a static endpoint. It’s in the code that the architectural metrics of coupling and cohesion play out. If you listen to Larry Constan-tine describe coupling and cohesion, he speaks in terms of code—not lofty abstract con-cepts that one might ﬁnd in UML. Richard Gabriel advises us in his essay, “Abstraction Descant” that abstraction is evil. Code is anti-evil, and clean code is perhaps divine.  
Tất nhiên, tôi vẫn là người ủng hộ suy nghĩ ở phạm vi rộng hơn và đặc biệt là giá trị của các phương pháp tiếp cận kiến trúc bắt nguồn từ kiến thức miền sâu và khả năng sử dụng phần mềm. Cuốn sách không nói về điều đó – hoặc, ít nhất, rõ ràng là không nói về điều đó. Cuốn sách này có một thông điệp tinh tế hơn mà sự sâu sắc của nó không nên được đánh giá thấp. Nó phù hợp với cách nhìn hiện tại của những người thực sự dựa trên mã như Peter Sommerlad, Kevlin Henney và Giovanni Asproni. “Mã là thiết kế” và “Mã đơn giản” là câu thần chú của họ. Mặc dù chúng ta phải lưu ý rằng giao diện chính là chương trình và cấu trúc của nó nói lên nhiều điều về cấu trúc chương trình của chúng ta, nhưng điều quan trọng là phải liên tục áp dụng quan điểm khiêm tốn rằng thiết kế nằm trong mã. Và trong khi việc làm lại trong ẩn dụ sản xuất dẫn đến chi phí, thì việc làm lại thiết kế dẫn đến giá trị. Chúng ta nên xem mã của mình là sự khớp nối tuyệt đẹp của những nỗ lực cao quý của thiết kế—thiết kế là một quá trình, không phải là một điểm cuối tĩnh. Chính trong mã mà các thước đo kiến trúc của khớp nối và sự gắn kết thể hiện. Nếu bạn nghe Larry Constantine mô tả về khớp nối và sự gắn kết, thì anh ấy nói về mã - không phải những khái niệm trừu tượng cao cả mà người ta có thể tìm thấy trong UML. Richard Gabriel khuyên chúng ta trong bài luận của anh ấy, “Sự suy giảm trừu tượng” rằng sự trừu tượng là xấu xa. Mã là chống lại cái ác, và mã sạch có lẽ là thần thánh.

Going back to my little box of Ga-Jol, I think it’s important to note that the Danish wisdom advises us not just to pay attention to small things, but also to be *honest* in small things. This means being honest to the code, honest to our colleagues about the state of our code and, most of all, being honest with ourselves about our code. Did we Do our Best to “leave the campground cleaner than we found it”? Did we re-factor our code before check-ing in? These are not peripheral concerns but concerns that lie squarely in the center of Agile values. It is a recommended practice in Scrum that re-factoring be part of the con-cept of “Done.” Neither architecture nor clean code insist on perfection, only on honesty and doing the best we can. *To err is human; to forgive, divine.* In Scrum, we make every-thing visible. We air our dirty laundry. We are honest about the state of our code because  
Quay trở lại với hộp Ga-Jol nhỏ của tôi, tôi nghĩ điều quan trọng cần lưu ý là trí tuệ Đan Mạch khuyên chúng ta không chỉ chú ý đến những điều nhỏ nhặt mà còn phải trung thực trong những điều nhỏ nhặt. Điều này có nghĩa là trung thực với mã, trung thực với đồng nghiệp về trạng thái mã của chúng tôi và trên hết, trung thực với chính chúng tôi về mã của chúng tôi. Chúng ta đã cố gắng hết sức để “làm cho khu cắm trại sạch sẽ hơn chúng ta thấy” chưa? Chúng tôi đã xác định lại mã của mình trước khi đăng ký chưa? Đây không phải là những mối quan tâm ngoại vi mà là những mối quan tâm nằm thẳng ở trung tâm của các giá trị Agile. Một thực tiễn được khuyến nghị trong Scrum là tái bao thanh toán là một phần của khái niệm “Hoàn thành”. Cả kiến trúc và mã sạch đều không đòi hỏi sự hoàn hảo, mà chỉ đòi hỏi sự trung thực và làm tốt nhất có thể. Sai lầm là con người; để tha thứ, thiêng liêng. Trong Scrum, chúng tôi hiển thị mọi thứ. Chúng tôi phơi quần áo bẩn của mình. Chúng tôi trung thực về trạng thái mã của chúng tôi bởi vì

<https://www.facebook.com/groups/phattrienlaptrinh>

**Foreword** xxiii  
Lời nói đầu xxiii

code is never perfect. We become more fully human, more worthy of the divine, and closer to that greatness in the details.  
mã không bao giờ hoàn hảo. Chúng ta trở thành con người trọn vẹn hơn, xứng đáng hơn với thần thánh và gần gũi hơn với sự vĩ đại đó đến từng chi tiết.

In our profession, we desperately need all the help we can get. If a clean shop ﬂoor reduces accidents, and well-organized shop tools increase productivity, then I’m all for them. As for this book, it is the best pragmatic application of Lean principles to software I have ever seen in print. I expected no less from this practical little group of thinking indi-viduals that has been striving together for years not only to become better, but also to gift their knowledge to the industry in works such as you now ﬁnd in your hands. It leaves the world a little better than I found it before Uncle Bob sent me the manuscript.  
Trong nghề nghiệp của chúng tôi, chúng tôi rất cần tất cả sự giúp đỡ mà chúng tôi có thể nhận được. Nếu sàn nhà sạch sẽ giúp giảm thiểu tai nạn và các công cụ trong cửa hàng được sắp xếp hợp lý giúp tăng năng suất, thì tôi hoàn toàn ủng hộ họ. Đối với cuốn sách này, nó là ứng dụng thực tế nhất của các nguyên tắc Lean cho phần mềm mà tôi từng thấy trên bản in. Tôi mong đợi không ít từ nhóm nhỏ thực tế này gồm những cá nhân có suy nghĩ, những người đã cùng nhau phấn đấu trong nhiều năm không chỉ để trở nên tốt hơn mà còn đóng góp kiến thức của họ cho ngành công nghiệp trong những tác phẩm như bạn đang tìm thấy trong tay. Nó rời khỏi thế giới tốt hơn một chút so với những gì tôi tìm thấy trước khi chú Bob gửi cho tôi bản thảo.

Having completed this exercise in lofty insights, I am off to clean my desk.  
Sau khi hoàn thành bài tập này với những hiểu biết sâu sắc, tôi đi dọn dẹp bàn làm việc của mình.

**James O. Coplien** Mørdrup, Denmark

<https://www.facebook.com/groups/phattrienlaptrinh>

*This page intentionally left blank*

<https://www.facebook.com/groups/phattrienlaptrinh>

[**Introduction**](#_page_37_0)  
Giới thiệu

Reproduced with the kind permission of Thom Holwerda. <http://www.osnews.com/story/19266/WTFs_m>  
Sao chép với sự cho phép của Thom Holwerda. http://www.osnews.com/story/19266/WTFs\_m

Which door represents your code? Which door represents your team or your company? Why are we in that room? Is this just a normal code review or have we found a stream of horrible problems shortly after going live? Are we debugging in a panic, poring over code that we thought worked? Are customers leaving in droves and managers breathing down  
Cửa nào đại diện cho mã của bạn? Cánh cửa nào đại diện cho nhóm của bạn hoặc công ty của bạn? Tại sao chúng ta lại ở trong căn phòng đó? Đây chỉ là một đánh giá mã thông thường hay chúng tôi đã tìm thấy một loạt các sự cố khủng khiếp ngay sau khi phát hành trực tuyến? Có phải chúng ta đang gỡ lỗi một cách hoảng loạn, nghiền ngẫm mã mà chúng ta nghĩ là hiệu quả? Có phải khách hàng bỏ đi hàng loạt và người quản lý thở dốc

xxv

<https://www.facebook.com/groups/phattrienlaptrinh>

xxvi **Introduction**  
xxviGiới thiệu

our necks? How can we make sure we wind up behind the *right* door when the going gets tough? The answer is: *craftsmanship*.  
cổ của chúng ta? Làm thế nào chúng ta có thể chắc chắn rằng mình sẽ đến đúng cánh cửa khi mọi việc trở nên khó khăn? Câu trả lời là: nghề thủ công.

There are two parts to learning craftsmanship: knowledge and work. You must gain the knowledge of principles, patterns, practices, and heuristics that a craftsman knows, and you must also grind that knowledge into your ﬁngers, eyes, and gut by working hard and practicing.  
Có hai phần để học nghề thủ công: kiến thức và công việc. Bạn phải đạt được kiến thức về các nguyên tắc, mô hình, thực hành và kinh nghiệm mà một người thợ thủ công biết, và bạn cũng phải nghiền nát kiến thức đó vào đầu ngón tay, đôi mắt và trái tim của mình bằng cách làm việc chăm chỉ và luyện tập.

I can teach you the physics of riding a bicycle. Indeed, the classical mathematics is relatively straightforward. Gravity, friction, angular momentum, center of mass, and so forth, can be demonstrated with less than a page full of equations. Given those formulae I could prove to you that bicycle riding is practical and give you all the knowledge you needed to make it work. And you’d still fall down the ﬁrst time you climbed on that bike.  
Tôi có thể dạy bạn vật lý của việc đi xe đạp. Thật vậy, toán học cổ điển tương đối đơn giản. Trọng lực, ma sát, động lượng góc, tâm khối lượng, v.v., có thể được chứng minh với chưa đầy một trang đầy phương trình. Với những công thức đó, tôi có thể chứng minh cho bạn thấy rằng đi xe đạp là thiết thực và cung cấp cho bạn tất cả kiến thức cần thiết để thực hiện việc này. Và bạn vẫn sẽ bị ngã trong lần đầu tiên leo lên chiếc xe đạp đó.

Coding is no different. We could write down all the “feel good” principles of clean code and then trust you to do the work (in other words, let you fall down when you get on the bike), but then what kind of teachers would that make us, and what kind of student would that make you?  
Mã hóa là không khác nhau. Chúng tôi có thể viết ra tất cả các nguyên tắc “cảm thấy tốt” về mã sạch và sau đó tin tưởng giao cho bạn thực hiện công việc (nói cách khác, để bạn ngã khi bạn lên xe đạp), nhưng sau đó, điều đó sẽ khiến chúng tôi trở thành loại giáo viên như thế nào, và loại sinh viên đó sẽ làm cho bạn?

No. That’s not the way this book is going to work.  
Không. Đó không phải là cách cuốn sách này sẽ hoạt động.

Learning to write clean code is *hard work*. It requires more than just the knowledge of principles and patterns. You must *sweat* over it. You must practice it yourself, and watch yourself fail. You must watch others practice it and fail. You must see them stumble and retrace their steps. You must see them agonize over decisions and see the price they pay for making those decisions the wrong way.  
Học cách viết mã sạch là một công việc khó khăn. Nó đòi hỏi nhiều hơn là kiến thức về các nguyên tắc và khuôn mẫu. Bạn phải đổ mồ hôi cho nó. Bạn phải tự mình thực hành và xem mình thất bại. Bạn phải xem những người khác thực hành nó và thất bại. Bạn phải thấy họ vấp ngã và lùi bước. Bạn phải thấy họ đau đớn trước những quyết định và thấy cái giá mà họ phải trả cho việc đưa ra những quyết định sai lầm đó.

Be prepared to work hard while reading this book. This is not a “feel good” book that you can read on an airplane and ﬁnish before you land. This book will make you work, *and work hard*. What kind of work will you be doing? You’ll be reading code—lots of code. And you will be challenged to think about what’s right about that code and what’s wrong with it. You’ll be asked to follow along as we take modules apart and put them back together again. This will take time and effort; but we think it will be worth it.  
Hãy sẵn sàng làm việc chăm chỉ khi đọc cuốn sách này. Đây không phải là một cuốn sách “dễ chịu” mà bạn có thể đọc trên máy bay và đọc xong trước khi hạ cánh. Cuốn sách này sẽ khiến bạn làm việc và làm việc chăm chỉ. Bạn sẽ làm công việc gì? Bạn sẽ đọc mã—rất nhiều mã. Và bạn sẽ được thử thách suy nghĩ về điều gì đúng về đoạn mã đó và điều gì sai với nó. Bạn sẽ được yêu cầu làm theo khi chúng tôi tách các mô-đun ra và ghép chúng lại với nhau. Điều này sẽ tốn thời gian và công sức; nhưng chúng tôi nghĩ rằng nó sẽ có giá trị nó.

We have divided this book into three parts. The ﬁrst several chapters describe the prin-ciples, patterns, and practices of writing clean code. There is quite a bit of code in these chapters, and they will be challenging to read. They’ll prepare you for the second section to come. If you put the book down after reading the ﬁrst section, good luck to you!  
Chúng tôi đã chia cuốn sách này thành ba phần. Một số chương đầu tiên mô tả các nguyên tắc, mẫu và thực hành viết mã sạch. Có khá nhiều mã trong các chương này và chúng sẽ khó đọc. Họ sẽ chuẩn bị cho bạn phần thứ hai sắp tới. Nếu bạn đặt cuốn sách xuống sau khi đọc phần đầu tiên, chúc bạn may mắn!

The second part of the book is the harder work. It consists of several case studies of ever-increasing complexity. Each case study is an exercise in cleaning up some code—of transforming code that has some problems into code that has fewer problems. The detail in this section is *intense*. You will have to ﬂip back and forth between the narrative and the code listings. You will have to analyze and understand the code we are working with and walk through our reasoning for making each change we make. Set aside some time because *this should take you days*.  
Phần thứ hai của cuốn sách là công việc khó khăn hơn. Nó bao gồm một số nghiên cứu điển hình về độ phức tạp ngày càng tăng. Mỗi nghiên cứu điển hình là một bài tập trong việc làm sạch một số mã—chuyển đổi mã có một số vấn đề thành mã có ít vấn đề hơn. Các chi tiết trong phần này là dữ dội. Bạn sẽ phải lật đi lật lại giữa tường thuật và danh sách mã. Bạn sẽ phải phân tích và hiểu mã mà chúng tôi đang làm việc và xem xét lý do của chúng tôi để thực hiện từng thay đổi mà chúng tôi thực hiện. Dành một chút thời gian vì điều này sẽ khiến bạn mất nhiều ngày.

The third part of this book is the payoff. It is a single chapter containing a list of heu-ristics and smells gathered while creating the case studies. As we walked through and cleaned up the code in the case studies, we documented every reason for our actions as a  
Phần thứ ba của cuốn sách này là phần thưởng. Đó là một chương duy nhất chứa danh sách các heu-ristic và mùi được thu thập trong khi tạo các nghiên cứu điển hình. Khi chúng tôi duyệt qua và làm sạch mã trong các nghiên cứu điển hình, chúng tôi đã ghi lại mọi lý do cho hành động của mình dưới dạng

<https://www.facebook.com/groups/phattrienlaptrinh>

**Introduction** xxvii  
Giới thiệu xxvii

heuristic or smell. We tried to understand our own reactions to the code we were reading and changing, and worked hard to capture why we felt what we felt and did what we did. The result is a knowledge base that desribes the way we think when we write, read, and clean code.  
heuristic hoặc mùi. Chúng tôi cố gắng hiểu phản ứng của chính mình đối với mã mà chúng tôi đang đọc và thay đổi, đồng thời làm việc chăm chỉ để nắm bắt lý do tại sao chúng tôi cảm thấy những gì chúng tôi cảm thấy và làm những gì chúng tôi đã làm. Kết quả là một cơ sở tri thức mô tả cách chúng ta nghĩ khi viết, đọc và làm sạch mã.

This knowledge base is of limited value if you don’t do the work of carefully reading through the case studies in the second part of this book. In those case studies we have care-fully annotated each change we made with forward references to the heuristics. These for-ward references appear in square brackets like this: [H22]. This lets you see the *context* in which those heuristics were applied and written! It is not the heuristics themselves that are so valuable, it is the *relationship between those heuristics and the discrete decisions we made while cleaning up the code in the case studies*.  
Cơ sở kiến thức này sẽ có giá trị hạn chế nếu bạn không đọc kỹ các nghiên cứu điển hình trong phần thứ hai của cuốn sách này. Trong những nghiên cứu điển hình đó, chúng tôi đã chú thích cẩn thận từng thay đổi mà chúng tôi đã thực hiện với các tham chiếu chuyển tiếp đến phương pháp phỏng đoán. Các tham chiếu chuyển tiếp này xuất hiện trong dấu ngoặc vuông như sau: [H22]. Điều này cho phép bạn thấy bối cảnh trong đó các phương pháp phỏng đoán đó được áp dụng và viết ra! Bản thân các phương pháp phỏng đoán không có giá trị như vậy, mà là mối quan hệ giữa các phương pháp phỏng đoán đó và các quyết định rời rạc mà chúng tôi đưa ra khi dọn dẹp mã trong các nghiên cứu điển hình.

To further help you with those relationships, we have placed a cross-reference at the end of the book that shows the page number for every forward reference. You can use it to look up each place where a certain heuristic was applied.  
Để tiếp tục giúp bạn với những mối quan hệ đó, chúng tôi đã đặt một tham chiếu chéo ở cuối cuốn sách hiển thị số trang cho mọi tham chiếu chuyển tiếp. Bạn có thể sử dụng nó để tra cứu từng nơi áp dụng một heuristic nhất định.

If you read the ﬁrst and third sections and skip over the case studies, then you will have read yet another “feel good” book about writing good software. But if you take the time to work through the case studies, following every tiny step, every minute decision—if you put yourself in our place, and force yourself to think along the same paths that we thought, then you will gain a much richer understanding of those principles, patterns, prac-tices, and heuristics. They won’t be “feel good” knowledge any more. They’ll have been ground into your gut, ﬁngers, and heart. They’ll have become part of you in the same way that a bicycle becomes an extension of your will when you have mastered how to ride it.  
Nếu bạn đọc phần đầu tiên và phần thứ ba và bỏ qua các nghiên cứu điển hình, thì bạn sẽ đọc thêm một cuốn sách “cảm thấy hài lòng” khác về việc viết phần mềm tốt. Nhưng nếu bạn dành thời gian để nghiên cứu các nghiên cứu điển hình, làm theo từng bước nhỏ, từng phút quyết định—nếu bạn đặt mình vào vị trí của chúng tôi và buộc mình phải suy nghĩ theo cùng những con đường mà chúng tôi đã nghĩ, thì bạn sẽ giàu có hơn nhiều. sự hiểu biết về các nguyên tắc, mô hình, thực hành và kinh nghiệm đó. Họ sẽ không còn là kiến thức “dễ chịu” nữa. Chúng sẽ ăn sâu vào ruột gan, ngón tay và trái tim của bạn. Chúng sẽ trở thành một phần của bạn giống như cách một chiếc xe đạp trở thành phần mở rộng ý chí của bạn khi bạn đã thành thạo cách lái nó.

**Acknowledgments  
Sự nhìn nhận**

**Artwork  
ảnh minh họa**

Thank you to my two artists, Jeniffer Kohnke and Angela Brooks. Jennifer is responsible for the stunning and creative pictures at the start of each chapter and also for the portraits of Kent Beck, Ward Cunningham, Bjarne Stroustrup, Ron Jeffries, Grady Booch, Dave Thomas, Michael Feathers, and myself.  
Xin cảm ơn hai nghệ sĩ của tôi, Jeniffer Kohnke và Angela Brooks. Jennifer chịu trách nhiệm về những bức ảnh tuyệt đẹp và sáng tạo ở đầu mỗi chương cũng như những bức chân dung của Kent Beck, Ward Cunningham, Bjarne Stroustrup, Ron Jeffries, Grady Booch, Dave Thomas, Michael Feathers và tôi.

Angela is responsible for the clever pictures that adorn the innards of each chapter. She has done quite a few pictures for me over the years, including many of the inside pic-tures in *Agile Software Develpment: Principles, Patterns, and Practices*. She is also my ﬁrstborn in whom I am well pleased.  
Angela chịu trách nhiệm về những hình ảnh thông minh tô điểm cho phần bên trong của mỗi chương. Cô ấy đã thực hiện khá nhiều bức tranh cho tôi trong nhiều năm, bao gồm nhiều bức tranh bên trong về Phát triển phần mềm linh hoạt: Nguyên tắc, Mẫu và Thực hành. Cô ấy cũng là đứa con đầu lòng của tôi và tôi rất hài lòng.

<https://www.facebook.com/groups/phattrienlaptrinh>

*This page intentionally left blank*

<https://www.facebook.com/groups/phattrienlaptrinh>

[**On the Cover**](#_page_37_0)  
Trên trang bìa

The image on the cover is M104: The Sombrero Galaxy. M104 is located in Virgo and is just under 30 million light-years from us. At it’s core is a supermassive black hole weigh-ing in at about a billion solar masses.  
Hình ảnh trên trang bìa là M104: The Sombrero Galaxy. M104 nằm ở Xử Nữ và chỉ cách chúng ta chưa đầy 30 triệu năm ánh sáng. Tại lõi của nó là một lỗ đen siêu lớn nặng khoảng một tỷ khối lượng mặt trời.

Does the image remind you of the explosion of the Klingon power moon *Praxis*? I vividly remember the scene in *Star Trek VI* that showed an equatorial ring of debris ﬂying away from that explosion. Since that scene, the equatorial ring has been a common artifact in sci-ﬁ movie explosions. It was even added to the explosion of Alderaan in later editions of the ﬁrst *Star Wars* movie.  
Hình ảnh có làm bạn nhớ đến vụ nổ của mặt trăng quyền lực Klingon Praxis không? Tôi nhớ rất rõ cảnh trong Star Trek VI cho thấy một vòng xích đạo gồm các mảnh vụn bay ra khỏi vụ nổ đó. Kể từ cảnh đó, vành đai xích đạo đã trở thành hiện vật phổ biến trong các vụ nổ trong phim khoa học viễn tưởng. Nó thậm chí còn được thêm vào vụ nổ của Alderaan trong các phiên bản sau của bộ phim Chiến tranh giữa các vì sao đầu tiên.

What caused this ring to form around M104? Why does it have such a huge central bulge and such a bright and tiny nucleus? It looks to me as though the central black hole lost its cool and blew a 30,000 light-year hole in the middle of the galaxy. Woe befell any civilizations that might have been in the path of that cosmic disruption.  
Điều gì đã khiến chiếc nhẫn này hình thành xung quanh M104? Tại sao nó lại có một chỗ phình to ở trung tâm và một hạt nhân nhỏ và sáng như vậy? Đối với tôi, có vẻ như lỗ đen trung tâm đã mất bình tĩnh và tạo ra một lỗ 30.000 năm ánh sáng ở giữa thiên hà. Khốn thay cho bất kỳ nền văn minh nào có thể nằm trong con đường của sự gián đoạn vũ trụ đó.

Supermassive black holes swallow whole stars for lunch, converting a sizeable frac-tion of their mass to energy. *E = MC*2 is leverage enough, but when *M* is a stellar mass: Look out! How many stars fell headlong into that maw before the monster was satiated? Could the size of the central void be a hint?  
Các lỗ đen siêu lớn nuốt chửng cả các ngôi sao để ăn trưa, biến một phần đáng kể khối lượng của chúng thành năng lượng. E = MC2 là đòn bẩy đủ, nhưng khi M là một khối sao: Coi chừng! Có bao nhiêu ngôi sao rơi thẳng vào cái bụng đó trước khi con quái vật được thỏa mãn? Kích thước của khoảng trống trung tâm có thể là một gợi ý?

The image of M104 on the cover is a combination of the famous visible light pho-tograph from Hubble (right), and the recent infrared image from the Spitzer orbiting observatory (below, right). It’s the infrared image that clearly shows us the ring nature of the galaxy. In visible light we only see the front edge of the ring in silhouette. The cen-tral bulge obscures the rest of the ring.  
Hình ảnh của M104 trên trang bìa là sự kết hợp giữa bức ảnh ánh sáng khả kiến nổi tiếng từ Hubble (phải) và hình ảnh hồng ngoại gần đây từ đài quan sát quỹ đạo Spitzer (bên dưới, bên phải). Đó là hình ảnh hồng ngoại cho chúng ta thấy rõ bản chất vòng của thiên hà. Trong ánh sáng khả kiến, chúng ta chỉ nhìn thấy cạnh trước của chiếc nhẫn trong hình bóng. Phần phình ra ở giữa che khuất phần còn lại của chiếc nhẫn.

But in the infrared, the hot particles in the ring shine through the central bulge. The two images combined give us a view we’ve not seen before and imply that long ago it was a raging inferno of activity.  
Nhưng trong tia hồng ngoại, các hạt nóng trong vòng chiếu qua chỗ phình trung tâm. Hai hình ảnh kết hợp cho chúng ta một cái nhìn mà chúng ta chưa từng thấy trước đây và ngụ ý rằng từ lâu nó là một địa ngục hoạt động dữ dội.

Cover image: © Spitzer Space Telescope

xxix

<https://www.facebook.com/groups/phattrienlaptrinh>

*This page intentionally left blank*

<https://www.facebook.com/groups/phattrienlaptrinh>



[**1**](#_page_37_0)

[**Clean Code**](#_page_37_0)  
Mã sạch

You are reading this book for two reasons. First, you are a programmer. Second, you want to be a better programmer. Good. We need better programmers.  
Bạn đang đọc cuốn sách này vì hai lý do. Đầu tiên, bạn là một lập trình viên. Thứ hai, bạn muốn trở thành một lập trình viên giỏi hơn. Tốt. Chúng tôi cần những lập trình viên giỏi hơn.

1

<https://www.facebook.com/groups/phattrienlaptrinh>

2 **Chapter 1: Clean Code**  
2 Chương 1: Code sạch

This is a book about good programming. It is ﬁlled with code. We are going to look at code from every different direction. We’ll look down at it from the top, up at it from the bottom, and through it from the inside out. By the time we are done, we’re going to know a lot about code. What’s more, we’ll be able to tell the difference between good code and bad code. We’ll know how to write good code. And we’ll know how to transform bad code into good code.  
Đây là một cuốn sách về lập trình hay. Nó chứa đầy mã. Chúng ta sẽ xem xét mã từ mọi hướng khác nhau. Chúng tôi sẽ xem xét nó từ trên xuống, nhìn từ dưới lên và xuyên suốt từ trong ra ngoài. Khi hoàn thành, chúng ta sẽ biết rất nhiều về mã. Hơn nữa, chúng ta sẽ có thể phân biệt giữa mã tốt và mã xấu. Chúng ta sẽ biết cách viết mã tốt. Và chúng ta sẽ biết cách biến mã xấu thành mã tốt.

[**There Will Be Code**](#_page_37_0)  
Sẽ Có Mã

One might argue that a book about code is somehow behind the times—that code is no longer the issue; that we should be concerned about models and requirements instead. Indeed some have suggested that we are close to the end of code. That soon all code will be generated instead of written. That programmers simply won’t be needed because busi-ness people will generate programs from speciﬁcations.  
Người ta có thể lập luận rằng một cuốn sách viết về mã bằng cách nào đó đã đi sau thời đại - mã đó không còn là vấn đề nữa; thay vào đó chúng ta nên quan tâm đến các mô hình và yêu cầu. Thật vậy, một số người đã gợi ý rằng chúng tôi đang ở gần cuối mã. Điều đó sẽ sớm tạo ra tất cả mã thay vì viết. Đơn giản là các lập trình viên đó sẽ không cần thiết vì những người kinh doanh sẽ tạo ra các chương trình từ các thông số kỹ thuật.

Nonsense! We will never be rid of code, because code represents the details of the requirements. At some level those details cannot be ignored or abstracted; they have to be speciﬁed. And specifying requirements in such detail that a machine can execute them *is programming*. Such a speciﬁcation *is code*.  
Vô lý! Chúng tôi sẽ không bao giờ loại bỏ mã, bởi vì mã đại diện cho các chi tiết của các yêu cầu. Ở một mức độ nào đó, những chi tiết đó không thể bị bỏ qua hoặc trừu tượng hóa; chúng phải được cụ thể hóa. Và việc chỉ định các yêu cầu một cách chi tiết để máy có thể thực hiện chúng là lập trình. Một đặc điểm kỹ thuật như vậy là mã.

I expect that the level of abstraction of our languages will continue to increase. I also expect that the number of domain-speciﬁc languages will continue to grow. This will be a good thing. But it will not eliminate code. Indeed, all the speciﬁcations written in these higher level and domain-speciﬁc language will *be* code! It will still need to be rigorous, accurate, and so formal and detailed that a machine can understand and execute it.  
Tôi hy vọng rằng mức độ trừu tượng của các ngôn ngữ của chúng ta sẽ tiếp tục tăng lên. Tôi cũng hy vọng rằng số lượng ngôn ngữ dành riêng cho miền sẽ tiếp tục tăng lên. Đây sẽ là một điều tốt. Nhưng nó sẽ không loại bỏ mã. Thật vậy, tất cả các thông số kỹ thuật được viết bằng ngôn ngữ cấp cao hơn và dành riêng cho miền này sẽ là mã! Nó vẫn cần phải chặt chẽ, chính xác, bài bản và chi tiết đến mức máy móc có thể hiểu và thực hiện được.

The folks who think that code will one day disappear are like mathematicians who hope one day to discover a mathematics that does not have to be formal. They are hoping that one day we will discover a way to create machines that can do what we want rather than what we say. These machines will have to be able to understand us so well that they can translate vaguely speciﬁed needs into perfectly executing programs that precisely meet those needs.  
Những người nghĩ rằng mã một ngày nào đó sẽ biến mất giống như các nhà toán học hy vọng một ngày nào đó sẽ khám phá ra một thứ toán học không cần phải hình thức. Họ hy vọng rằng một ngày nào đó chúng ta sẽ khám phá ra cách tạo ra những cỗ máy có thể làm những gì chúng ta muốn hơn là những gì chúng ta nói. Những cỗ máy này sẽ phải có khả năng hiểu chúng ta rõ đến mức chúng có thể chuyển những nhu cầu cụ thể mơ hồ thành những chương trình thực thi hoàn hảo đáp ứng chính xác những nhu cầu đó.

This will never happen. Not even humans, with all their intuition and creativity, have been able to create successful systems from the vague feelings of their customers. Indeed, if the discipline of requirements speciﬁcation has taught us anything, it is that well-speciﬁed requirements are as formal as code and can act as executable tests of that code!  
Điều này sẽ không bao giờ xảy ra. Ngay cả con người, với tất cả trực giác và sự sáng tạo của mình, cũng không thể tạo ra những hệ thống thành công từ những cảm nhận mơ hồ của khách hàng. Thật vậy, nếu nguyên tắc đặc tả yêu cầu đã dạy chúng ta bất cứ điều gì, thì đó chính là các yêu cầu được đặc tả rõ ràng cũng chính thức như mã và có thể đóng vai trò là các bài kiểm tra có thể thực thi được của mã đó!

Remember that code is really the language in which we ultimately express the require-ments. We may create languages that are closer to the requirements. We may create tools that help us parse and assemble those requirements into formal structures. But we will never eliminate necessary precision—so there will always be code.  
Hãy nhớ rằng mã thực sự là ngôn ngữ mà cuối cùng chúng ta thể hiện các yêu cầu. Chúng tôi có thể tạo ra các ngôn ngữ gần với yêu cầu hơn. Chúng tôi có thể tạo các công cụ giúp chúng tôi phân tích và tập hợp các yêu cầu đó thành các cấu trúc chính thức. Nhưng chúng tôi sẽ không bao giờ loại bỏ độ chính xác cần thiết—vì vậy sẽ luôn có mã.

<https://www.facebook.com/groups/phattrienlaptrinh>

**Bad Code** 3  
Mã Xấu 3

[**Bad Code**](#_page_37_0)  
mã xấu

I was recently reading the preface to Kent Beck’s book *Implementation Patterns.*1 He says, “. . . this book is based on a rather fragile premise: that good code matters. . . .” A *fragile* premise? I dis-agree! I think that premise is one of the most robust, supported, and overloaded of all the pre-mises in our craft (and I think Kent knows it). We know good code matters because we’ve had to deal for so long with its lack.  
Gần đây tôi đang đọc lời tựa cho cuốn sách của Kent Beck Các mẫu triển khai.1 Anh ấy nói, “. . . cuốn sách này dựa trên một tiền đề khá mong manh: đó là mã tốt. . . .” Một tiền đề mong manh? Tôi không đồng ý! Tôi nghĩ rằng tiền đề đó là một trong những tiền đề mạnh mẽ, được hỗ trợ và quá tải nhất trong tất cả các tiền đề trong nghề của chúng tôi (và tôi nghĩ Kent biết điều đó). Chúng tôi biết mã tốt rất quan trọng vì chúng tôi đã phải đối phó với sự thiếu sót của nó trong một thời gian dài.

I know of one company that, in the late 80s, wrote a *killer* app. It was very popular, and lots of professionals bought and used it. But then the release cycles began to stretch. Bugs were not repaired from one release to the next. Load times grew and crashes increased. I remember the day I shut the product down in frustration and never used it again. The company went out of business a short time after that.  
Tôi biết một công ty, vào cuối những năm 80, đã viết một ứng dụng sát thủ. Nó rất phổ biến, và rất nhiều chuyên gia đã mua và sử dụng nó. Nhưng sau đó chu kỳ phát hành bắt đầu kéo dài. Các lỗi không được sửa chữa từ bản phát hành này sang bản phát hành tiếp theo. Thời gian tải tăng lên và sự cố tăng lên. Tôi nhớ ngày tôi tắt sản phẩm trong sự thất vọng và không bao giờ sử dụng nó nữa. Công ty đã ngừng hoạt động một thời gian ngắn sau đó.

Two decades later I met one of the early employees of that company and asked him what had happened. The answer conﬁrmed my fears. They had rushed the product to market and had made a huge mess in the code. As they added more and more features, the code got worse and worse until they simply could not manage it any longer. *It was the bad code that brought the company down.*  
Hai thập kỷ sau, tôi gặp một trong những nhân viên đầu tiên của công ty đó và hỏi anh ấy chuyện gì đã xảy ra. Câu trả lời khẳng định nỗi sợ hãi của tôi. Họ đã vội vã đưa sản phẩm ra thị trường và đã tạo ra một mớ hỗn độn lớn trong mã nguồn. Khi họ thêm ngày càng nhiều tính năng, mã càng ngày càng tệ cho đến khi họ không thể quản lý nó được nữa. Chính mã xấu đã đưa công ty đi xuống.

Have *you* ever been signiﬁcantly impeded by bad code? If you are a programmer of any experience then you’ve felt this impediment many times. Indeed, we have a name for it. We call it *wading*. We wade through bad code. We slog through a morass of tangled brambles and hidden pitfalls. We struggle to ﬁnd our way, hoping for some hint, some clue, of what is going on; but all we see is more and more senseless code.  
Bạn đã bao giờ bị cản trở đáng kể bởi mã xấu chưa? Nếu bạn là một lập trình viên có kinh nghiệm thì bạn đã nhiều lần cảm thấy trở ngại này. Thật vậy, chúng tôi có một tên cho nó. Chúng tôi gọi đó là lội nước. Chúng tôi lội qua mã xấu. Chúng tôi vượt qua một đống bụi gai rối rắm và những cạm bẫy ẩn giấu. Chúng ta loay hoay tìm đường, hy vọng có một gợi ý nào đó, một manh mối nào đó về những gì đang diễn ra; nhưng tất cả những gì chúng ta thấy ngày càng là những đoạn mã vô nghĩa.

Of course you have been impeded by bad code. So then—why did you write it?  
Tất nhiên bạn đã bị cản trở bởi mã xấu. Vậy thì—tại sao bạn lại viết nó?

Were you trying to go fast? Were you in a rush? Probably so. Perhaps you felt that you didn’t have time to do a good job; that your boss would be angry with you if you took the time to clean up your code. Perhaps you were just tired of working on this program and wanted it to be over. Or maybe you looked at the backlog of other stuff that you had prom-ised to get done and realized that you needed to slam this module together so you could move on to the next. We’ve all done it.  
Bạn đã cố gắng để đi nhanh? Bạn có đang vội không? Có lẽ vậy. Có lẽ bạn cảm thấy rằng mình không có thời gian để làm tốt công việc; rằng sếp của bạn sẽ tức giận với bạn nếu bạn dành thời gian để dọn dẹp mã của mình. Có lẽ bạn đã quá mệt mỏi khi làm việc với chương trình này và muốn nó kết thúc. Hoặc có thể bạn đã xem xét công việc tồn đọng của những thứ khác mà bạn đã hứa sẽ hoàn thành và nhận ra rằng bạn cần kết hợp mô-đun này lại với nhau để có thể chuyển sang mô-đun tiếp theo. Tất cả chúng ta đã làm điều đó.

We’ve all looked at the mess we’ve just made and then have chosen to leave it for another day. We’ve all felt the relief of seeing our messy program work and deciding that a  
Tất cả chúng tôi đều xem xét mớ hỗn độn mà mình vừa tạo ra và sau đó quyết định để nó vào một ngày khác. Tất cả chúng tôi đều cảm thấy nhẹ nhõm khi thấy chương trình lộn xộn của mình hoạt động và quyết định rằng

1. [Beck07].

<https://www.facebook.com/groups/phattrienlaptrinh>

4 **Chapter 1: Clean Code**  
4 Chương 1: Code sạch

working mess is better than nothing. We’ve all said we’d go back and clean it up later. Of course, in those days we didn’t know LeBlanc’s law: *Later equals never*.  
làm việc lộn xộn vẫn tốt hơn là không có gì. Tất cả chúng tôi đã nói rằng chúng tôi sẽ quay lại và dọn dẹp nó sau. Tất nhiên, vào thời đó chúng ta không biết luật của LeBlanc: Sau này nghĩa là không bao giờ.

[**The Total Cost of Owning a Mess**](#_page_37_0)  
Tổng chi phí sở hữu một mớ hỗn độn

If you have been a programmer for more than two or three years, you have probably been signiﬁcantly slowed down by someone else’s messy code. If you have been a programmer for longer than two or three years, you have probably been slowed down by messy code. The degree of the slowdown can be signiﬁcant. Over the span of a year or two, teams that were moving very fast at the beginning of a project can ﬁnd themselves moving at a snail’s pace. Every change they make to the code breaks two or three other parts of the code. No change is trivial. Every addition or modiﬁcation to the system requires that the tangles, twists, and knots be “understood” so that more tangles, twists, and knots can be added. Over time the mess becomes so big and so deep and so tall, they can not clean it up. There is no way at all.  
Nếu bạn đã là một lập trình viên hơn hai hoặc ba năm, có lẽ bạn đã bị làm chậm đáng kể bởi mã lộn xộn của người khác. Nếu bạn đã là một lập trình viên lâu hơn hai hoặc ba năm, có lẽ bạn đã bị chậm lại bởi mã lộn xộn. Mức độ chậm lại có thể là đáng kể. Trong khoảng thời gian một hoặc hai năm, các nhóm đã tiến hành rất nhanh khi bắt đầu một dự án có thể thấy mình đang di chuyển với tốc độ của ốc sên. Mọi thay đổi họ thực hiện đối với mã sẽ phá vỡ hai hoặc ba phần khác của mã. Không có thay đổi nào là tầm thường. Mọi sự bổ sung hoặc sửa đổi đối với hệ thống đều yêu cầu các mớ rối, khúc xoắn và nút thắt phải được “hiểu” để có thể thêm vào nhiều khúc rối, khúc xoắn và nút thắt hơn. Theo thời gian, đống lộn xộn trở nên quá lớn, quá sâu và quá cao, họ không thể dọn dẹp được. Không có cách nào cả.

As the mess builds, the productivity of the team continues to decrease, asymptotically approaching zero. As productivity decreases, management does the only thing they can; they add more staff to the project in hopes of increasing productivity. But that new staff is not versed in the design of the system. They don’t know the difference between a change that matches the design intent and a change that thwarts the design intent. Furthermore, they, and everyone else on the team, are under horriﬁc pressure to increase productivity. So they all make more and more messes, driving the productivity ever further toward zero. (See Figure 1-1.)  
Khi tình trạng lộn xộn tăng lên, năng suất của nhóm tiếp tục giảm, tiến gần đến con số không. Khi năng suất giảm, quản lý làm điều duy nhất họ có thể; họ bổ sung thêm nhân viên cho dự án với hy vọng tăng năng suất. Nhưng nhân viên mới đó không thành thạo trong việc thiết kế hệ thống. Họ không biết sự khác biệt giữa thay đổi phù hợp với mục đích thiết kế và thay đổi cản trở mục đích thiết kế. Hơn nữa, họ và những người khác trong nhóm đang chịu áp lực khủng khiếp phải tăng năng suất. Vì vậy, tất cả họ ngày càng tạo ra nhiều mớ hỗn độn hơn, đẩy năng suất ngày càng tiến về con số không. (Xem Hình 1-1.)

**Figure 1-1** Productivity vs. time  
Hình 1-1 Năng suất so với thời gian

<https://www.facebook.com/groups/phattrienlaptrinh>

**The Total Cost of Owning a Mess** 5  
Tổng chi phí sở hữu một mớ hỗn độn 5

[**The Grand Redesign in the Sky**](#_page_37_0)  
Thiết kế lại vĩ đại trên bầu trời

Eventually the team rebels. They inform management that they cannot continue to develop in this odious code base. They demand a redesign. Management does not want to expend the resources on a whole new redesign of the project, but they cannot deny that productiv-ity is terrible. Eventually they bend to the demands of the developers and authorize the grand redesign in the sky.  
Cuối cùng đội nổi dậy. Họ thông báo cho ban quản lý rằng họ không thể tiếp tục phát triển trong cơ sở mã đáng ghét này. Họ yêu cầu thiết kế lại. Ban quản lý không muốn sử dụng các nguồn lực cho việc thiết kế lại toàn bộ dự án, nhưng họ không thể phủ nhận rằng năng suất là rất tệ. Cuối cùng, họ tuân theo yêu cầu của các nhà phát triển và cho phép thiết kế lại toàn bộ bầu trời.

A new tiger team is selected. Everyone wants to be on this team because it’s a green-ﬁeld project. They get to start over and create something truly beautiful. But only the best and brightest are chosen for the tiger team. Everyone else must continue to maintain the current system.  
Một đội hổ mới được chọn. Mọi người đều muốn tham gia vào nhóm này vì đó là một dự án xanh. Họ phải bắt đầu lại và tạo ra thứ gì đó thực sự đẹp đẽ. Nhưng chỉ những người giỏi nhất và thông minh nhất mới được chọn vào đội hổ. Mọi người khác phải tiếp tục duy trì hệ thống hiện tại.

Now the two teams are in a race. The tiger team must build a new system that does everything that the old system does. Not only that, they have to keep up with the changes that are continuously being made to the old system. Management will not replace the old system until the new system can do everything that the old system does.  
Bây giờ hai đội đang trong một cuộc đua. Nhóm hổ phải xây dựng một hệ thống mới làm mọi thứ mà hệ thống cũ làm. Không chỉ vậy, họ phải theo kịp những thay đổi liên tục được thực hiện đối với hệ thống cũ. Ban quản lý sẽ không thay thế hệ thống cũ cho đến khi hệ thống mới có thể làm mọi thứ mà hệ thống cũ làm.

This race can go on for a very long time. I’ve seen it take 10 years. And by the time it’s done, the original members of the tiger team are long gone, and the current members are demanding that the new system be redesigned because it’s such a mess.  
Cuộc đua này có thể diễn ra trong một thời gian rất dài. Tôi đã thấy nó mất 10 năm. Và khi nó hoàn thành, các thành viên ban đầu của đội hổ đã ra đi từ lâu, và các thành viên hiện tại đang yêu cầu thiết kế lại hệ thống mới vì nó quá lộn xộn.

If you have experienced even one small part of the story I just told, then you already know that spending time keeping your code clean is not just cost effective; it’s a matter of professional survival.  
Nếu bạn đã trải qua dù chỉ một phần nhỏ của câu chuyện tôi vừa kể, thì bạn đã biết rằng dành thời gian để giữ cho mã của bạn sạch sẽ không chỉ hiệu quả về mặt chi phí; đó là vấn đề sống còn trong nghề nghiệp.

[**Attitude**](#_page_37_0)  
Thái độ

Have you ever waded through a mess so grave that it took weeks to do what should have taken hours? Have you seen what should have been a one-line change, made instead in hundreds of different modules? These symptoms are all too common.  
Bạn đã bao giờ lội qua một mớ hỗn độn nghiêm trọng đến mức phải mất hàng tuần để làm những việc lẽ ra phải mất hàng giờ chưa? Bạn đã thấy những thay đổi đáng lẽ phải là một dòng, thay vào đó được thực hiện trong hàng trăm mô-đun khác nhau chưa? Những triệu chứng này đều quá phổ biến.

Why does this happen to code? Why does good code rot so quickly into bad code? We have lots of explanations for it. We complain that the requirements changed in ways that thwart the original design. We bemoan the schedules that were too tight to do things right. We blather about stupid managers and intolerant customers and useless marketing types and telephone sanitizers. But the fault, dear Dilbert, is not in our stars, but in ourselves. We are unprofessional.  
Tại sao điều này xảy ra với mã? Tại sao mã tốt lại nhanh chóng biến thành mã xấu? Chúng tôi có rất nhiều lời giải thích cho nó. Chúng tôi phàn nàn rằng các yêu cầu đã thay đổi theo cách cản trở thiết kế ban đầu. Chúng tôi phàn nàn về lịch trình quá chặt chẽ để làm mọi việc đúng. Chúng tôi ba hoa về những người quản lý ngu ngốc và những khách hàng không khoan dung cũng như các kiểu tiếp thị vô dụng và chất khử trùng điện thoại. Nhưng lỗi, Dilbert thân mến, không phải ở các vì sao của chúng ta, mà ở chính chúng ta. Chúng tôi không chuyên nghiệp.

This may be a bitter pill to swallow. How could this mess be *our* fault? What about the requirements? What about the schedule? What about the stupid managers and the useless marketing types? Don’t they bear some of the blame?  
Đây có thể là một viên thuốc đắng để nuốt. Làm thế nào mớ hỗn độn này có thể là lỗi của chúng tôi? Điều gì về các yêu cầu? Còn lịch trình thì sao? Còn những người quản lý ngu ngốc và các loại tiếp thị vô dụng thì sao? Họ không chịu một số trách nhiệm sao?

No. The managers and marketers look to *us* for the information they need to make promises and commitments; and even when they don’t look to us, we should not be shy about telling them what we think. The users look to us to validate the way the requirements will ﬁt into the system. The project managers look to us to help work out the schedule. We  
Không. Các nhà quản lý và nhà tiếp thị tìm đến chúng tôi để biết thông tin họ cần để đưa ra lời hứa và cam kết; và ngay cả khi họ không để ý đến chúng ta, chúng ta cũng không nên ngại nói cho họ biết suy nghĩ của mình. Người dùng tìm đến chúng tôi để xác nhận cách các yêu cầu sẽ phù hợp với hệ thống. Các nhà quản lý dự án tìm đến chúng tôi để giúp lập kế hoạch. Chúng tôi

<https://www.facebook.com/groups/phattrienlaptrinh>

6 **Chapter 1: Clean Code**  
6 Chương 1: Code sạch

are deeply complicit in the planning of the project and share a great deal of the responsi-bility for any failures; especially if those failures have to do with bad code!  
đồng lõa sâu sắc trong việc lập kế hoạch của dự án và chia sẻ rất nhiều trách nhiệm đối với bất kỳ thất bại nào; đặc biệt nếu những lỗi đó liên quan đến mã xấu!

“But wait!” you say. “If I don’t do what my manager says, I’ll be ﬁred.” Probably not. Most managers want the truth, even when they don’t act like it. Most managers want good code, even when they are obsessing about the schedule. They may defend the schedule and requirements with passion; but that’s their job. It’s *your* job to defend the code with equal passion.  
“Nhưng đợi đã!” bạn nói. “Nếu tôi không làm theo những gì người quản lý của tôi nói, tôi sẽ bị sa thải.” Chắc là không. Hầu hết các nhà quản lý đều muốn sự thật, ngay cả khi họ không hành động như vậy. Hầu hết các nhà quản lý đều muốn viết mã tốt, ngay cả khi họ đang bị ám ảnh bởi lịch trình. Họ có thể bảo vệ lịch trình và các yêu cầu một cách say mê; nhưng đó là công việc của họ. Công việc của bạn là bảo vệ mã với niềm đam mê bình đẳng.

To drive this point home, what if you were a doctor and had a patient who demanded that you stop all the silly hand-washing in preparation for surgery because it was taking too much time?2 Clearly the patient is the boss; and yet the doctor should absolutely refuse to comply. Why? Because the doctor knows more than the patient about the risks of dis-ease and infection. It would be unprofessional (never mind criminal) for the doctor to comply with the patient.  
Để hiểu rõ điểm này, điều gì sẽ xảy ra nếu bạn là một bác sĩ và có một bệnh nhân yêu cầu bạn ngừng ngay việc rửa tay ngớ ngẩn để chuẩn bị cho cuộc phẫu thuật vì nó mất quá nhiều thời gian?2 Rõ ràng bệnh nhân là ông chủ; và bác sĩ nên tuyệt đối từ chối tuân thủ. Tại sao? Vì bác sĩ biết nhiều hơn bệnh nhân về nguy cơ bệnh tật và nhiễm trùng. Sẽ là không chuyên nghiệp (đừng nói đến tội phạm) nếu bác sĩ tuân theo bệnh nhân.

So too it is unprofessional for programmers to bend to the will of managers who don’t understand the risks of making messes.  
Vì vậy, thật thiếu chuyên nghiệp khi các lập trình viên chiều theo ý muốn của những người quản lý không hiểu những rủi ro của việc tạo ra những mớ hỗn độn.

[**The Primal Conundrum**](#_page_37_0)  
Câu hỏi hóc búa cơ bản

Programmers face a conundrum of basic values. All developers with more than a few years experience know that previous messes slow them down. And yet all developers feel the pressure to make messes in order to meet deadlines. In short, they don’t take the time to go fast!  
Các lập trình viên phải đối mặt với một câu hỏi hóc búa về các giá trị cơ bản. Tất cả các nhà phát triển có hơn một vài năm kinh nghiệm đều biết rằng những mớ hỗn độn trước đây làm họ chậm lại. Tuy nhiên, tất cả các nhà phát triển đều cảm thấy áp lực phải làm bừa bộn để đáp ứng thời hạn. Nói tóm lại, họ không dành thời gian để đi nhanh!

True professionals know that the second part of the conundrum is wrong. You will *not* make the deadline by making the mess. Indeed, the mess will slow you down instantly, and will force you to miss the deadline. The *only* way to make the deadline—the only way to go fast—is to keep the code as clean as possible at all times.  
Các chuyên gia chân chính biết rằng phần thứ hai của câu hỏi hóc búa là sai. Bạn sẽ không hoàn thành deadline bằng cách làm bừa bộn. Thật vậy, mớ hỗn độn sẽ làm bạn chậm lại ngay lập tức và buộc bạn phải bỏ lỡ thời hạn. Cách duy nhất để hoàn thành đúng thời hạn—cách duy nhất để tiến hành nhanh chóng—là luôn giữ mã sạch nhất có thể.

[**The Art of Clean Code?**](#_page_37_0)  
Nghệ thuật của mã sạch?

Let’s say you believe that messy code is a signiﬁcant impediment. Let’s say that you accept that the only way to go fast is to keep your code clean. Then you must ask yourself: “How do I write clean code?” It’s no good trying to write clean code if you don’t know what it means for code to be clean!  
Giả sử bạn tin rằng mã lộn xộn là một trở ngại đáng kể. Giả sử bạn chấp nhận rằng cách duy nhất để chạy nhanh là giữ cho mã của bạn sạch sẽ. Sau đó, bạn phải tự hỏi: “Làm cách nào để viết mã sạch?” Sẽ không tốt nếu bạn cố gắng viết mã sạch nếu bạn không biết ý nghĩa của mã sạch!

The bad news is that writing clean code is a lot like painting a picture. Most of us know when a picture is painted well or badly. But being able to recognize good art from bad does not mean that we know how to paint. So too being able to recognize clean code from dirty code does not mean that we know how to write clean code!  
Tin xấu là viết mã sạch cũng giống như vẽ một bức tranh. Hầu hết chúng ta đều biết khi nào một bức tranh được vẽ đẹp hay xấu. Nhưng có thể nhận ra nghệ thuật tốt từ xấu không có nghĩa là chúng ta biết cách vẽ. Vì vậy, có thể nhận ra mã sạch từ mã bẩn không có nghĩa là chúng ta biết cách viết mã sạch!

2. When hand-washing was ﬁrst recommended to physicians by Ignaz Semmelweis in 1847, it was rejected on the basis that doctors were too busy and wouldn’t have time to wash their hands between patient visits.  
2. Khi Ignaz Semmelweis lần đầu tiên khuyến nghị các bác sĩ rửa tay vào năm 1847, nó đã bị từ chối với lý do các bác sĩ quá bận rộn và không có thời gian rửa tay giữa các lần thăm khám cho bệnh nhân.

<https://www.facebook.com/groups/phattrienlaptrinh>

**The Total Cost of Owning a Mess** 7  
Tổng chi phí sở hữu một mớ hỗn độn 7

Writing clean code requires the disciplined use of a myriad little techniques applied through a painstakingly acquired sense of “cleanliness.” This “code-sense” is the key. Some of us are born with it. Some of us have to ﬁght to acquire it. Not only does it let us see whether code is good or bad, but it also shows us the strategy for applying our disci-pline to transform bad code into clean code.  
Viết mã sạch yêu cầu sử dụng có kỷ luật vô số kỹ thuật nhỏ được áp dụng thông qua cảm giác “sạch sẽ” mà bạn đã dày công đạt được. “Cảm giác mã” này là chìa khóa. Một số người trong chúng ta được sinh ra với nó. Một số người trong chúng ta phải đấu tranh để có được nó. Nó không chỉ cho chúng ta biết mã tốt hay xấu mà còn cho chúng ta thấy chiến lược áp dụng kỷ luật của mình để biến mã xấu thành mã sạch.

A programmer without “code-sense” can look at a messy module and recognize the mess but will have no idea what to do about it. A programmer *with* “code-sense” will look at a messy module and see options and variations. The “code-sense” will help that pro-grammer choose the best variation and guide him or her to plot a sequence of behavior preserving transformations to get from here to there.  
Một lập trình viên không có “cảm giác mã” có thể nhìn vào một mô-đun lộn xộn và nhận ra mớ hỗn độn đó nhưng sẽ không biết phải làm gì với nó. Một lập trình viên có “cảm giác mã” sẽ nhìn vào một mô-đun lộn xộn và thấy các tùy chọn và biến thể. “Cảm giác về mã” sẽ giúp lập trình viên đó chọn biến thể tốt nhất và hướng dẫn anh ta hoặc cô ta vẽ sơ đồ một chuỗi các hành vi bảo tồn các phép biến đổi để đi từ đây đến đó.

In short, a programmer who writes clean code is an artist who can take a blank screen through a series of transformations until it is an elegantly coded system.  
Nói tóm lại, một lập trình viên viết mã sạch là một nghệ sĩ có thể biến một màn hình trống qua một loạt các phép biến đổi cho đến khi nó trở thành một hệ thống được mã hóa trang nhã.

[**What Is Clean Code?**](#_page_37_0)  
Mã sạch là gì?

There are probably as many deﬁnitions as there are programmers. So I asked some very well-known and deeply experienced programmers what they thought.  
Có thể có nhiều định nghĩa như có nhiều lập trình viên. Vì vậy, tôi đã hỏi một số lập trình viên rất nổi tiếng và có kinh nghiệm sâu sắc về suy nghĩ của họ.

**Bjarne Stroustrup, inventor of C++ and author of *The C++ Programming Language*  
Bjarne Stroustrup, người phát minh ra C++ và là tác giả của Ngôn ngữ lập trình C++**

*I like my code to be elegant and efﬁcient. The logic should be straightforward to make it hard for bugs to hide, the dependencies minimal to ease maintenance, error handling complete according to an articulated strategy, and per-formance close to optimal so as not to tempt people to make the code messy with unprinci-pled optimizations. Clean code does one thing well.  
Tôi thích mã của mình thanh lịch và hiệu quả. Logic phải đơn giản để làm cho các lỗi khó che giấu, các yếu tố phụ thuộc ở mức tối thiểu để dễ bảo trì, xử lý lỗi hoàn chỉnh theo một chiến lược rõ ràng và hiệu suất gần với mức tối ưu để không khiến mọi người làm cho mã lộn xộn với các nguyên tắc không chính xác. -Pled tối ưu hóa. Clean code làm tốt một việc.*

Bjarne uses the word “elegant.” That’s quite a word! The dictionary in my MacBook® provides the following deﬁnitions: *pleasingly*  
Bjarne sử dụng từ “thanh lịch”. Đó là một từ khá! Từ điển trong MacBook® của tôi cung cấp các định nghĩa sau:

*graceful and stylish in appearance or manner; pleasingly ingenious and simple.* Notice the emphasis on the word “pleasing.” Apparently Bjarne thinks that clean code is *pleasing* to read. Reading it should make you smile the way a well-crafted music box or well-designed car would.  
duyên dáng và phong cách về ngoại hình hoặc phong cách; một cách dễ chịu khéo léo và đơn giản. Lưu ý sự nhấn mạnh vào từ “làm hài lòng”. Rõ ràng Bjarne nghĩ rằng mã sạch sẽ dễ đọc. Đọc nó sẽ khiến bạn mỉm cười giống như một chiếc hộp nhạc được làm thủ công hay một chiếc ô tô được thiết kế đẹp mắt.

Bjarne also mentions efﬁciency—*twice*. Perhaps this should not surprise us coming from the inventor of C++; but I think there’s more to it than the sheer desire for speed. Wasted cycles are inelegant, they are not pleasing. And now note the word that Bjarne uses  
Bjarne cũng đề cập đến tính hiệu quả - hai lần. Có lẽ điều này không làm chúng ta ngạc nhiên khi đến từ người phát minh ra C++; nhưng tôi nghĩ có nhiều thứ hơn là mong muốn tốc độ tuyệt đối. Chu kỳ lãng phí là không thanh lịch, họ không hài lòng. Và bây giờ hãy lưu ý từ mà Bjarne sử dụng

<https://www.facebook.com/groups/phattrienlaptrinh>

8 **Chapter 1: Clean Code**  
8 Chương 1: Code sạch

to describe the consequence of that inelegance. He uses the word “tempt.” There is a deep truth here. Bad code *tempts* the mess to grow! When others change bad code, they tend to make it worse.  
để diễn tả hậu quả của sự thiếu thanh lịch đó. Anh ấy dùng từ “cám dỗ”. Có một sự thật sâu sắc ở đây. Mã xấu cám dỗ sự lộn xộn phát triển! Khi những người khác thay đổi mã xấu, họ có xu hướng làm cho nó tệ hơn.

Pragmatic DaveThomas and Andy Hunt said this a different way.They used the meta-phor of broken windows.3 A building with broken windows looks like nobody cares about it. So other people stop caring. They allow more windows to become broken. Eventually they actively break them. They despoil the facade with grafﬁti and allow garbage to col-lect. One broken window starts the process toward decay.  
DaveThomas và Andy Hunt theo chủ nghĩa thực dụng lại nói điều này theo một cách khác. Họ đã sử dụng thuật ngữ meta-phor về cửa sổ vỡ.3 Một tòa nhà có cửa sổ vỡ trông giống như không ai quan tâm đến nó. Vì vậy, những người khác ngừng quan tâm. Chúng cho phép nhiều cửa sổ bị vỡ hơn. Cuối cùng họ chủ động phá vỡ chúng. Họ phá hỏng mặt tiền bằng những bức vẽ bậy và cho phép thu gom rác. Một cửa sổ bị vỡ bắt đầu quá trình phân rã.

Bjarne also mentions that error handing should be complete. This goes to the disci-pline of paying attention to details. Abbreviated error handling is just one way that pro-grammers gloss over details. Memory leaks are another, race conditions still another. Inconsistent naming yet another. The upshot is that clean code exhibits close attention to detail.  
Bjarne cũng đề cập rằng việc xử lý lỗi phải hoàn tất. Điều này dẫn đến kỷ luật chú ý đến chi tiết. Xử lý lỗi viết tắt chỉ là một cách mà các nhà ngữ pháp chuyên nghiệp chú ý đến các chi tiết. Rò rỉ bộ nhớ là khác, điều kiện cuộc đua vẫn còn khác. Đặt tên không nhất quán khác. Kết quả cuối cùng là mã sạch thể hiện sự chú ý chặt chẽ đến từng chi tiết.

Bjarne closes with the assertion that clean code does one thing well. It is no accident that there are so many principles of software design that can be boiled down to this simple admonition. Writer after writer has tried to communicate this thought. Bad code tries to do too much, it has muddled intent and ambiguity of purpose. Clean code is *focused*. Each function, each class, each module exposes a single-minded attitude that remains entirely undistracted, and unpolluted, by the surrounding details.  
Bjarne kết thúc với khẳng định rằng mã sạch sẽ làm tốt một việc. Không phải ngẫu nhiên mà có rất nhiều nguyên tắc thiết kế phần mềm có thể được rút gọn thành lời khuyên đơn giản này. Hết nhà văn này đến nhà văn khác đã cố gắng truyền đạt tư tưởng này. Mã xấu cố gắng làm quá nhiều, nó có mục đích lộn xộn và mục đích mơ hồ. Mã sạch được tập trung. Mỗi chức năng, mỗi lớp, mỗi mô-đun thể hiện một thái độ tập trung hoàn toàn không bị phân tâm và không bị ô nhiễm bởi các chi tiết xung quanh.

**Grady Booch, author of *Object Oriented Analysis and Design with Applications*  
Grady Booch, tác giả của Phân tích và thiết kế hướng đối tượng với các ứng dụng**

*Clean code is simple and direct. Clean code reads like well-written prose. Clean code never obscures the designer’s intent but rather is full of crisp abstractions and straightforward lines of control.  
Mã sạch là đơn giản và trực tiếp. Mã sạch đọc như văn xuôi được viết tốt. Mã sạch không bao giờ che khuất ý định của nhà thiết kế mà thay vào đó chứa đầy những phần trừu tượng rõ ràng và các dòng điều khiển đơn giản.*

Grady makes some of the same points as Bjarne, but he takes a *readability* perspective. I especially like his view that clean code should read like well-written prose. Think back on a  
Grady đưa ra một số điểm giống như Bjarne, nhưng anh ấy có quan điểm dễ đọc. Tôi đặc biệt thích quan điểm của anh ấy rằng mã sạch nên đọc như văn xuôi được viết tốt. Nghĩ lại về một

really good book that you’ve read. Remember how the words disappeared to be replaced by images! It was like watching a movie, wasn’t it? Better! You saw the characters, you heard the sounds, you experienced the pathos and the humor.  
cuốn sách thực sự hay mà bạn đã đọc. Hãy nhớ làm thế nào các từ biến mất để được thay thế bằng hình ảnh! Nó giống như xem một bộ phim, phải không? Tốt hơn! Bạn đã nhìn thấy các nhân vật, bạn đã nghe thấy âm thanh, bạn đã trải nghiệm những điều đáng tiếc và sự hài hước.

Reading clean code will never be quite like reading *Lord of the Rings*. Still, the liter-ary metaphor is not a bad one. Like a good novel, clean code should clearly expose the ten-sions in the problem to be solved. It should build those tensions to a climax and then give  
Đọc mã sạch sẽ không bao giờ giống như đọc Chúa tể của những chiếc nhẫn. Tuy nhiên, ẩn dụ văn học không phải là một ẩn dụ xấu. Giống như một cuốn tiểu thuyết hay, mã sạch sẽ phơi bày rõ ràng những điểm mấu chốt trong vấn đề cần giải quyết. Nó nên xây dựng những căng thẳng đó lên đến đỉnh điểm và sau đó đưa ra

3. [http://www.pragmaticprogrammer.com/booksellers/2004-12.html](http://www.pragmaticprogrammer.com/booksellers/2004-12.html%20///http://www.pragmaticprogrammer.com/booksellers/2004-12.html\\\)  
3. http://www.pragmaticprogrammer.com/bookeller/2004-12.html

<https://www.facebook.com/groups/phattrienlaptrinh>

**The Total Cost of Owning a Mess** 9  
Tổng chi phí sở hữu một mớ hỗn độn 9

the reader that “Aha! Of course!” as the issues and tensions are resolved in the revelation of an obvious solution.  
người đọc rằng “Aha! Tất nhiên rồi!" khi các vấn đề và căng thẳng được giải quyết trong sự tiết lộ về một giải pháp rõ ràng.

I ﬁnd Grady’s use of the phrase “crisp abstraction” to be a fascinating oxymoron! After all the word “crisp” is nearly a synonym for “concrete.” My MacBook’s dictionary holds the following deﬁnition of “crisp”: *briskly decisive and matter-of-fact, without hesi-tation or unnecessary detail.* Despite this seeming juxtaposition of meaning, the words carry a powerful message. Our code should be matter-of-fact as opposed to speculative. It should contain only what is necessary. Our readers should perceive us to have been decisive.  
Tôi thấy việc Grady sử dụng cụm từ “sự trừu tượng sắc nét” là một nghịch lý thú vị! Xét cho cùng, từ “sắc nét” gần như đồng nghĩa với “bê tông”. Từ điển MacBook của tôi giữ định nghĩa sau đây về "sắc nét": quyết đoán nhanh chóng và thực tế, không do dự hoặc chi tiết không cần thiết. Mặc dù ý nghĩa dường như nằm cạnh nhau, nhưng các từ này mang một thông điệp mạnh mẽ. Mã của chúng tôi phải là vấn đề thực tế thay vì suy đoán. Nó chỉ nên chứa những gì cần thiết. Độc giả của chúng tôi nên nhận thấy chúng tôi đã được quyết định.

**“Big” Dave Thomas, founder of OTI, godfather of the Eclipse strategy  
“Ông lớn” Dave Thomas, người sáng lập OTI, cha đỡ đầu của chiến lược Eclipse**

*Clean code can be read, and enhanced by a developer other than its original author. It has unit and acceptance tests. It has meaningful names. It provides one way rather than many ways for doing one thing. It has minimal depen-dencies, which are explicitly deﬁned, and pro-vides a clear and minimal API. Code should be literate since depending on the language, not all necessary information can be expressed clearly in code alone.  
Mã sạch có thể được đọc và được cải tiến bởi một nhà phát triển không phải là tác giả ban đầu của nó. Nó có các bài kiểm tra đơn vị và chấp nhận. Nó có tên có ý nghĩa. Nó cung cấp một cách chứ không phải nhiều cách để làm một việc. Nó có các phụ thuộc tối thiểu, được xác định rõ ràng và cung cấp một API rõ ràng và tối thiểu. Mã phải được biết chữ vì tùy thuộc vào ngôn ngữ, không phải tất cả thông tin cần thiết đều có thể được thể hiện rõ ràng chỉ bằng mã.*

Big Dave shares Grady’s desire for readabil-ity, but with an important twist. Dave asserts that  
Big Dave chia sẻ mong muốn của Grady về tính dễ đọc, nhưng với một bước ngoặt quan trọng. Dave khẳng định rằng

clean code makes it easy for *other* people to enhance it. This may seem obvious, but it can-not be overemphasized. There is, after all, a difference between code that is easy to read and code that is easy to change.  
mã sạch sẽ giúp người khác dễ dàng nâng cao nó. Điều này có vẻ hiển nhiên, nhưng nó không thể được nhấn mạnh quá mức. Rốt cuộc, có một sự khác biệt giữa mã dễ đọc và mã dễ thay đổi.

Dave ties cleanliness to tests! Ten years ago this would have raised a lot of eyebrows. But the discipline of Test Driven Development has made a profound impact upon our industry and has become one of our most fundamental disciplines. Dave is right. Code, without tests, is not clean. No matter how elegant it is, no matter how readable and acces-sible, if it hath not tests, it be unclean.  
Dave ràng buộc sự sạch sẽ với các bài kiểm tra! Mười năm trước, điều này sẽ khiến nhiều người nhướn mày. Nhưng nguyên tắc Phát triển dựa trên thử nghiệm đã có tác động sâu sắc đến ngành của chúng tôi và đã trở thành một trong những nguyên tắc cơ bản nhất của chúng tôi. Dave nói đúng. Mã, không có bài kiểm tra, không sạch. Cho dù nó có thanh lịch đến đâu, cho dù nó có thể đọc và truy cập được như thế nào, nếu nó không được kiểm tra, thì nó không trong sạch.

Dave uses the word *minimal* twice. Apparently he values code that is small, rather than code that is large. Indeed, this has been a common refrain throughout software litera-ture since its inception. Smaller is better.  
Dave sử dụng từ tối thiểu hai lần. Rõ ràng anh ấy coi trọng mã nhỏ hơn là mã lớn. Thật vậy, đây là một điệp khúc phổ biến trong toàn bộ tài liệu phần mềm kể từ khi thành lập. Nhỏ hơn là tốt hơn.

Dave also says that code should be *literate*. This is a soft reference to Knuth’s *literate programming.*4 The upshot is that the code should be composed in such a form as to make it readable by humans.  
Dave cũng nói rằng mã phải biết chữ. Đây là một tham chiếu mềm đến chương trình biết chữ của Knuth.4 Kết quả cuối cùng là mã nên được soạn ở dạng sao cho con người có thể đọc được.

4. [Knuth92].

<https://www.facebook.com/groups/phattrienlaptrinh>

10 **Chapter 1: Clean Code**  
10 Chương 1: Code sạch

**Michael Feathers, author of *Working Effectively with Legacy Code***  
Michael Feathers, tác giả của Làm việc hiệu quả với Mã kế thừa

*I could list all of the qualities that I notice in clean code, but there is one overarching quality that leads to all of them. Clean code always looks like it was written by someone who cares. There is nothing obvious that you can do to make it better. All of those things were thought about by the code’s author, and if you try to imagine improvements, you’re led back to where you are, sitting in appreciation of the code someone left for you—code left by some-one who cares deeply about the craft.  
Tôi có thể liệt kê tất cả các phẩm chất mà tôi nhận thấy trong mã sạch, nhưng có một phẩm chất bao trùm dẫn đến tất cả chúng. Mã sạch luôn có vẻ như được viết bởi một người quan tâm. Không có gì rõ ràng mà bạn có thể làm để làm cho nó tốt hơn. Tất cả những điều đó đã được tác giả của mã nghĩ đến, và nếu bạn cố gắng tưởng tượng ra những cải tiến, bạn sẽ được dẫn trở lại vị trí của mình, ngồi đó để đánh giá cao mã mà ai đó để lại cho bạn—mã do một người nào đó quan tâm sâu sắc để lại về nghề.*

One word: care. That’s really the topic of this book. Perhaps an appropriate subtitle would be *How to Care for Code*.  
Một từ: quan tâm. Đó thực sự là chủ đề của cuốn sách này. Có lẽ một phụ đề thích hợp sẽ là How to Care for Code.

Michael hit it on the head. Clean code is  
Michael đánh nó vào đầu. Mã sạch là

code that has been taken care of. Someone has taken the time to keep it simple and orderly. They have paid appropriate attention to details. They have cared.  
mã đã được chăm sóc. Ai đó đã dành thời gian để giữ cho nó đơn giản và có trật tự. Họ đã chú ý thích hợp đến các chi tiết. Họ đã quan tâm.

**Ron Jeffries, author of *Extreme Programming Installed* and *Extreme Programming Adventures in C#*  
Ron Jeffries, tác giả của Extreme Programming Installed và Extreme Programming Adventures in C#**

Ron began his career programming in Fortran at the Strategic Air Command and has written code in almost every language and on almost every machine. It pays to consider his words carefully.  
Ron bắt đầu sự nghiệp lập trình của mình ở Fortran tại Bộ Tư lệnh Không quân Chiến lược và đã viết mã bằng hầu hết mọi ngôn ngữ và trên hầu hết mọi máy. Nó trả tiền để xem xét lời nói của mình một cách cẩn thận.

*In recent years I begin, and nearly end, with Beck’s rules of simple code. In priority order, simple code:  
Trong những năm gần đây, tôi bắt đầu và gần như kết thúc với các quy tắc viết mã đơn giản của Beck. Theo thứ tự ưu tiên, mã đơn giản:*

*• Runs all the tests;  
• Chạy tất cả các bài kiểm tra;*

*• Contains no duplication;  
• Không chứa bản sao;*

*• Expresses all the design ideas that are in the system;  
• Thể hiện tất cả các ý tưởng thiết kế có trong hệ thống;*

*• Minimizes the number of entities such as classes, methods, functions, and the like.  
• Giảm thiểu số lượng thực thể như lớp, phương thức, hàm, v.v.*

*Of these, I focus mostly on duplication. When the same thing is done over and over, it’s a sign that there is an idea in our mind that is not well represented in the code. I try to ﬁgure out what it is. Then I try to express that idea more clearly.  
Trong số này, tôi tập trung chủ yếu vào sự trùng lặp. Khi cùng một việc được thực hiện lặp đi lặp lại, đó là dấu hiệu cho thấy có một ý tưởng trong đầu chúng ta không được trình bày rõ ràng trong mã. Tôi cố gắng tìm ra nó là gì. Sau đó, tôi cố gắng diễn đạt ý tưởng đó rõ ràng hơn.*

*Expressiveness to me includes meaningful names, and I am likely to change the names of things several times before I settle in. With modern coding tools such as Eclipse, renaming is quite inexpensive, so it doesn’t trouble me to change. Expressiveness goes  
Tính biểu cảm đối với tôi bao gồm những cái tên có ý nghĩa và tôi có thể thay đổi tên của mọi thứ vài lần trước khi ổn định. Với các công cụ mã hóa hiện đại như Eclipse, việc đổi tên khá rẻ nên tôi không gặp khó khăn gì khi thay đổi. biểu cảm đi*

<https://www.facebook.com/groups/phattrienlaptrinh>

**The Total Cost of Owning a Mess** 11  
Tổng chi phí sở hữu một mớ hỗn độn 11

*beyond names, however. I also look at whether an object or method is doing more than one thing. If it’s an object, it probably needs to be broken into two or more objects. If it’s a method, I will always use the Extract Method refactoring on it, resulting in one method that says more clearly what it does, and some submethods saying how it is done.  
ngoài tên, tuy nhiên. Tôi cũng xem liệu một đối tượng hoặc phương thức có đang làm nhiều việc hay không. Nếu đó là một đối tượng, thì có lẽ nó cần được chia thành hai hoặc nhiều đối tượng. Nếu đó là một phương thức, tôi sẽ luôn sử dụng tái cấu trúc Phương thức trích xuất trên nó, dẫn đến một phương thức nói rõ hơn những gì nó làm và một số phương thức con cho biết cách thực hiện.*

*Duplication and expressiveness take me a very long way into what I consider clean code, and improving dirty code with just these two things in mind can make a huge differ-ence. There is, however, one other thing that I’m aware of doing, which is a bit harder to explain.  
Tính trùng lặp và tính biểu cảm đưa tôi đi một chặng đường rất dài tới cái mà tôi coi là mã sạch, và việc cải thiện mã bẩn chỉ với hai điều này trong tâm trí có thể tạo ra sự khác biệt lớn. Tuy nhiên, có một điều khác mà tôi biết đang làm, điều này khó giải thích hơn một chút.*

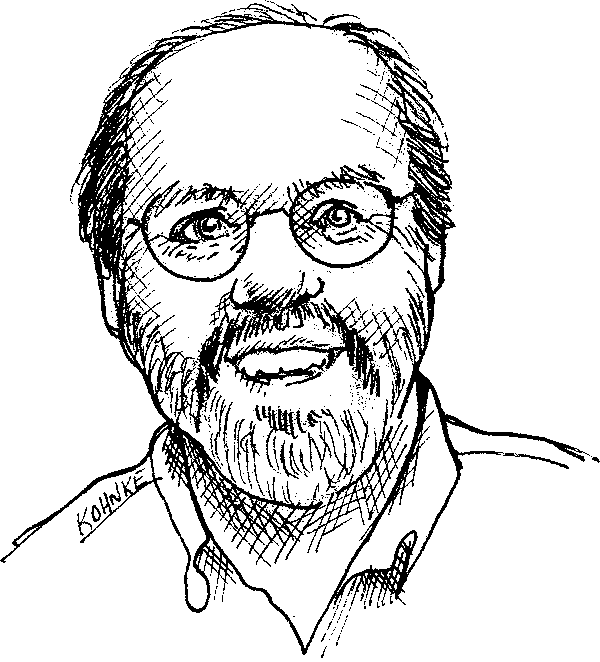
*After years of doing this work, it seems to me that all programs are made up of very similar elements. One example is “ﬁnd things in a collection.” Whether we have a data-base of employee records, or a hash map of keys and values, or an array of items of some kind, we often ﬁnd ourselves wanting a particular item from that collection. When I ﬁnd that happening, I will often wrap the particular implementation in a more abstract method or class. That gives me a couple of interesting advantages.  
Sau nhiều năm làm công việc này, đối với tôi, dường như tất cả các chương trình đều được tạo thành từ những yếu tố rất giống nhau. Một ví dụ là “tìm đồ vật trong một bộ sưu tập.” Cho dù chúng ta có một cơ sở dữ liệu gồm các bản ghi của nhân viên, hoặc một bản đồ băm của các khóa và giá trị, hoặc một mảng các mục thuộc loại nào đó, chúng ta thường thấy mình muốn có một mục cụ thể từ bộ sưu tập đó. Khi tôi thấy điều đó xảy ra, tôi sẽ thường bọc việc triển khai cụ thể trong một phương thức hoặc lớp trừu tượng hơn. Điều đó mang lại cho tôi một vài lợi thế thú vị.*

*I can implement the functionality now with something simple, say a hash map, but since now all the references to that search are covered by my little abstraction, I can change the implementation any time I want. I can go forward quickly while preserving my ability to change later.  
Bây giờ tôi có thể triển khai chức năng này bằng một thứ gì đó đơn giản, chẳng hạn như bản đồ băm, nhưng vì bây giờ tất cả các tham chiếu đến tìm kiếm đó được bao phủ bởi phần trừu tượng nhỏ của tôi, tôi có thể thay đổi cách triển khai bất kỳ lúc nào tôi muốn. Tôi có thể tiến lên phía trước một cách nhanh chóng trong khi vẫn giữ được khả năng thay đổi sau này.*

*In addition, the collection abstraction often calls my attention to what’s “really” going on, and keeps me from running down the path of implementing arbitrary collection behavior when all I really need is a few fairly simple ways of ﬁnding what I want.  
Ngoài ra, sự trừu tượng hóa bộ sưu tập thường thu hút sự chú ý của tôi đến những gì “thực sự” đang diễn ra và ngăn tôi chạy theo con đường thực hiện hành vi bộ sưu tập tùy ý khi tất cả những gì tôi thực sự cần là một vài cách khá đơn giản để tìm thấy những gì tôi muốn.*

*Reduced duplication, high expressiveness, and early building of simple abstractions. That’s what makes clean code for me.  
Giảm sự trùng lặp, tính biểu cảm cao và sớm xây dựng các khái niệm trừu tượng đơn giản. Đó là những gì làm cho mã sạch đối với tôi.*

Here, in a few short paragraphs, Ron has summarized the contents of this book. No duplication, one thing, expressiveness, tiny abstractions. Everything is there.  
Ở đây, trong một vài đoạn ngắn, Ron đã tóm tắt nội dung của cuốn sách này. Không trùng lặp, một điều, biểu cảm, trừu tượng nhỏ. Mọi thứ đều ở đó.

**Ward Cunningham, inventor of Wiki, inventor of Fit, coinventor of eXtreme Programming. Motive force behind Design Patterns. Smalltalk and OO thought leader. The godfather of all those who care about code.**  
Ward Cunningham, người phát minh ra Wiki, người phát minh ra Fit, người phát minh ra Lập trình cực đoan. Động lực đằng sau các Mẫu thiết kế. Smalltalk và nhà lãnh đạo tư tưởng OO. Cha đỡ đầu của tất cả những ai quan tâm đến mã.

*You know you are working on clean code when each routine you read turns out to be pretty much what you expected. You can call it beautiful code when the code also makes it look like the language was made for the problem.  
Bạn biết rằng bạn đang làm việc với mã sạch khi mỗi quy trình bạn đọc hóa ra lại giống như những gì bạn mong đợi. Bạn có thể gọi nó là mã đẹp khi mã cũng làm cho nó trông giống như ngôn ngữ được tạo ra cho vấn đề.*

Statements like this are characteristic of Ward. You read it, nod your head, and then go on to the  
Những tuyên bố như thế này là đặc trưng của Ward. Bạn đọc nó, gật đầu, và sau đó tiếp tục

next topic. It sounds so reasonable, so obvious, that it barely registers as something profound. You might think it was pretty much what you expected. But let’s take a closer look.  
chủ đề tiếp theo. Nghe có vẻ hợp lý, quá rõ ràng đến nỗi nó hầu như không được coi là một điều gì đó sâu sắc. Bạn có thể nghĩ rằng đó là khá nhiều những gì bạn mong đợi. Nhưng chúng ta hãy xem xét kỹ hơn.

<https://www.facebook.com/groups/phattrienlaptrinh>

12 **Chapter 1: Clean Code**  
12 Chương 1: Code sạch

“. . . pretty much what you expected.” When was the last time you saw a module that was pretty much what you expected? Isn’t it more likely that the modules you look at will be puzzling, complicated, tangled? Isn’t misdirection the rule? Aren’t you used to ﬂailing about trying to grab and hold the threads of reasoning that spew forth from the whole sys-tem and weave their way through the module you are reading? When was the last time you read through some code and nodded your head the way you might have nodded your head at Ward’s statement?  
“. . . khá nhiều những gì bạn mong đợi. Lần cuối cùng bạn nhìn thấy một mô-đun giống như những gì bạn mong đợi là khi nào? Có nhiều khả năng là các mô-đun mà bạn xem sẽ khó hiểu, phức tạp, rối rắm? Định hướng sai không phải là quy luật sao? Chẳng phải bạn đã từng loay hoay với việc cố gắng nắm bắt và nắm giữ các luồng lý luận nảy sinh từ toàn bộ hệ thống và len lỏi qua mô-đun mà bạn đang đọc hay sao? Lần cuối cùng bạn đọc qua một đoạn mã nào đó và gật đầu như cách bạn có thể gật đầu trước tuyên bố của Ward là khi nào?

Ward expects that when you read clean code you won’t be surprised at all. Indeed, you won’t even expend much effort. You will read it, and it will be pretty much what you expected. It will be obvious, simple, and compelling. Each module will set the stage for the next. Each tells you how the next will be written. Programs that are *that* clean are so profoundly well written that you don’t even notice it. The designer makes it look ridicu-lously simple like all exceptional designs.  
Ward hy vọng rằng khi bạn đọc mã sạch, bạn sẽ không ngạc nhiên chút nào. Thật vậy, bạn thậm chí sẽ không tốn nhiều công sức. Bạn sẽ đọc nó, và nó sẽ giống như những gì bạn mong đợi. Nó sẽ rõ ràng, đơn giản và hấp dẫn. Mỗi mô-đun sẽ tạo tiền đề cho phần tiếp theo. Mỗi người cho bạn biết phần tiếp theo sẽ được viết như thế nào. Các chương trình rõ ràng được viết sâu sắc đến mức bạn thậm chí không nhận thấy điều đó. Nhà thiết kế làm cho nó trông đơn giản đến kỳ cục giống như tất cả các thiết kế đặc biệt.

And what about Ward’s notion of beauty? We’ve all railed against the fact that our lan-guages weren’t designed for our problems. But Ward’s statement puts the onus back on us. He says that beautiful code *makes the language look like it was made for the problem*! So it’s *our* responsibility to make the language look simple! Language bigots everywhere, beware! It is not the language that makes programs appear simple. It is the programmer that make the language appear simple!  
Còn quan niệm về cái đẹp của Ward thì sao? Tất cả chúng ta đều phản đối thực tế là ngôn ngữ của chúng ta không được thiết kế cho các vấn đề của chúng ta. Nhưng tuyên bố của Ward đặt lại trách nhiệm cho chúng tôi. Anh ấy nói rằng mã đẹp làm cho ngôn ngữ trông giống như nó được tạo ra để giải quyết vấn đề! Vì vậy, trách nhiệm của chúng tôi là làm cho ngôn ngữ trông thật đơn giản! Ngôn ngữ cố chấp ở khắp mọi nơi, hãy cẩn thận! Nó không phải là ngôn ngữ làm cho các chương trình có vẻ đơn giản. Lập trình viên làm cho ngôn ngữ có vẻ đơn giản!

[**Schools of Thought**](#_page_37_0)  
Trường học trong tưởng tượng

What about me (Uncle Bob)? What do I think clean code is? This book will tell you, in hideous detail, what I and my compatriots think about clean code. We will tell you what we think makes a clean variable name, a clean function, a clean class, etc. We will present these opinions as abso-lutes, and we will not apologize for our stridence. To us, at this point in our careers, they *are* abso-lutes. They are *our school of thought* about clean code.  
Còn tôi (chú Bob) thì sao? Tôi nghĩ mã sạch là gì? Cuốn sách này sẽ cho bạn biết, một cách chi tiết ghê gớm, những gì tôi và những người đồng hương của tôi nghĩ về mã sạch. Chúng tôi sẽ cho bạn biết những gì chúng tôi nghĩ tạo nên một tên biến sạch, một hàm sạch, một lớp sạch, v.v. Đối với chúng tôi, tại thời điểm này trong sự nghiệp của mình, họ là những người tuyệt đối. Họ là trường phái tư tưởng của chúng tôi về mã sạch.

Martial artists do not all agree about the best martial art, or the best technique within a martial art. Often master martial artists will form their own schools of thought and gather students to learn from them. So we see *Gracie Jiu Jistu*,  
Các võ sĩ không phải tất cả đều đồng ý về võ thuật tốt nhất, hoặc kỹ thuật tốt nhất trong một môn võ thuật. Thông thường, các bậc thầy võ thuật sẽ hình thành các trường phái tư tưởng của riêng họ và thu thập các học sinh để học hỏi từ họ. Vì vậy, chúng ta thấy Gracie Jiu Jistu,

founded and taught by the Gracie family in Brazil. We see *Hakkoryu Jiu Jistu*, founded and taught by Okuyama Ryuho in Tokyo. We see *Jeet Kune Do*, founded and taught by Bruce Lee in the United States.  
được thành lập và giảng dạy bởi gia đình Gracie ở Brazil. Chúng tôi thấy Hakkoryu Jiu Jistu, được thành lập và giảng dạy bởi Okuyama Ryuho ở Tokyo. Chúng ta thấy Triệt Quyền Đạo do Lý Tiểu Long sáng lập và giảng dạy tại Hoa Kỳ.

<https://www.facebook.com/groups/phattrienlaptrinh>

**WeAre Authors** 13  
Chúng Tôi Là Tác Giả 13

Students of these approaches immerse themselves in the teachings of the founder. They dedicate themselves to learn what that particular master teaches, often to the exclu-sion of any other master’s teaching. Later, as the students grow in their art, they may become the student of a different master so they can broaden their knowledge and practice. Some eventually go on to reﬁne their skills, discovering new techniques and founding their own schools.  
Học sinh của những phương pháp này đắm mình trong những lời dạy của người sáng lập. Họ cống hiến hết mình để học những gì mà bậc thầy cụ thể đó dạy, thường là loại trừ sự giảng dạy của bất kỳ bậc thầy nào khác. Sau này, khi các học viên trưởng thành hơn trong nghệ thuật, họ có thể trở thành học trò của một bậc thầy khác để họ có thể mở rộng kiến thức và thực hành. Một số cuối cùng tiếp tục hoàn thiện các kỹ năng của họ, khám phá các kỹ thuật mới và thành lập trường học của riêng họ.

None of these different schools is absolutely *right*. Yet within a particular school we *act* as though the teachings and techniques *are* right. After all, there is a right way to prac-tice Hakkoryu Jiu Jitsu, or Jeet Kune Do. But this rightness within a school does not inval-idate the teachings of a different school.  
Không có trường phái nào trong số này là hoàn toàn đúng. Tuy nhiên, trong một trường học cụ thể, chúng ta hành động như thể những lời dạy và kỹ thuật là đúng. Rốt cuộc, có một cách đúng đắn để luyện tập Hakkoryu Jiu Jitsu, hay Jeet Kune Do. Nhưng sự đúng đắn này trong một trường phái không làm mất giá trị những lời dạy của một trường phái khác.

Consider this book a description of the *Object Mentor School of Clean Code*. The techniques and teachings within are the way that *we* practice *our* art. We are willing to claim that if you follow these teachings, you will enjoy the beneﬁts that we have enjoyed, and you will learn to write code that is clean and professional. But don’t make the mistake of thinking that we are somehow “right” in any absolute sense. There are other schools and other masters that have just as much claim to professionalism as we. It would behoove you to learn from them as well.  
Hãy coi cuốn sách này là một mô tả về Object Mentor School of Clean Code. Các kỹ thuật và giáo lý bên trong là cách chúng ta thực hành nghệ thuật của mình. Chúng tôi sẵn sàng tuyên bố rằng nếu bạn làm theo những hướng dẫn này, bạn sẽ được hưởng những lợi ích mà chúng tôi đã được hưởng, và bạn sẽ học cách viết mã rõ ràng và chuyên nghiệp. Nhưng đừng mắc sai lầm khi nghĩ rằng bằng cách nào đó chúng ta “đúng” theo bất kỳ nghĩa tuyệt đối nào. Có những trường khác và những bậc thầy khác cũng có nhiều yêu cầu về tính chuyên nghiệp như chúng tôi. Bạn cũng nên học hỏi từ họ.

Indeed, many of the recommendations in this book are controversial. You will proba-bly not agree with all of them. You might violently disagree with some of them. That’s ﬁne. We can’t claim ﬁnal authority. On the other hand, the recommendations in this book are things that we have thought long and hard about. We have learned them through decades of experience and repeated trial and error. So whether you agree or disagree, it would be a shame if you did not see, and respect, our point of view.  
Thật vậy, nhiều khuyến nghị trong cuốn sách này đang gây tranh cãi. Bạn có thể sẽ không đồng ý với tất cả chúng. Bạn có thể không đồng ý với một số người trong số họ. Không sao đâu. Chúng tôi không thể yêu cầu thẩm quyền cuối cùng. Mặt khác, những khuyến nghị trong cuốn sách này là những điều mà chúng tôi đã suy nghĩ rất lâu và kỹ lưỡng. Chúng tôi đã học được chúng qua nhiều thập kỷ kinh nghiệm và thử nghiệm và sai sót lặp đi lặp lại. Vì vậy, cho dù bạn đồng ý hay không đồng ý, sẽ thật đáng tiếc nếu bạn không nhìn thấy và tôn trọng quan điểm của chúng tôi.

[**WeAre Authors**](#_page_37_0)  
Chúng tôi là tác giả

The @authorﬁeld of a Javadoc tells us who we are. We are authors. And one thing about authors is that they have readers. Indeed, authors are *responsible* for communicating well with their readers. The next time you write a line of code, remember you are an author, writing for readers who will judge your effort.  
@authorfield của Javadoc cho chúng tôi biết chúng tôi là ai. Chúng tôi là tác giả. Và một điều về các tác giả là họ có độc giả. Thật vậy, các tác giả chịu trách nhiệm giao tiếp tốt với độc giả của họ. Lần tới khi bạn viết một dòng mã, hãy nhớ rằng bạn là một tác giả, viết cho độc giả, những người sẽ đánh giá nỗ lực của bạn.

You might ask: How much is code really read? Doesn’t most of the effort go into writing it?  
Bạn có thể hỏi: Mã thực sự được đọc bao nhiêu? Không phải hầu hết nỗ lực đều dành cho việc viết nó sao?

Have you ever played back an edit session? In the 80s and 90s we had editors like Emacs that kept track of every keystroke. You could work for an hour and then play back your whole edit session like a high-speed movie. When I did this, the results were fascinating.  
Bạn đã bao giờ phát lại một phiên chỉnh sửa chưa? Trong những năm 80 và 90, chúng tôi có các trình soạn thảo như Emacs theo dõi mọi lần nhấn phím. Bạn có thể làm việc trong một giờ và sau đó phát lại toàn bộ phiên chỉnh sửa của mình như một bộ phim tốc độ cao. Khi tôi làm điều này, kết quả thật hấp dẫn.

The vast majority of the playback was scrolling and navigating to other modules!  
Phần lớn quá trình phát lại là cuộn và điều hướng đến các mô-đun khác!

*Bob enters the module.  
Bob vào mô-đun.*

*He scrolls down to the function needing change. He pauses, considering his options.  
Anh ấy cuộn xuống chức năng cần thay đổi. Anh dừng lại, xem xét các lựa chọn của mình.*

*Oh, he’s scrolling up to the top of the module to check the initialization of a variable. Now he scrolls back down and begins to type.  
Ồ, anh ấy đang cuộn lên đầu mô-đun để kiểm tra việc khởi tạo một biến. Bây giờ anh ấy cuộn xuống và bắt đầu gõ.*

<https://www.facebook.com/groups/phattrienlaptrinh>

14 **Chapter 1: Clean Code**  
14 Chương 1: Code sạch

*Ooops, he’s erasing what he typed! He types it again.  
Rất tiếc, anh ấy đang xóa những gì anh ấy đã nhập! Anh gõ lại.*

*He erases it again!  
Anh lại xóa đi!*

*He types half of something else but then erases that!  
Anh ấy gõ một nửa thứ khác nhưng sau đó lại xóa thứ đó!*

*He scrolls down to another function that calls the function he’s changing to see how it is called.  
Anh ấy cuộn xuống một chức năng khác gọi chức năng mà anh ấy đang thay đổi để xem nó được gọi như thế nào.*

*He scrolls back up and types the same code he just erased. He pauses.  
Anh ấy cuộn lại và nhập cùng mã mà anh ấy vừa xóa. Anh dừng lại.*

*He erases that code again!  
Anh ta xóa mã đó một lần nữa!*

*He pops up another window and looks at a subclass. Is that function overridden?  
Anh ta bật lên một cửa sổ khác và nhìn vào một phân lớp. Chức năng đó có bị ghi đè không?*

. . .  
. . .

You get the drift. Indeed, the ratio of time spent reading vs. writing is well over 10:1. We are *constantly* reading old code as part of the effort to write new code.  
Bạn nhận được sự trôi dạt. Thật vậy, tỷ lệ thời gian dành cho đọc so với viết là hơn 10:1. Chúng tôi liên tục đọc mã cũ như một phần của nỗ lực viết mã mới.

Because this ratio is so high, we want the reading of code to be easy, even if it makes the writing harder. Of course there’s no way to write code without reading it, so *making it easy to read actually makes it easier to write*.  
Vì tỷ lệ này quá cao nên chúng tôi muốn việc đọc mã trở nên dễ dàng, ngay cả khi điều đó khiến việc viết khó hơn. Tất nhiên, không có cách nào để viết mã mà không đọc nó, vì vậy làm cho nó dễ đọc thực sự giúp viết dễ dàng hơn.

There is no escape from this logic. You cannot write code if you cannot read the sur-rounding code. The code you are trying to write today will be hard or easy to write depending on how hard or easy the surrounding code is to read. So if you want to go fast, if you want to get done quickly, if you want your code to be easy to write, make it easy to read.  
Không có lối thoát khỏi logic này. Bạn không thể viết mã nếu bạn không thể đọc mã bao quanh. Mã bạn đang cố gắng viết ngày hôm nay sẽ khó viết hay dễ viết tùy thuộc vào mức độ khó đọc hay dễ dàng của mã xung quanh. Vì vậy, nếu bạn muốn đi nhanh, nếu bạn muốn hoàn thành nhanh chóng, nếu bạn muốn mã của mình dễ viết, hãy làm cho nó dễ đọc.

[**The Boy Scout Rule**](#_page_37_0)  
Quy tắc Hướng đạo sinh

It’s not enough to write the code well. The code has to be *kept clean* over time. We’ve all seen code rot and degrade as time passes. So we must take an active role in preventing this degradation.  
Nó không đủ để viết mã tốt. Mã phải được giữ sạch theo thời gian. Tất cả chúng ta đều đã thấy mã bị thối rữa và xuống cấp theo thời gian. Vì vậy chúng ta phải đóng vai trò tích cực trong việc ngăn chặn sự xuống cấp này.

The Boy Scouts of America have a simple rule that we can apply to our profession.  
Hội Nam Hướng đạo Hoa Kỳ có một quy tắc đơn giản mà chúng tôi có thể áp dụng cho nghề nghiệp của mình.

*Leave the campground cleaner than you found it.5  
Để khu cắm trại sạch sẽ hơn bạn thấy.5*

If we all checked-in our code a little cleaner than when we checked it out, the code simply could not rot. The cleanup doesn’t have to be something big. Change one variable name for the better, break up one function that’s a little too large, eliminate one small bit of duplication, clean up one composite if statement.  
Nếu tất cả chúng ta đã đăng ký mã của mình gọn gàng hơn một chút so với khi chúng ta kiểm tra, thì đơn giản là mã không thể bị hỏng. Việc dọn dẹp không phải là một cái gì đó lớn. Thay đổi một tên biến cho tốt hơn, chia nhỏ một hàm hơi quá lớn, loại bỏ một chút trùng lặp nhỏ, làm sạch một câu lệnh if tổng hợp.

Can you imagine working on a project where the code *simply got better* as time passed? Do you believe that any other option is professional? Indeed, isn’t continuous improvement an intrinsic part of professionalism?  
Bạn có thể tưởng tượng làm việc trên một dự án mà mã đơn giản trở nên tốt hơn theo thời gian không? Bạn có tin rằng bất kỳ lựa chọn nào khác là chuyên nghiệp? Thật vậy, không phải cải tiến liên tục là một phần nội tại của tính chuyên nghiệp sao?

5. This was adapted from Robert Stephenson Smyth Baden-Powell’s farewell message to the Scouts: “Try and leave this world a little better than you found it . . .”  
5. Điều này được phỏng theo lời nhắn chia tay của Robert Stephenson Smyth Baden-Powell gửi cho các Hướng đạo sinh: “Hãy cố gắng rời bỏ thế giới này tốt hơn một chút so với những gì bạn đã tìm thấy. . .”

<https://www.facebook.com/groups/phattrienlaptrinh>

**Bibliography** 15  
Thư mục 15

[**Prequel and Principles**](#_page_37_0)  
Tiền truyện và nguyên tắc

In many ways this book is a “prequel” to a book I wrote in 2002 entitled *Agile Software Development: Principles, Patterns, and Practices* (PPP). The PPP book concerns itself with the principles of object-oriented design, and many of the practices used by profes-sional developers. If you have not read PPP, then you may ﬁnd that it continues the story told by this book. If you have already read it, then you’ll ﬁnd many of the sentiments of that book echoed in this one at the level of code.  
Theo nhiều cách, cuốn sách này là “phần tiền truyện” của cuốn sách tôi viết năm 2002 có tựa đề Phát triển phần mềm linh hoạt: Nguyên tắc, khuôn mẫu và thực tiễn (PPP). Cuốn sách PPP liên quan đến các nguyên tắc của thiết kế hướng đối tượng và nhiều phương pháp được các nhà phát triển chuyên nghiệp sử dụng. Nếu bạn chưa đọc PPP, thì bạn có thể thấy rằng nó tiếp nối câu chuyện được kể trong cuốn sách này. Nếu bạn đã đọc nó, thì bạn sẽ thấy nhiều cảm xúc của cuốn sách đó được lặp lại trong cuốn sách này ở cấp độ mã.

In this book you will ﬁnd sporadic references to various principles of design. These include the Single Responsibility Principle (SRP), the Open Closed Principle (OCP), and the Dependency Inversion Principle (DIP) among others. These principles are described in depth in PPP.  
Trong cuốn sách này, bạn sẽ tìm thấy các tài liệu tham khảo lẻ tẻ về các nguyên tắc thiết kế khác nhau. Chúng bao gồm Nguyên tắc trách nhiệm duy nhất (SRP), Nguyên tắc đóng mở (OCP) và Nguyên tắc đảo ngược phụ thuộc (DIP) trong số những nguyên tắc khác. Những nguyên tắc này được mô tả chi tiết trong PPP.

[**Conclusion**](#_page_37_0)  
Phần kết luận

Books on art don’t promise to make you an artist. All they can do is give you some of the tools, techniques, and thought processes that other artists have used. So too this book can-not promise to make you a good programmer. It cannot promise to give you “code-sense.” All it can do is show you the thought processes of good programmers and the tricks, tech-niques, and tools that they use.  
Sách về nghệ thuật không hứa hẹn biến bạn thành một nghệ sĩ. Tất cả những gì họ có thể làm là cung cấp cho bạn một số công cụ, kỹ thuật và quy trình tư duy mà các nghệ sĩ khác đã sử dụng. Vì vậy, cuốn sách này cũng không thể hứa hẹn giúp bạn trở thành một lập trình viên giỏi. Nó không thể hứa sẽ cung cấp cho bạn “cảm giác về mã”. Tất cả những gì nó có thể làm là cho bạn thấy quá trình suy nghĩ của các lập trình viên giỏi và các thủ thuật, kỹ thuật và công cụ mà họ sử dụng.

Just like a book on art, this book will be full of details. There will be lots of code. You’ll see good code and you’ll see bad code. You’ll see bad code transformed into good code. You’ll see lists of heuristics, disciplines, and techniques. You’ll see example after example. After that, it’s up to you.  
Giống như một cuốn sách về nghệ thuật, cuốn sách này sẽ có đầy đủ các chi tiết. Sẽ có rất nhiều mã. Bạn sẽ thấy mã tốt và bạn sẽ thấy mã xấu. Bạn sẽ thấy mã xấu được chuyển thành mã tốt. Bạn sẽ thấy danh sách các kinh nghiệm, nguyên tắc và kỹ thuật. Bạn sẽ thấy hết ví dụ này đến ví dụ khác. Sau đó, nó tùy thuộc vào bạn.

Remember the old joke about the concert violinist who got lost on his way to a perfor-mance? He stopped an old man on the corner and asked him how to get to Carnegie Hall. The old man looked at the violinist and the violin tucked under his arm, and said: “Prac-tice, son. Practice!”  
Bạn có nhớ câu chuyện cười cũ về nghệ sĩ vĩ cầm hòa nhạc bị lạc trên đường đến buổi biểu diễn không? Anh ta chặn một ông già ở góc phố và hỏi ông ta đường đến Carnegie Hall. Ông già nhìn người nghệ sĩ vĩ cầm và cây vĩ cầm kẹp dưới cánh tay anh ta, và nói: “Tập đi, con trai. Luyện tập!"

[**Bibliography**](#_page_37_0)  
Thư mục

**[Beck07]:** *Implementation Patterns*, Kent Beck, Addison-Wesley, 2007.  
[Beck07]: Các mẫu triển khai, Kent Beck, Addison-Wesley, 2007.

**[Knuth92]:** *Literate Programming*, Donald E. Knuth, Center for the Study of Language and Information, Leland Stanford Junior University, 1992.  
[Knuth92]: Literate Programming, Donald E. Knuth, Trung tâm Nghiên cứu Ngôn ngữ và Thông tin, Leland Stanford Junior University, 1992.

<https://www.facebook.com/groups/phattrienlaptrinh>

*This page intentionally left blank*

<https://www.facebook.com/groups/phattrienlaptrinh>

[**2**](#_page_37_0)

[**Meaningful Names**](#_page_37_0)  
tên ý nghĩa

by Tim Ottinger  
của Tim Ottinger

[**Introduction**](#_page_37_0)  
Giới thiệu

Names are everywhere in software. We name our variables, our functions, our arguments, classes, and packages. We name our source ﬁles and the directories that contain them. We name our jar ﬁles and war ﬁles and ear ﬁles. We name and name and name. Because we do  
Tên ở khắp mọi nơi trong phần mềm. Chúng tôi đặt tên cho các biến, hàm, đối số, lớp và gói của chúng tôi. Chúng tôi đặt tên cho các tệp nguồn của chúng tôi và các thư mục chứa chúng. Chúng tôi đặt tên cho các tệp jar, tệp chiến tranh và tệp tai. Chúng tôi đặt tên và đặt tên và đặt tên. Bởi vì chúng tôi làm

17

<https://www.facebook.com/groups/phattrienlaptrinh>

18 **Chapter 2: Meaningful Names**  
18 Chương 2: Ý Nghĩa Tên

so much of it, we’d better do it well. What follows are some simple rules for creating good names.  
rất nhiều trong số đó, chúng ta nên làm tốt hơn. Sau đây là một số quy tắc đơn giản để tạo ra những cái tên hay.

[**Use Intention-Revealing Names**](#_page_37_0)  
Sử dụng tên tiết lộ ý định

It is easy to say that names should reveal intent. What we want to impress upon you is that we are *serious* about this. Choosing good names takes time but saves more than it takes. So take care with your names and change them when you ﬁnd better ones. Everyone who reads your code (including you) will be happier if you do.  
Thật dễ dàng để nói rằng tên nên tiết lộ ý định. Điều chúng tôi muốn gây ấn tượng với bạn là chúng tôi rất nghiêm túc về việc này. Chọn tên hay cần thời gian nhưng tiết kiệm được nhiều hơn là mất. Vì vậy, hãy cẩn thận với tên của bạn và thay đổi chúng khi bạn tìm thấy những cái tốt hơn. Mọi người đọc mã của bạn (bao gồm cả bạn) sẽ vui hơn nếu bạn đọc.

The name of a variable, function, or class, should answer all the big questions. It should tell you why it exists, what it does, and how it is used. If a name requires a com-ment, then the name does not reveal its intent.  
Tên của một biến, hàm hoặc lớp phải trả lời tất cả các câu hỏi lớn. Nó sẽ cho bạn biết tại sao nó tồn tại, nó làm gì và nó được sử dụng như thế nào. Nếu một tên yêu cầu bình luận, thì tên đó không tiết lộ ý định của nó.

int d; // elapsed time in days

The name dreveals nothing. It does not evoke a sense of elapsed time, nor of days. We should choose a name that speciﬁes what is being measured and the unit of that measure-ment:  
Cái tên không tiết lộ điều gì. Nó không gợi lên cảm giác về thời gian trôi qua, cũng như ngày tháng. Chúng ta nên chọn một cái tên xác định những gì đang được đo lường và đơn vị đo lường đó:

int elapsedTimeInDays; int daysSinceCreation; int daysSinceModification; int fileAgeInDays;

Choosing names that reveal intent can make it much easier to understand and change code. What is the purpose of this code?  
Chọn tên tiết lộ ý định có thể giúp dễ hiểu và thay đổi mã hơn nhiều. Mục đích của mã này là gì?

public List<int[]> getThem() {

List<int[]> list1 = new ArrayList<int[]>(); for (int[] x : theList)

if (x[0] == 4) list1.add(x);

return list1; }

Why is it hard to tell what this code is doing? There are no complex expressions. Spacing and indentation are reasonable. There are only three variables and two constants mentioned. There aren’t even any fancy classes or polymorphic methods, just a list of arrays (or so it seems).  
Tại sao khó nói mã này đang làm gì? Không có biểu thức phức tạp. Khoảng cách và thụt đầu dòng hợp lý. Chỉ có ba biến và hai hằng số được đề cập. Thậm chí không có bất kỳ lớp ưa thích hoặc phương thức đa hình nào, chỉ là một danh sách các mảng (hoặc có vẻ như vậy).

The problem isn’t the simplicity of the code but the *implicity* of the code (to coin a phrase): the degree to which the context is not explicit in the code itself. The code implic-itly requires that we know the answers to questions such as:  
Vấn đề không phải là sự đơn giản của mã mà là tính ẩn của mã (để tạo ra một cụm từ): mức độ mà ngữ cảnh không rõ ràng trong chính mã đó. Mã ngầm yêu cầu chúng tôi biết câu trả lời cho các câu hỏi như:

**1.** What kinds of things are in theList?  
1. Có những loại gì trong Danh sách?

**2.** What is the signiﬁcance of the zeroth subscript of an item in theList? **3.** What is the signiﬁcance of the value 4?  
2. Ý nghĩa của chỉ số con thứ 0 của một mục trong Danh sách là gì? 3. Ý nghĩa của giá trị 4 là gì?

**4.** How would I use the list being returned?  
4. Tôi sẽ sử dụng danh sách được trả về như thế nào?

<https://www.facebook.com/groups/phattrienlaptrinh>

**Avoid Disinformation** 19  
Tránh thông tin sai lệch 19

The answers to these questions are not present in the code sample, *but they could have been*. Say that we’re working in a mine sweeper game. We ﬁnd that the board is a list of cells called theList. Let’s rename that to gameBoard.  
Câu trả lời cho những câu hỏi này không có trong mẫu mã, nhưng chúng có thể có. Giả sử chúng ta đang làm việc trong trò chơi quét mìn. Chúng tôi thấy rằng bảng là một danh sách các ô được gọi là Danh sách. Hãy đổi tên nó thành gameBoard.

Each cell on the board is represented by a simple array. We further ﬁnd that the zeroth subscript is the location of a status value and that a status value of 4 means “ﬂagged.” Just by giving these concepts names we can improve the code considerably:  
Mỗi ô trên bàn cờ được biểu diễn bằng một mảng đơn giản. Chúng tôi còn phát hiện ra rằng chỉ số thứ 0 là vị trí của một giá trị trạng thái và giá trị trạng thái là 4 có nghĩa là “được gắn cờ”. Chỉ bằng cách đặt tên cho các khái niệm này, chúng ta có thể cải thiện mã một cách đáng kể:

public List<int[]> getFlaggedCells() {

List<int[]> flaggedCells = new ArrayList<int[]>(); for (int[] cell : gameBoard)

if (cell[STATUS\_VALUE] == FLAGGED) flaggedCells.add(cell);

return flaggedCells; }

Notice that the simplicity of the code has not changed. It still has exactly the same number of operators and constants, with exactly the same number of nesting levels. But the code has become much more explicit.  
Lưu ý rằng tính đơn giản của mã không thay đổi. Nó vẫn có cùng một số lượng toán tử và hằng số, với cùng một số cấp độ lồng nhau. Nhưng mã đã trở nên rõ ràng hơn nhiều.

We can go further and write a simple class for cells instead of using an array of ints. It can include an intention-revealing function (call it isFlagged) to hide the magic num-bers. It results in a new version of the function:  
Chúng ta có thể tiến xa hơn và viết một lớp đơn giản cho các ô thay vì sử dụng một mảng các số nguyên. Nó có thể bao gồm chức năng tiết lộ ý định (gọi là isFlagged) để ẩn các con số ma thuật. Nó dẫn đến một phiên bản mới của chức năng:

public List<Cell> getFlaggedCells() {

List<Cell> flaggedCells = new ArrayList<Cell>(); for (Cell cell : gameBoard)

if (cell.isFlagged()) flaggedCells.add(cell);

return flaggedCells; }

With these simple name changes, it’s not difﬁcult to understand what’s going on. This is the power of choosing good names.  
Với những thay đổi tên đơn giản này, không khó để hiểu chuyện gì đang xảy ra. Đây là sức mạnh của việc chọn tên tốt.

[**Avoid Disinformation**](#_page_37_0)  
Tránh thông tin sai lệch

Programmers must avoid leaving false clues that obscure the meaning of code. We should avoid words whose entrenched meanings vary from our intended meaning. For example, hp, aix, and scowould be poor variable names because they are the names of Unix plat-forms or variants. Even if you are coding a hypotenuse and hplooks like a good abbrevia-tion, it could be disinformative.  
Các lập trình viên phải tránh để lại các manh mối sai làm che khuất ý nghĩa của mã. Chúng ta nên tránh những từ có ý nghĩa cố thủ khác với ý nghĩa dự định của chúng ta. Ví dụ: hp, aix và sco sẽ là những tên biến kém vì chúng là tên của các biến thể hoặc dạng nền tảng Unix. Ngay cả khi bạn đang mã hóa một cạnh huyền và hp trông giống như một từ viết tắt tốt, thì nó có thể là sai thông tin.

Do not refer to a grouping of accounts as an accountListunless it’s actually a List. The word list means something speciﬁc to programmers. If the container holding the accounts is not actually a List, it may lead to false conclusions.1 So accountGroup or bunchOfAccounts or just plain accounts would be better.  
Không coi một nhóm tài khoản là một danh sách tài khoản trừ khi đó thực sự là một Danh sách. Danh sách từ có nghĩa là một cái gì đó cụ thể cho các lập trình viên. Nếu vùng chứa chứa các tài khoản không thực sự là một Danh sách, nó có thể dẫn đến kết luận sai.1 Vì vậy, accountGroup hoặc bundleOfAccounts hoặc chỉ các tài khoản đơn giản sẽ tốt hơn.

1. As we’ll see later on, even if the container *is* a List, it’s probably better not to encode the container type into the name.  
1. Như chúng ta sẽ thấy sau này, ngay cả khi vùng chứa là Danh sách, có lẽ tốt hơn là không mã hóa loại vùng chứa thành tên.

<https://www.facebook.com/groups/phattrienlaptrinh>

20 **Chapter 2: Meaningful Names**  
20 Chương 2: Những Cái Tên Ý Nghĩa

Beware of using names which vary in small ways. How long does it take to spot the subtle difference between a XYZControllerForEfficientHandlingOfStringsin one module and, somewhere a little more distant, XYZControllerForEfficientStorageOfStrings? The words have frightfully similar shapes.  
Cẩn thận với việc sử dụng các tên khác nhau theo những cách nhỏ. Mất bao lâu để phát hiện ra sự khác biệt tinh tế giữa một mô-đun XYZControllerForEfficientHandlingOfStrings và, ở đâu đó xa hơn một chút, XYZControllerForEfficientStorageOfStrings? Các từ có hình dạng giống nhau đến đáng sợ.

Spelling similar concepts similarly is *information*. Using inconsistent spellings is *dis-information*. With modern Java environments we enjoy automatic code completion. We write a few characters of a name and press some hotkey combination (if that) and are rewarded with a list of possible completions for that name. It is very helpful if names for very similar things sort together alphabetically and if the differences are very obvious, because the developer is likely to pick an object by name without seeing your copious comments or even the list of methods supplied by that class.  
Đánh vần các khái niệm tương tự nhau là thông tin. Sử dụng cách viết không nhất quán là thông tin không chính xác. Với các môi trường Java hiện đại, chúng tôi thích hoàn thành mã tự động. Chúng tôi viết một vài ký tự của tên và nhấn một số tổ hợp phím nóng (nếu vậy) và được thưởng một danh sách các cách hoàn thành có thể có cho tên đó. Sẽ rất hữu ích nếu tên của những thứ rất giống nhau được sắp xếp theo thứ tự bảng chữ cái và nếu sự khác biệt là rất rõ ràng, bởi vì nhà phát triển có thể chọn một đối tượng theo tên mà không nhìn thấy nhiều nhận xét của bạn hoặc thậm chí danh sách các phương thức do lớp đó cung cấp.

A truly awful example of disinformative names would be the use of lower-case Lor uppercase Oas variable names, especially in combination. The problem, of course, is that they look almost entirely like the constants one and zero, respectively.  
Một ví dụ thực sự tồi tệ về các tên sai thông tin sẽ là việc sử dụng các tên biến Oas viết hoa Lor viết thường, đặc biệt là khi kết hợp. Tất nhiên, vấn đề là chúng trông gần như hoàn toàn giống các hằng số một và không tương ứng.

int a = l; if ( O == l )

a = O1; else

l = 01;

The reader may think this a contrivance, but we have examined code where such things were abundant. In one case the author of the code suggested using a different font so that the differences were more obvious, a solution that would have to be passed down to all future developers as oral tradition or in a written document. The problem is conquered with ﬁnality and without creating new work products by a simple renaming.  
Người đọc có thể nghĩ rằng đây là một thủ thuật, nhưng chúng tôi đã kiểm tra mã nơi có rất nhiều thứ như vậy. Trong một trường hợp, tác giả của mã đã đề xuất sử dụng một phông chữ khác để sự khác biệt rõ ràng hơn, một giải pháp sẽ phải được truyền lại cho tất cả các nhà phát triển trong tương lai dưới dạng truyền miệng hoặc trong tài liệu viết. Vấn đề được giải quyết dứt điểm và không tạo ra sản phẩm công việc mới bằng cách đổi tên đơn giản.

[**Make Meaningful**](#_page_37_0)  
làm cho có ý nghĩa

[**Distinctions**](#_page_37_0)  
sự khác biệt

Programmers create problems for them-selves when they write code solely to sat-isfy a compiler or interpreter. For example, because you can’t use the same name to refer to two different things in the same scope, you might be tempted to change one name  
Các lập trình viên tự tạo ra vấn đề cho chính họ khi họ viết mã chỉ để phục vụ trình biên dịch hoặc trình thông dịch. Ví dụ: vì bạn không thể sử dụng cùng một tên để chỉ hai thứ khác nhau trong cùng một phạm vi, nên bạn có thể muốn thay đổi một tên

in an arbitrary way. Sometimes this is done by misspelling one, leading to the surprising situation where correcting spelling errors leads to an inability to compile.2  
một cách tùy ý. Đôi khi điều này được thực hiện bằng cách viết sai chính tả, dẫn đến một tình huống đáng ngạc nhiên là sửa lỗi chính tả dẫn đến không thể biên dịch.2

It is not sufﬁcient to add number series or noise words, even though the compiler is satisﬁed. If names must be different, then they should also mean something different.  
Việc thêm chuỗi số hoặc từ nhiễu là không đủ, mặc dù trình biên dịch đã thỏa mãn. Nếu tên phải khác, thì chúng cũng phải có ý nghĩa khác.

2. Consider, for example, the truly hideous practice of creating a variable named klass just because the name class was used for something else.  
2. Ví dụ, hãy xem xét một hành vi thực sự gớm ghiếc là tạo một biến có tên là klass chỉ vì tên lớp đã được sử dụng cho mục đích khác.

<https://www.facebook.com/groups/phattrienlaptrinh>

**Use Pronounceable Names** 21  
Sử dụng tên dễ phát âm 21

Number-series naming (a1, a2, .. aN)is the opposite of intentional naming. Such names are not disinformative—they are noninformative; they provide no clue to the author’s intention. Consider:  
Đặt tên theo dãy số (a1, a2, .. aN) ngược lại với đặt tên có chủ ý. Những cái tên như vậy không phải là sai thông tin - chúng không phải là thông tin; họ không cung cấp manh mối cho ý định của tác giả. Coi như:

public static void copyChars(char a1[], char a2[]) { for (int i = 0; i < a1.length; i++) {

a2[i] = a1[i]; }

}

This function reads much better when sourceand destinationare used for the argument names.  
Hàm này đọc tốt hơn nhiều khi nguồn và đích được sử dụng cho tên đối số.

Noise words are another meaningless distinction. Imagine that you have a Product class. If you have another called ProductInfoor ProductData, you have made the names dif-ferent without making them mean anything different. Infoand Dataare indistinct noise words like a, an, and the.  
Từ ồn ào là một sự phân biệt vô nghĩa khác. Hãy tưởng tượng rằng bạn có một lớp Sản phẩm. Nếu bạn có một cái khác được gọi là ProductInfoor ProductData, thì bạn đã làm cho các tên khác nhau mà không làm cho chúng có ý nghĩa khác đi. Thông tin và Dữ liệu là những từ nhiễu không rõ ràng như a, an và the.

Note that there is nothing wrong with using preﬁx conventions like aand theso long as they make a meaningful distinction. For example you might use afor all local variables and thefor all function arguments.3 The problem comes in when you decide to call a vari-able theZork because you already have another variable named zork.  
Lưu ý rằng không có gì sai khi sử dụng các quy ước tiền tố như aand the miễn là chúng tạo ra sự khác biệt có ý nghĩa. Ví dụ: bạn có thể sử dụng afor all biến cục bộ và thefor all đối số hàm.3 Vấn đề xảy ra khi bạn quyết định gọi một biến có thể là theZork vì bạn đã có một biến khác tên là zork.

Noise words are redundant. The word variableshould never appear in a variable name. The word tableshould never appear in a table name. How is NameStringbetter than Name? Would a Nameever be a ﬂoating point number? If so, it breaks an earlier rule about disinformation. Imagine ﬁnding one class named Customer and another named CustomerObject. What should you understand as the distinction? Which one will represent the best path to a customer’s payment history?  
Từ ồn ào là dư thừa. Từ biến không bao giờ được xuất hiện trong tên biến. Từ bảng không bao giờ được xuất hiện trong tên bảng. NameString tốt hơn Name như thế nào? Nameever có phải là một số dấu phẩy động không? Nếu vậy, nó sẽ phá vỡ một quy tắc trước đó về thông tin sai lệch. Hãy tưởng tượng bạn tìm thấy một lớp có tên là Khách hàng và một lớp khác có tên là Đối tượng khách hàng. Bạn nên hiểu thế nào là sự khác biệt? Cái nào sẽ đại diện cho đường dẫn tốt nhất đến lịch sử thanh toán của khách hàng?

There is an application we know of where this is illustrated. we’ve changed the names to protect the guilty, but here’s the exact form of the error:  
Có một ứng dụng mà chúng tôi biết nơi điều này được minh họa. chúng tôi đã thay đổi tên để bảo vệ người có tội, nhưng đây là hình thức chính xác của lỗi:

getActiveAccount(); getActiveAccounts(); getActiveAccountInfo();

How are the programmers in this project supposed to know which of these functions to call?  
Làm thế nào để các lập trình viên trong dự án này biết nên gọi hàm nào trong số các hàm này?

In the absence of speciﬁc conventions, the variable moneyAmountis indistinguishable from money, customerInfois indistinguishable from customer, accountDatais indistinguish-able from account, and theMessage is indistinguishable from message. Distinguish names in such a way that the reader knows what the differences offer.  
Trong trường hợp không có quy ước cụ thể, biến moneyAmount không thể phân biệt được với tiền, thông tin khách hàng không thể phân biệt được với khách hàng, Dữ liệu tài khoản không thể phân biệt với tài khoản và Tin nhắn không thể phân biệt được với tin nhắn. Phân biệt tên theo cách mà người đọc biết những gì sự khác biệt cung cấp.

[**Use Pronounceable Names**](#_page_37_0)  
Sử dụng tên dễ phát âm

Humans are good at words. A signiﬁcant part of our brains is dedicated to the concept of words. And words are, by deﬁnition, pronounceable. It would be a shame not to take  
Con người rất giỏi trong lời nói. Một phần quan trọng trong bộ não của chúng ta được dành riêng cho khái niệm về từ ngữ. Và các từ, theo định nghĩa, có thể phát âm được. Sẽ là một sự xấu hổ nếu không lấy

3. Uncle Bob used to do this in C++ but has given up the practice because modern IDEs make it unnecessary.  
3. Chú Bob đã từng làm điều này trong C++ nhưng đã từ bỏ thực hành này vì các IDE hiện đại khiến nó không cần thiết.

<https://www.facebook.com/groups/phattrienlaptrinh>

22 **Chapter 2: Meaningful Names**  
22 Chương 2: Tên Ý Nghĩa

advantage of that huge portion of our brains that has evolved to deal with spoken lan-guage. So make your names pronounceable.  
lợi thế của phần lớn bộ não của chúng ta đã phát triển để xử lý ngôn ngữ nói. Vì vậy, làm cho tên của bạn có thể phát âm được.

If you can’t pronounce it, you can’t discuss it without sounding like an idiot. “Well, over here on the bee cee arr three cee enn tee we have a pee ess zee kyew int, see?” This matters because programming is a social activity.  
Nếu bạn không thể phát âm nó, bạn không thể thảo luận về nó mà không giống như một thằng ngốc. “Chà, ở đây trên bee cee arr three cee enn tee, chúng ta có một pee ess zee kyew int, thấy không?” Điều này quan trọng vì lập trình là một hoạt động xã hội.

A company I know has genymdhms(generation date, year, month, day, hour, minute, and second) so they walked around saying “gen why emm dee aich emm ess”. I have an annoying habit of pronouncing everything as written, so I started saying “gen-yah-mudda-hims.” It later was being called this by a host of designers and analysts, and we still sounded silly. But we were in on the joke, so it was fun. Fun or not, we were tolerating poor naming. New developers had to have the variables explained to them, and then they spoke about it in silly made-up words instead of using proper English terms. Compare  
Một công ty mà tôi biết có genymdhms (ngày tạo, năm, tháng, ngày, giờ, phút và giây) nên họ đi loanh quanh và nói “gen sao emm dee aich emm ess”. Tôi có một thói quen khó chịu là phát âm mọi thứ như đã viết, vì vậy tôi bắt đầu nói “gen-yah-mudda-hims”. Sau đó, nó đã được gọi như vậy bởi một loạt các nhà thiết kế và nhà phân tích, và chúng tôi vẫn nghe có vẻ ngớ ngẩn. Nhưng chúng tôi đã tham gia vào trò đùa, vì vậy nó rất vui. Vui hay không, chúng tôi đã chấp nhận việc đặt tên kém. Các nhà phát triển mới phải giải thích các biến cho họ, và sau đó họ nói về nó bằng những từ bịa đặt ngớ ngẩn thay vì sử dụng các thuật ngữ tiếng Anh thích hợp. So sánh

class DtaRcrd102 { private Date genymdhms; private Date modymdhms;

private final String pszqint = "102"; /\* ... \*/

};

to

class Customer {

private Date generationTimestamp; private Date modificationTimestamp;; private final String recordId = "102"; /\* ... \*/

};

Intelligent conversation is now possible: “Hey, Mikey, take a look at this record! The gen-eration timestamp is set to tomorrow’s date! How can that be?”  
Hiện có thể trò chuyện thông minh: “Này, Mikey, hãy xem bản ghi này! Dấu thời gian thế hệ được đặt thành ngày mai! Làm thế nào mà có thể được?"

[**Use Searchable Names**](#_page_37_0)  
Sử dụng tên có thể tìm kiếm

Single-letter names and numeric constants have a particular problem in that they are not easy to locate across a body of text.  
Tên một chữ cái và hằng số có một vấn đề cụ thể là chúng không dễ định vị trên một phần văn bản.

One might easily grep for MAX\_CLASSES\_PER\_STUDENT, but the number 7 could be more troublesome. Searches may turn up the digit as part of ﬁle names, other constant deﬁni-tions, and in various expressions where the value is used with different intent. It is even worse when a constant is a long number and someone might have transposed digits, thereby creating a bug while simultaneously evading the programmer’s search.  
Người ta có thể dễ dàng grep cho MAX\_CLASSES\_PER\_STUDENT, nhưng số 7 có thể rắc rối hơn. Các tìm kiếm có thể đưa ra chữ số như một phần của tên tệp, các định nghĩa hằng số khác và trong các biểu thức khác nhau trong đó giá trị được sử dụng với mục đích khác nhau. Thậm chí còn tồi tệ hơn khi một hằng số là một số dài và ai đó có thể đã hoán vị các chữ số, do đó tạo ra một lỗi đồng thời trốn tránh sự tìm kiếm của lập trình viên.

Likewise, the name eis a poor choice for any variable for which a programmer might need to search. It is the most common letter in the English language and likely to show up in every passage of text in every program. In this regard, longer names trump shorter names, and any searchable name trumps a constant in code.  
Tương tự như vậy, tên eis là một lựa chọn tồi cho bất kỳ biến nào mà một lập trình viên có thể cần tìm kiếm. Đây là chữ cái phổ biến nhất trong tiếng Anh và có khả năng xuất hiện trong mọi đoạn văn bản trong mọi chương trình. Về vấn đề này, tên dài hơn tên ngắn hơn và bất kỳ tên có thể tìm kiếm nào cũng vượt trội hơn một hằng số trong mã.

My personal preference is that single-letter names can ONLY be used as local vari-ables inside short methods. *The length of a name should correspond to the size of its scope*  
Sở thích cá nhân của tôi là các tên có một chữ cái CHỈ có thể được sử dụng làm biến cục bộ bên trong các phương thức ngắn. Độ dài của tên phải tương ứng với kích thước phạm vi của nó

<https://www.facebook.com/groups/phattrienlaptrinh>

**Avoid Encodings** 23  
Tránh mã hóa 23

[N5]. If a variable or constant might be seen or used in multiple places in a body of code, it is imperative to give it a search-friendly name. Once again compare  
[N5]. Nếu một biến hoặc hằng số có thể được nhìn thấy hoặc sử dụng ở nhiều nơi trong một đoạn mã, thì bắt buộc phải đặt cho nó một tên thân thiện với tìm kiếm. Một lần nữa so sánh

for (int j=0; j<34; j++) { s += (t[j]\*4)/5;

}

to

int realDaysPerIdealDay = 4; const int WORK\_DAYS\_PER\_WEEK = 5; int sum = 0;

for (int j=0; j < NUMBER\_OF\_TASKS; j++) {

int realTaskDays = taskEstimate[j] \* realDaysPerIdealDay; int realTaskWeeks = (realdays / WORK\_DAYS\_PER\_WEEK);

sum += realTaskWeeks; }

Note that sum, above, is not a particularly useful name but at least is searchable. The intentionally named code makes for a longer function, but consider how much easier it will be to ﬁnd WORK\_DAYS\_PER\_WEEKthan to ﬁnd all the places where 5 was used and ﬁlter the list down to just the instances with the intended meaning.  
Lưu ý rằng tổng ở trên không phải là một tên đặc biệt hữu ích nhưng ít nhất là có thể tìm kiếm được. Mã được đặt tên có chủ ý tạo ra một hàm dài hơn, nhưng hãy xem xét việc tìm WORK\_DAYS\_PER\_WEEK sẽ dễ dàng hơn nhiều so với tìm tất cả những nơi mà 5 đã được sử dụng và lọc danh sách xuống chỉ những trường hợp có ý nghĩa dự kiến.

[**Avoid Encodings**](#_page_91_0)  
Tránh mã hóa

We have enough encodings to deal with without adding more to our burden. Encoding type or scope information into names simply adds an extra burden of deciphering. It hardly seems reasonable to require each new employee to learn yet another encoding “lan-guage” in addition to learning the (usually considerable) body of code that they’ll be work-ing in. It is an unnecessary mental burden when trying to solve a problem. Encoded names are seldom pronounceable and are easy to mis-type.  
Chúng tôi có đủ mã hóa để xử lý mà không cần thêm gánh nặng. Mã hóa thông tin loại hoặc phạm vi thành tên chỉ đơn giản là tăng thêm gánh nặng giải mã. Có vẻ khó hợp lý khi yêu cầu mỗi nhân viên mới học thêm một “ngôn ngữ” mã hóa khác ngoài việc học nội dung mã (thường là đáng kể) mà họ sẽ làm việc. Đó là một gánh nặng tinh thần không cần thiết khi cố gắng Giải quyết vấn đề. Tên được mã hóa hiếm khi phát âm được và dễ gõ nhầm.

[**Hungarian Notation**](#_page_91_0)  
Ký hiệu Hungary

In days of old, when we worked in name-length-challenged languages, we violated this rule out of necessity, and with regret. Fortran forced encodings by making the ﬁrst letter a code for the type. Early versions of BASIC allowed only a letter plus one digit. Hungarian Notation (HN) took this to a whole new level.  
Ngày xưa, khi chúng tôi làm việc với các ngôn ngữ thách thức độ dài tên, chúng tôi đã vi phạm quy tắc này một cách không cần thiết và rất tiếc. Fortran bắt buộc mã hóa bằng cách biến chữ cái đầu tiên thành mã cho loại. Các phiên bản đầu tiên của BASIC chỉ cho phép một chữ cái cộng với một chữ số. Ký hiệu Hungary (HN) đã đưa điều này lên một cấp độ hoàn toàn mới.

HN was considered to be pretty important back in the Windows C API, when every-thing was an integer handle or a long pointer or a voidpointer, or one of several implemen-tations of “string” (with different uses and attributes). The compiler did not check types in those days, so the programmers needed a crutch to help them remember the types.  
HN được coi là khá quan trọng trong API Windows C, khi mọi thứ là một số nguyên xử lý hoặc một con trỏ dài hoặc một con trỏ trống hoặc một trong số các triển khai của “chuỗi” (với các cách sử dụng và thuộc tính khác nhau). Trình biên dịch không kiểm tra các loại trong những ngày đó, vì vậy các lập trình viên cần một cái nạng để giúp họ ghi nhớ các loại.

In modern languages we have much richer type systems, and the compilers remember and enforce the types. What’s more, there is a trend toward smaller classes and shorter functions so that people can usually see the point of declaration of each variable they’re using.  
Trong các ngôn ngữ hiện đại, chúng ta có nhiều hệ thống kiểu phong phú hơn và trình biên dịch ghi nhớ và thực thi các kiểu. Hơn nữa, có xu hướng hướng tới các lớp nhỏ hơn và các hàm ngắn hơn để mọi người thường có thể thấy điểm khai báo của từng biến mà họ đang sử dụng.

<https://www.facebook.com/groups/phattrienlaptrinh>

24 **Chapter 2: Meaningful Names**  
24 Chương 2: Tên Ý Nghĩa

Java programmers don’t need type encoding. Objects are strongly typed, and editing environments have advanced such that they detect a type error long before you can run a compile! So nowadays HN and other forms of type encoding are simply impediments. They make it harder to change the name or type of a variable, function, or class. They make it harder to read the code. And they create the possibility that the encoding system will mislead the reader.  
Các lập trình viên Java không cần mã hóa kiểu. Các đối tượng được gõ mạnh và môi trường chỉnh sửa đã nâng cao để chúng phát hiện lỗi loại từ lâu trước khi bạn có thể chạy biên dịch! Vì vậy, ngày nay HN và các hình thức mã hóa kiểu khác chỉ đơn giản là trở ngại. Chúng làm cho việc thay đổi tên hoặc loại biến, hàm hoặc lớp trở nên khó khăn hơn. Chúng làm cho mã khó đọc hơn. Và chúng tạo ra khả năng hệ thống mã hóa sẽ đánh lừa người đọc.

PhoneNumber phoneString;

// name not changed when type changed!

[**Member Preﬁxes**](#_page_91_0)  
Tiền tố thành viên

You also don’t need to preﬁx member variables with m\_anymore. Your classes and func-tions should be small enough that you don’t need them. And you should be using an edit-ing environment that highlights or colorizes members to make them distinct.  
Bạn cũng không cần thêm tiền tố vào các biến thành viên với m\_anymore. Các lớp và chức năng của bạn phải đủ nhỏ để bạn không cần đến chúng. Và bạn nên sử dụng một môi trường chỉnh sửa làm nổi bật hoặc tô màu các thành viên để làm cho chúng trở nên khác biệt.

public class Part {

private String m\_dsc; // The textual description void setName(String name) {

m\_dsc = name; }

} \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

public class Part { String description;

void setDescription(String description) { this.description = description;

} }

Besides, people quickly learn to ignore the preﬁx (or sufﬁx) to see the meaningful part of the name. The more we read the code, the less we see the preﬁxes. Eventually the preﬁxes become unseen clutter and a marker of older code.  
Bên cạnh đó, mọi người nhanh chóng học cách bỏ qua tiền tố (hoặc hậu tố) để thấy phần ý nghĩa của tên. Càng đọc mã, chúng ta càng thấy ít tiền tố hơn. Cuối cùng, các tiền tố trở nên lộn xộn không nhìn thấy được và là dấu hiệu của mã cũ hơn.

[**Interfaces and Implementations**](#_page_91_0)  
Giao diện và triển khai

These are sometimes a special case for encodings. For example, say you are building an ABSTRACT FACTORY for the creation of shapes. This factory will be an interface and will be implemented by a concrete class. What should you name them? IShapeFactoryand ShapeFactory? I prefer to leave interfaces unadorned. The preceding I, so common in today’s legacy wads, is a distraction at best and too much information at worst. I don’t want my users knowing that I’m handing them an interface. I just want them to know that it’s a ShapeFactory. So if I must encode either the interface or the implementation, I choose the implementation. Calling it ShapeFactoryImp, or even the hideous CShapeFactory, is pref-erable to encoding the interface.  
Đây đôi khi là một trường hợp đặc biệt cho mã hóa. Ví dụ: giả sử bạn đang xây dựng NHÀ MÁY TÓM TẮT để tạo các hình dạng. Nhà máy này sẽ là một giao diện và sẽ được triển khai bởi một lớp cụ thể. Bạn nên đặt tên cho chúng là gì? IShapeFactory và ShapeFactory? Tôi thích để giao diện không trang trí. Cái tôi đứng trước, rất phổ biến trong các trang kế thừa ngày nay, tốt nhất là gây xao nhãng và tệ nhất là quá nhiều thông tin. Tôi không muốn người dùng của mình biết rằng tôi đang cung cấp cho họ một giao diện. Tôi chỉ muốn họ biết rằng đó là ShapeFactory. Vì vậy, nếu tôi phải mã hóa giao diện hoặc triển khai, tôi sẽ chọn triển khai. Gọi nó là ShapeFactoryImp, hoặc thậm chí là CShapeFactory gớm ghiếc, thích hợp hơn để mã hóa giao diện.

<https://www.facebook.com/groups/phattrienlaptrinh>

**Method Names** 25  
Tên Phương thức 25

[**Avoid Mental Mapping**](#_page_91_0)  
Tránh bản đồ tinh thần

Readers shouldn’t have to mentally translate your names into other names they already know. This problem generally arises from a choice to use neither problem domain terms nor solution domain terms.  
Người đọc không cần phải dịch nhẩm tên của bạn sang những tên khác mà họ đã biết. Vấn đề này thường phát sinh từ lựa chọn không sử dụng thuật ngữ miền vấn đề cũng như thuật ngữ miền giải pháp.

This is a problem with single-letter variable names. Certainly a loop counter may be named ior jor k(though never l!) if its scope is very small and no other names can con-ﬂict with it. This is because those single-letter names for loop counters are traditional. However, in most other contexts a single-letter name is a poor choice; it’s just a place holder that the reader must mentally map to the actual concept. There can be no worse rea-son for using the name c than because a and b were already taken.  
Đây là một vấn đề với tên biến một chữ cái. Chắc chắn một bộ đếm vòng lặp có thể được đặt tên là ior jor k (mặc dù không bao giờ là l!) nếu phạm vi của nó rất nhỏ và không có tên nào khác có thể xung đột với nó. Điều này là do những tên đơn chữ cái cho bộ đếm vòng lặp là truyền thống. Tuy nhiên, trong hầu hết các bối cảnh khác, tên có một chữ cái là một lựa chọn tồi; nó chỉ là một trình giữ chỗ mà người đọc phải ánh xạ trong đầu tới khái niệm thực tế. Không có lý do nào tồi tệ hơn cho việc sử dụng tên c hơn là vì a và b đã được sử dụng.

In general programmers are pretty smart people. Smart people sometimes like to show off their smarts by demonstrating their mental juggling abilities. After all, if you can reli-ably remember that r is the lower-cased version of the url with the host and scheme removed, then you must clearly be very smart.  
Nói chung các lập trình viên là những người khá thông minh. Những người thông minh đôi khi thích thể hiện sự thông minh của họ bằng cách thể hiện khả năng tung hứng tinh thần của họ. Rốt cuộc, nếu bạn có thể nhớ một cách đáng tin cậy rằng r là phiên bản chữ thường của url đã loại bỏ máy chủ và lược đồ, thì rõ ràng bạn phải rất thông minh.

One difference between a smart programmer and a professional programmer is that the professional understands that *clarity is king*. Professionals use their powers for good and write code that others can understand.  
Một điểm khác biệt giữa một lập trình viên thông minh và một lập trình viên chuyên nghiệp là họ hiểu rằng sự rõ ràng là quan trọng nhất. Các chuyên gia sử dụng quyền hạn của họ cho mục đích tốt và viết mã mà người khác có thể hiểu được.

[**Class Names**](#_page_91_0)  
Tên Lớp

Classes and objects should have noun or noun phrase names like Customer, WikiPage, Account, and AddressParser. Avoid words like Manager, Processor, Data, or Infoin the name of a class. A class name should not be a verb.  
Các lớp và đối tượng phải có tên danh từ hoặc cụm danh từ như Khách hàng, WikiPage, Tài khoản và Trình phân tích địa chỉ. Tránh các từ như Trình quản lý, Bộ xử lý, Dữ liệu hoặc Thông tin trong tên của lớp. Một tên lớp không nên là một động từ.

[**Method Names**](#_page_91_0)  
Tên phương pháp

Methods should have verb or verb phrase names like postPayment, deletePage, or save. Accessors, mutators, and predicates should be named for their value and preﬁxed with get, set, and is according to the javabean standard.4  
Các phương thức phải có tên động từ hoặc cụm động từ như postPayment, deletePage hoặc save. Các bộ truy cập, bộ biến đổi và vị từ phải được đặt tên theo giá trị của chúng và có tiền tố là get, set và theo tiêu chuẩn javabean.4

string name = employee.getName(); customer.setName("mike");

if (paycheck.isPosted())...

When constructors are overloaded, use static factory methods with names that describe the arguments. For example,  
Khi các hàm tạo bị quá tải, hãy sử dụng các phương thức xuất xưởng tĩnh có tên mô tả các đối số. Ví dụ,

Complex fulcrumPoint = Complex.FromRealNumber(23.0); is generally better than

Complex fulcrumPoint = new Complex(23.0);

Consider enforcing their use by making the corresponding constructors private.  
Xem xét việc thực thi việc sử dụng chúng bằng cách đặt các hàm tạo tương ứng ở chế độ riêng tư.

4. <http://java.sun.com/products/javabeans/docs/spec.html>

<https://www.facebook.com/groups/phattrienlaptrinh>

26 **Chapter 2: Meaningful Names**  
26 Chương 2: Những Cái Tên Có Ý Nghĩa

[**Don’t Be Cute**](#_page_91_0)  
Đừng dễ thương

If names are too clever, they will be memorable only to people who share the author’s sense of humor, and only as long as these people remember the joke. Will they know what the function named HolyHandGrenadeis supposed to do? Sure, it’s cute, but maybe in this case DeleteItems might be a better name. Choose clarity over entertainment value.  
Nếu những cái tên quá thông minh, chúng sẽ chỉ đáng nhớ đối với những người có cùng khiếu hài hước với tác giả và chỉ khi những người này nhớ được câu chuyện cười. Liệu họ có biết chức năng có tên HolyHandGrenadeis phải làm gì không? Chắc chắn, nó rất dễ thương, nhưng có lẽ trong trường hợp này, DeleteItems có thể là một cái tên hay hơn. Chọn sự rõ ràng hơn giá trị giải trí.

Cuteness in code often appears in the form of colloquialisms or slang. For example, don’t use the name whack()to mean kill(). Don’t tell little culture-dependent jokes like eatMyShorts()to mean abort().  
Sự dễ thương trong mã thường xuất hiện dưới dạng thông tục hoặc tiếng lóng. Ví dụ: không sử dụng tên whack() có nghĩa là kill(). Đừng kể những câu chuyện cười phụ thuộc vào văn hóa như eatMyShorts() có nghĩa là hủy bỏ().

Say what you mean. Mean what you say.  
Nói những gì bạn có nghĩa là. Có nghĩa là những gì bạn nói.

[**Pick One Word per Concept**](#_page_91_0)  
Chọn một từ cho mỗi khái niệm

Pick one word for one abstract concept and stick with it. For instance, it’s confusing to have fetch, retrieve,and getas equivalent methods of different classes. How do you remember which method name goes with which class? Sadly, you often have to remember which company, group, or individual wrote the library or class in order to remember which term was used. Otherwise, you spend an awful lot of time browsing through headers and previous code samples.  
Chọn một từ cho một khái niệm trừu tượng và gắn bó với nó. Chẳng hạn, thật khó hiểu khi có các phương thức tìm nạp, truy xuất và getas tương đương của các lớp khác nhau. Làm thế nào để bạn nhớ tên phương thức nào đi với lớp nào? Đáng buồn thay, bạn thường phải nhớ công ty, nhóm hoặc cá nhân nào đã viết thư viện hoặc lớp học để nhớ thuật ngữ nào đã được sử dụng. Mặt khác, bạn dành rất nhiều thời gian để duyệt qua các tiêu đề và các mẫu mã trước đó.

Modern editing environments like Eclipse and IntelliJ-provide context-sensitive clues, such as the list of methods you can call on a given object. But note that the list doesn’t usu-ally give you the comments you wrote around your function names and parameter lists. You are lucky if it gives the parameter *names* from function declarations. The function names have to stand alone, and they have to be consistent in order for you to pick the cor-rect method without any additional exploration.  
Các môi trường chỉnh sửa hiện đại như Eclipse và IntelliJ-cung cấp manh mối theo ngữ cảnh, chẳng hạn như danh sách các phương thức bạn có thể gọi trên một đối tượng nhất định. Nhưng lưu ý rằng danh sách thường không cung cấp cho bạn nhận xét mà bạn đã viết xung quanh tên hàm và danh sách tham số của mình. Bạn thật may mắn nếu nó cung cấp tên tham số từ các khai báo hàm. Các tên hàm phải đứng độc lập và chúng phải nhất quán để bạn có thể chọn phương thức chính xác mà không cần thăm dò thêm.

Likewise, it’s confusing to have a controllerand a managerand a driverin the same code base. What is the essential difference between a DeviceManager and a Protocol-Controller? Why are both not controllers or both not managers? Are they both Drivers really? The name leads you to expect two objects that have very different type as well as having different classes.  
Tương tự như vậy, thật khó hiểu khi có một bộ điều khiển và một trình quản lý và một trình điều khiển trong cùng một cơ sở mã. Sự khác biệt cơ bản giữa Trình quản lý thiết bị và Bộ điều khiển giao thức là gì? Tại sao cả hai không phải là bộ điều khiển hoặc cả hai không phải là người quản lý? Cả hai đều là Trình điều khiển thực sự? Tên khiến bạn mong đợi hai đối tượng có kiểu rất khác nhau cũng như có các lớp khác nhau.

A consistent lexicon is a great boon to the programmers who must use your code.  
Một từ vựng nhất quán là một lợi ích tuyệt vời cho các lập trình viên, những người phải sử dụng mã của bạn.

[**Don’t Pun**](#_page_91_0)  
Đừng chơi chữ

Avoid using the same word for two purposes. Using the same term for two different ideas is essentially a pun.  
Tránh sử dụng cùng một từ cho hai mục đích. Sử dụng cùng một thuật ngữ cho hai ý tưởng khác nhau về cơ bản là một cách chơi chữ.

<https://www.facebook.com/groups/phattrienlaptrinh>

**Add Meaningful Context** 27  
Thêm bối cảnh có ý nghĩa 27

If you follow the “one word per concept” rule, you could end up with many classes that have, for example, an addmethod. As long as the parameter lists and return values of the various add methods are semantically equivalent, all is well.  
Nếu bạn tuân theo quy tắc “một từ cho mỗi khái niệm”, bạn có thể kết thúc với nhiều lớp có phương thức bổ sung chẳng hạn. Miễn là danh sách tham số và giá trị trả về của các phương thức thêm khác nhau tương đương về mặt ngữ nghĩa, thì tất cả đều ổn.

However one might decide to use the word addfor “consistency” when he or she is not in fact adding in the same sense. Let’s say we have many classes where addwill create a new value by adding or concatenating two existing values. Now let’s say we are writing a new class that has a method that puts its single parameter into a collection. Should we call this method add? It might seem consistent because we have so many other addmethods, but in this case, the semantics are different, so we should use a name like insertor append instead. To call the new method add would be a pun.  
Tuy nhiên, một người có thể quyết định sử dụng từ bổ sung cho “tính nhất quán” khi thực tế người đó không bổ sung theo nghĩa tương tự. Giả sử chúng ta có nhiều lớp trong đó addwill tạo một giá trị mới bằng cách thêm hoặc nối hai giá trị hiện có. Bây giờ, giả sử chúng ta đang viết một lớp mới có một phương thức đặt tham số duy nhất của nó vào một tập hợp. Chúng ta có nên gọi phương thức này là add không? Nó có vẻ nhất quán bởi vì chúng ta có rất nhiều phương thức bổ sung khác, nhưng trong trường hợp này, ngữ nghĩa là khác nhau, vì vậy chúng ta nên sử dụng một tên như phần bổ sung bộ chèn để thay thế. Để gọi phương thức mới add sẽ là một cách chơi chữ.

Our goal, as authors, is to make our code as easy as possible to understand. We want our code to be a quick skim, not an intense study. We want to use the popular paperback model whereby the author is responsible for making himself clear and not the academic model where it is the scholar’s job to dig the meaning out of the paper.  
Mục tiêu của chúng tôi, với tư cách là tác giả, là làm cho mã của chúng tôi dễ hiểu nhất có thể. Chúng tôi muốn mã của mình được đọc lướt qua nhanh chóng, không phải là một nghiên cứu chuyên sâu. Chúng tôi muốn sử dụng mô hình bìa mềm phổ biến, theo đó tác giả chịu trách nhiệm trình bày rõ ràng chứ không phải mô hình hàn lâm, nơi công việc của học giả là tìm hiểu ý nghĩa của bài báo.

[**Use Solution Domain Names**](#_page_91_0)  
Sử dụng tên miền giải pháp

Remember that the people who read your code will be programmers. So go ahead and use computer science (CS) terms, algorithm names, pattern names, math terms, and so forth. It is not wise to draw every name from the problem domain because we don’t want our coworkers to have to run back and forth to the customer asking what every name means when they already know the concept by a different name.  
Hãy nhớ rằng những người đọc mã của bạn sẽ là lập trình viên. Vì vậy, hãy tiếp tục và sử dụng các thuật ngữ khoa học máy tính (CS), tên thuật toán, tên mẫu, thuật ngữ toán học, v.v. Sẽ không khôn ngoan nếu rút ra mọi tên từ miền vấn đề vì chúng tôi không muốn đồng nghiệp của mình phải chạy tới chạy lui hỏi khách hàng ý nghĩa của từng tên khi họ đã biết khái niệm bằng một tên khác.

The name AccountVisitormeans a great deal to a programmer who is familiar with the VISITOR pattern. What programmer would not know what a JobQueuewas? There are lots of very technical things that programmers have to do. Choosing technical names for those things is usually the most appropriate course.  
Cái tên AccountVisitor có ý nghĩa rất lớn đối với một lập trình viên đã quen thuộc với mẫu VISITOR. Lập trình viên nào lại không biết JobQueuewas là gì? Có rất nhiều thứ rất kỹ thuật mà các lập trình viên phải làm. Chọn tên kỹ thuật cho những thứ đó thường là cách thích hợp nhất.

[**Use Problem Domain Names**](#_page_91_0)  
Sử dụng tên miền có vấn đề

When there is no “programmer-eese” for what you’re doing, use the name from the prob-lem domain. At least the programmer who maintains your code can ask a domain expert what it means.  
Khi không có "lập trình viên-eese" cho những gì bạn đang làm, hãy sử dụng tên từ miền có vấn đề. Ít nhất lập trình viên duy trì mã của bạn có thể hỏi chuyên gia tên miền ý nghĩa của nó.

Separating solution and problem domain concepts is part of the job of a good pro-grammer and designer. The code that has more to do with problem domain concepts should have names drawn from the problem domain.  
Tách các khái niệm miền giải pháp và vấn đề là một phần công việc của một lập trình viên và nhà thiết kế giỏi. Mã liên quan nhiều hơn đến các khái niệm miền vấn đề nên có tên được rút ra từ miền vấn đề.

[**Add Meaningful Context**](#_page_91_0)  
Thêm bối cảnh có ý nghĩa

There are a few names which are meaningful in and of themselves—most are not. Instead, you need to place names in context for your reader by enclosing them in well-named classes, functions, or namespaces. When all else fails, then preﬁxing the name may be nec-essary as a last resort.  
Có một vài cái tên tự chúng có ý nghĩa—hầu hết thì không. Thay vào đó, bạn cần đặt tên trong ngữ cảnh cho người đọc của mình bằng cách đặt chúng trong các lớp, hàm hoặc không gian tên được đặt tên hợp lý. Khi vẫn thất bại, thì tiền tố tên có thể là cần thiết như là phương sách cuối cùng.

<https://www.facebook.com/groups/phattrienlaptrinh>

28 **Chapter 2: Meaningful Names**  
28 Chương 2: Những Cái Tên Có Ý Nghĩa

Imagine that you have variables named firstName, lastName, street, houseNumber, city, state, and zipcode. Taken together it’s pretty clear that they form an address. But what if you just saw the statevariable being used alone in a method? Would you automatically infer that it was part of an address?  
Hãy tưởng tượng rằng bạn có các biến có tên là FirstName, LastName, street, houseNumber, city, state và zipcode. Khi kết hợp với nhau, rõ ràng là chúng tạo thành một địa chỉ. Nhưng nếu bạn chỉ thấy biến trạng thái được sử dụng một mình trong một phương thức thì sao? Bạn có tự động suy luận rằng đó là một phần của địa chỉ không?

You can add context by using preﬁxes: addrFirstName, addrLastName, addrState, and so on. At least readers will understand that these variables are part of a larger structure. Of course, a better solution is to create a class named Address. Then, even the compiler knows that the variables belong to a bigger concept.  
Bạn có thể thêm ngữ cảnh bằng cách sử dụng tiền tố: addrFirstName, addrLastName, addrState, v.v. Ít nhất độc giả sẽ hiểu rằng những biến số này là một phần của cấu trúc lớn hơn. Tất nhiên, giải pháp tốt hơn là tạo một lớp có tên là Địa chỉ. Sau đó, ngay cả trình biên dịch cũng biết rằng các biến thuộc về một khái niệm lớn hơn.

Consider the method in Listing 2-1. Do the variables need a more meaningful con-text? The function name provides only part of the context; the algorithm provides the rest. Once you read through the function, you see that the three variables, number, verb, and pluralModifier, are part of the “guess statistics” message. Unfortunately, the context must be inferred. When you ﬁrst look at the method, the meanings of the variables are opaque.  
Hãy xem xét phương pháp trong Liệt kê 2-1. Các biến có cần ngữ cảnh có ý nghĩa hơn không? Tên hàm chỉ cung cấp một phần ngữ cảnh; thuật toán cung cấp phần còn lại. Sau khi đọc qua hàm này, bạn sẽ thấy rằng ba biến số, số, động từ và Công cụ sửa đổi số nhiều là một phần của thông báo "đoán số liệu thống kê". Thật không may, bối cảnh phải được suy ra. Khi bạn lần đầu tiên nhìn vào phương pháp, ý nghĩa của các biến không rõ ràng.

**Listing 2-1**  
Liệt kê 2-1

**Variables with unclear context.  
Các biến có ngữ cảnh không rõ ràng.**

private void printGuessStatistics(char candidate, int count) { String number;

String verb;

String pluralModifier; if (count == 0) {

number = "no"; verb = "are";

pluralModifier = "s"; } else if (count == 1) {

number = "1"; verb = "is";

pluralModifier = ""; } else {

number = Integer.toString(count); verb = "are";

pluralModifier = "s"; }

String guessMessage = String.format(

"There %s %s %s%s", verb, number, candidate, pluralModifier );

print(guessMessage); }

The function is a bit too long and the variables are used throughout. To split the func-tion into smaller pieces we need to create a GuessStatisticsMessageclass and make the three variables ﬁelds of this class. This provides a clear context for the three variables. They are *deﬁnitively* part of the GuessStatisticsMessage. The improvement of context also allows the algorithm to be made much cleaner by breaking it into many smaller functions. (See Listing 2-2.)  
Hàm hơi dài và các biến được sử dụng xuyên suốt. Để chia chức năng thành các phần nhỏ hơn, chúng ta cần tạo một lớp GuessStatisticsMessage và tạo ba trường biến của lớp này. Điều này cung cấp một bối cảnh rõ ràng cho ba biến. Chúng chắc chắn là một phần của GuessStatisticsMessage. Việc cải thiện ngữ cảnh cũng cho phép thuật toán trở nên gọn gàng hơn nhiều bằng cách chia nó thành nhiều hàm nhỏ hơn. (Xem Liệt kê 2-2.)

<https://www.facebook.com/groups/phattrienlaptrinh>

**Don’t Add Gratuitous Context** 29  
Không thêm bối cảnh vô cớ 29

**Listing 2-2  
Liệt kê 2-2**

**Variables have a context.  
Các biến có một bối cảnh.**

public class GuessStatisticsMessage { private String number;

private String verb;

private String pluralModifier;

public String make(char candidate, int count) { createPluralDependentMessageParts(count); return String.format(

"There %s %s %s%s",

verb, number, candidate, pluralModifier ); }

private void createPluralDependentMessageParts(int count) { if (count == 0) {

thereAreNoLetters(); } else if (count == 1) {

thereIsOneLetter(); } else {

thereAreManyLetters(count); }

}

private void thereAreManyLetters(int count) { number = Integer.toString(count);

verb = "are"; pluralModifier = "s";

}

private void thereIsOneLetter() { number = "1";

verb = "is"; pluralModifier = "";

}

private void thereAreNoLetters() { number = "no";

verb = "are"; pluralModifier = "s";

} }

[**Don’t Add Gratuitous Context**](#_page_91_0)  
Không thêm bối cảnh vô cớ

In an imaginary application called “Gas Station Deluxe,” it is a bad idea to preﬁx every class with GSD. Frankly, you are working against your tools. You type Gand press the com-pletion key and are rewarded with a mile-long list of every class in the system. Is that wise? Why make it hard for the IDE to help you?  
Trong một ứng dụng tưởng tượng có tên là “Gas Station Deluxe”, sẽ là một ý tưởng tồi khi đặt trước mỗi lớp bằng GSD. Thành thật mà nói, bạn đang làm việc chống lại các công cụ của bạn. Bạn gõ Gand nhấn phím hoàn thành và được thưởng một danh sách dài hàng dặm về mọi lớp trong hệ thống. Điều đó có khôn ngoan không? Tại sao làm cho IDE khó giúp bạn?

Likewise, say you invented a MailingAddressclass in GSD’s accounting module, and you named it GSDAccountAddress. Later, you need a mailing address for your customer con-tact application. Do you use GSDAccountAddress? Does it sound like the right name? Ten of 17 characters are redundant or irrelevant.  
Tương tự như vậy, giả sử bạn đã tạo ra một lớp Địa chỉ thư trong mô-đun kế toán của GSD và bạn đặt tên cho nó là GSDAccountAddress. Sau đó, bạn cần một địa chỉ gửi thư cho ứng dụng liên hệ với khách hàng của mình. Bạn có sử dụng GSDAccountAddress không? Nghe có vẻ đúng tên phải không? Mười trong số 17 ký tự là dư thừa hoặc không liên quan.

<https://www.facebook.com/groups/phattrienlaptrinh>

30 **Chapter 2: Meaningful Names**  
30 Chương 2: Tên Ý Nghĩa

Shorter names are generally better than longer ones, so long as they are clear. Add no more context to a name than is necessary.  
Tên ngắn hơn thường tốt hơn tên dài, miễn là chúng rõ ràng. Không thêm ngữ cảnh vào tên hơn mức cần thiết.

The names accountAddressand customerAddressare ﬁne names for instances of the class Addressbut could be poor names for classes. Addressis a ﬁne name for a class. If I need to differentiate between MAC addresses, port addresses, and Web addresses, I might consider PostalAddress, MAC, and URI. The resulting names are more precise, which is the point of all naming.  
Tên tài khoảnĐịa chỉ và địa chỉ khách hàng là những tên tốt cho các thể hiện của lớp Địa chỉ nhưng có thể là những tên kém cho các lớp. Địa chỉ là một tên đẹp cho một lớp. Nếu tôi cần phân biệt giữa địa chỉ MAC, địa chỉ cổng và địa chỉ Web, tôi có thể xem xét PostalAddress, MAC và URI. Các tên kết quả chính xác hơn, đó là điểm của tất cả các cách đặt tên.

[**Final Words**](#_page_91_0)  
Từ cuối cùng

The hardest thing about choosing good names is that it requires good descriptive skills and a shared cultural background. This is a teaching issue rather than a technical, business, or management issue. As a result many people in this ﬁeld don’t learn to do it very well.  
Điều khó nhất khi chọn tên hay là nó đòi hỏi kỹ năng mô tả tốt và nền tảng văn hóa chung. Đây là một vấn đề giảng dạy hơn là một vấn đề kỹ thuật, kinh doanh, hoặc quản lý. Kết quả là nhiều người trong lĩnh vực này không học cách làm tốt lắm.

People are also afraid of renaming things for fear that some other developers will object. We do not share that fear and ﬁnd that we are actually grateful when names change (for the better). Most of the time we don’t really memorize the names of classes and meth-ods. We use the modern tools to deal with details like that so we can focus on whether the code reads like paragraphs and sentences, or at least like tables and data structure (a sen-tence isn’t always the best way to display data). You will probably end up surprising some-one when you rename, just like you might with any other code improvement. Don’t let it stop you in your tracks.  
Mọi người cũng ngại đổi tên vì sợ một số nhà phát triển khác sẽ phản đối. Chúng tôi không chia sẻ nỗi sợ hãi đó và thấy rằng chúng tôi thực sự biết ơn khi tên thay đổi (để tốt hơn). Hầu hết thời gian chúng tôi không thực sự ghi nhớ tên của các lớp và phương pháp. Chúng tôi sử dụng các công cụ hiện đại để xử lý các chi tiết như vậy để chúng tôi có thể tập trung vào việc mã có đọc giống như đoạn văn và câu hay ít nhất giống như bảng và cấu trúc dữ liệu hay không (một câu không phải lúc nào cũng là cách tốt nhất để hiển thị dữ liệu). Bạn có thể sẽ khiến ai đó ngạc nhiên khi đổi tên, giống như bạn có thể làm với bất kỳ cải tiến mã nào khác. Đừng để nó ngăn bạn theo dõi.

Follow some of these rules and see whether you don’t improve the readability of your code. If you are maintaining someone else’s code, use refactoring tools to help resolve these problems. It will pay off in the short term and continue to pay in the long run.  
Thực hiện theo một số quy tắc này và xem liệu bạn có cải thiện khả năng đọc mã của mình hay không. Nếu bạn đang bảo trì mã của người khác, hãy sử dụng các công cụ tái cấu trúc để giúp giải quyết những vấn đề này. Nó sẽ trả hết trong ngắn hạn và tiếp tục trả trong dài hạn.

<https://www.facebook.com/groups/phattrienlaptrinh>



[**3**](#_page_91_0)

[**Functions**](#_page_91_0)  
Chức năng

In the early days of programming we composed our systems of routines and subroutines. Then, in the era of Fortran and PL/1 we composed our systems of programs, subprograms, and functions. Nowadays only the function survives from those early days. Functions are the ﬁrst line of organization in any program. Writing them well is the topic of this chapter.  
Trong những ngày đầu lập trình, chúng tôi đã soạn các hệ thống quy trình và quy trình con của mình. Sau đó, trong kỷ nguyên của Fortran và PL/1, chúng tôi đã soạn thảo các hệ thống chương trình, chương trình con và chức năng của mình. Ngày nay chỉ có chức năng tồn tại từ những ngày đầu đó. Hàm là dòng tổ chức đầu tiên trong bất kỳ chương trình nào. Viết chúng tốt là chủ đề của chương này.

31

<https://www.facebook.com/groups/phattrienlaptrinh>

32 **Chapter 3: Functions**  
32 Chương 3: Hàm

Consider the code in Listing 3-1. It’s hard to ﬁnd a long function in FitNesse,1 but after a bit of searching I came across this one. Not only is it long, but it’s got duplicated code, lots of odd strings, and many strange and inobvious data types and APIs. See how much sense you can make of it in the next three minutes.  
Hãy xem xét đoạn mã trong Liệt kê 3-1. Thật khó để tìm thấy một chức năng dài trong FitNesse,1 nhưng sau khi tìm kiếm một chút, tôi đã tìm thấy chức năng này. Nó không chỉ dài mà còn có mã trùng lặp, nhiều chuỗi lẻ và nhiều loại dữ liệu và API kỳ lạ và không rõ ràng. Xem bạn có thể hiểu được bao nhiêu ý nghĩa của nó trong ba phút tới.

**Listing 3-1**

**HtmlUtil.java (FitNesse 20070619)  
HtmlUtil.java (FitNesse 20070619)**

public static String testableHtml( PageData pageData,

boolean includeSuiteSetup ) throws Exception {

WikiPage wikiPage = pageData.getWikiPage(); StringBuffer buffer = new StringBuffer(); if (pageData.hasAttribute("Test")) {

if (includeSuiteSetup) { WikiPage suiteSetup =

PageCrawlerImpl.getInheritedPage( SuiteResponder.SUITE\_SETUP\_NAME, wikiPage

);

if (suiteSetup != null) { WikiPagePath pagePath =

suiteSetup.getPageCrawler().getFullPath(suiteSetup); String pagePathName = PathParser.render(pagePath); buffer.append("!include -setup .")

.append(pagePathName) .append("\n");

} }

WikiPage setup = PageCrawlerImpl.getInheritedPage("SetUp", wikiPage);

if (setup != null) { WikiPagePath setupPath =

wikiPage.getPageCrawler().getFullPath(setup); String setupPathName = PathParser.render(setupPath); buffer.append("!include -setup .")

.append(setupPathName) .append("\n");

} }

buffer.append(pageData.getContent()); if (pageData.hasAttribute("Test")) {

WikiPage teardown = PageCrawlerImpl.getInheritedPage("TearDown", wikiPage);

if (teardown != null) { WikiPagePath tearDownPath =

wikiPage.getPageCrawler().getFullPath(teardown);

String tearDownPathName = PathParser.render(tearDownPath); buffer.append("\n")

.append("!include -teardown .") .append(tearDownPathName) .append("\n");

}

1. An open-source testing tool. www.ﬁtnese.org  
1. Một công cụ kiểm tra mã nguồn mở. www.fitnese.org

<https://www.facebook.com/groups/phattrienlaptrinh>

**Functions** 33  
Chức năng 33

**Listing 3-1 (continued)**

**HtmlUtil.java (FitNesse 20070619)  
HtmlUtil.java (FitNesse 20070619)**

if (includeSuiteSetup) { WikiPage suiteTeardown =

PageCrawlerImpl.getInheritedPage( SuiteResponder.SUITE\_TEARDOWN\_NAME, wikiPage

);

if (suiteTeardown != null) { WikiPagePath pagePath =

suiteTeardown.getPageCrawler().getFullPath (suiteTeardown); String pagePathName = PathParser.render(pagePath); buffer.append("!include -teardown .")

.append(pagePathName) .append("\n");

} }

} pageData.setContent(buffer.toString()); return pageData.getHtml();

}

Do you understand the function after three minutes of study? Probably not. There’s too much going on in there at too many different levels of abstraction. There are strange strings and odd function calls mixed in with doubly nested ifstatements controlled by ﬂags.  
Bạn có hiểu chức năng sau ba phút nghiên cứu? Chắc là không. Có quá nhiều thứ đang diễn ra trong đó với quá nhiều mức độ trừu tượng khác nhau. Có các chuỗi kỳ lạ và các lệnh gọi hàm kỳ lạ được trộn lẫn với các câu lệnh if lồng nhau gấp đôi được kiểm soát bởi các ﬂag.

However, with just a few simple method extractions, some renaming, and a little restructuring, I was able to capture the intent of the function in the nine lines of Listing 3-2. See whether you can understand *that* in the next 3 minutes.  
Tuy nhiên, chỉ với một vài trích xuất phương thức đơn giản, một số đổi tên và một chút cấu trúc lại, tôi đã có thể nắm bắt được mục đích của hàm trong chín dòng của Liệt kê 3-2. Xem liệu bạn có thể hiểu điều đó trong 3 phút tới không.

**Listing 3-2**

**HtmlUtil.java (refactored)  
HtmlUtil.java (tái cấu trúc)**

public static String renderPageWithSetupsAndTeardowns( PageData pageData, boolean isSuite

) throws Exception {

boolean isTestPage = pageData.hasAttribute("Test"); if (isTestPage) {

WikiPage testPage = pageData.getWikiPage(); StringBuffer newPageContent = new StringBuffer(); includeSetupPages(testPage, newPageContent, isSuite); newPageContent.append(pageData.getContent()); includeTeardownPages(testPage, newPageContent, isSuite); pageData.setContent(newPageContent.toString());

}

return pageData.getHtml(); }

<https://www.facebook.com/groups/phattrienlaptrinh>

34 **Chapter 3: Functions**  
34 Chương 3: Hàm

Unless you are a student of FitNesse, you probably don’t understand all the details. Still, you probably understand that this function performs the inclusion of some setup and teardown pages into a test page and then renders that page into HTML. If you are familiar with JUnit,2 you probably realize that this function belongs to some kind of Web-based testing framework. And, of course, that is correct. Divining that information from Listing 3-2 is pretty easy, but it’s pretty well obscured by Listing 3-1.  
Trừ khi bạn là học viên của FitNesse, nếu không bạn có thể không hiểu hết các chi tiết. Tuy nhiên, bạn có thể hiểu rằng chức năng này thực hiện việc đưa một số trang thiết lập và chia nhỏ vào một trang thử nghiệm rồi kết xuất trang đó thành HTML. Nếu bạn đã quen thuộc với JUnit,2 thì bạn có thể nhận ra rằng chức năng này thuộc về một loại khung kiểm tra dựa trên Web nào đó. Và, tất nhiên, điều đó đúng. Việc dự đoán thông tin đó từ Liệt kê 3-2 khá dễ dàng, nhưng nó bị che khuất bởi Liệt kê 3-1.

So what is it that makes a function like Listing 3-2 easy to read and understand? How can we make a function communicate its intent? What attributes can we give our functions that will allow a casual reader to intuit the kind of program they live inside?  
Vậy điều gì làm cho một chức năng như Liệt kê 3-2 trở nên dễ đọc và dễ hiểu? Làm thế nào chúng ta có thể làm cho một chức năng truyền đạt ý định của nó? Những thuộc tính nào chúng ta có thể cung cấp cho các chức năng của mình để cho phép một người đọc bình thường trực giác được loại chương trình mà họ sống bên trong?

[**Small!**](#_page_91_0)  
Bé nhỏ!

The ﬁrst rule of functions is that they should be small. The second rule of functions is that *they should be smaller than that*. This is not an assertion that I can justify. I can’t provide any references to research that shows that very small functions are better. What I can tell you is that for nearly four decades I have written functions of all different sizes. I’ve writ-ten several nasty 3,000-line abominations. I’ve written scads of functions in the 100 to 300 line range. And I’ve written functions that were 20 to 30 lines long. What this experience has taught me, through long trial and error, is that functions should be very small.  
Nguyên tắc đầu tiên của hàm là chúng phải nhỏ. Quy tắc thứ hai của các chức năng là chúng phải nhỏ hơn thế. Đây không phải là một khẳng định mà tôi có thể biện minh. Tôi không thể cung cấp bất kỳ tài liệu tham khảo nào cho nghiên cứu cho thấy rằng các chức năng rất nhỏ sẽ tốt hơn. Điều tôi có thể nói với bạn là trong gần bốn thập kỷ qua, tôi đã viết các hàm ở mọi kích cỡ khác nhau. Tôi đã viết mười vài điều ghê tởm dài 3.000 dòng. Tôi đã viết rất nhiều hàm trong phạm vi 100 đến 300 dòng. Và tôi đã viết các hàm dài từ 20 đến 30 dòng. Trải nghiệm này đã dạy tôi điều gì, qua quá trình thử và sai lâu dài, là các chức năng nên rất nhỏ.

In the eighties we used to say that a function should be no bigger than a screen-full. Of course we said that at a time when VT100 screens were 24 lines by 80 columns, and our editors used 4 lines for administrative purposes. Nowadays with a cranked-down font and a nice big monitor, you can ﬁt 150 characters on a line and a 100 lines or more on a screen. Lines should not be 150 characters long. Functions should not be 100 lines long. Functions should hardly ever be 20 lines long.  
Vào những năm tám mươi, chúng tôi thường nói rằng một chức năng không được lớn hơn toàn màn hình. Tất nhiên, chúng tôi đã nói rằng vào thời điểm màn hình VT100 có 24 dòng x 80 cột và các biên tập viên của chúng tôi đã sử dụng 4 dòng cho mục đích quản trị. Ngày nay với một phông chữ nhỏ gọn và một màn hình lớn đẹp đẽ, bạn có thể viết 150 ký tự trên một dòng và 100 dòng hoặc hơn trên một màn hình. Các dòng không được dài 150 ký tự. Các chức năng không nên dài 100 dòng. Các chức năng hầu như không bao giờ dài 20 dòng.

How short should a function be? In 1999 I went to visit Kent Beck at his home in Ore-gon. We sat down and did some programming together. At one point he showed me a cute little Java/Swing program that he called *Sparkle*. It produced a visual effect on the screen very similar to the magic wand of the fairy godmother in the movie Cinderella. As you moved the mouse, the sparkles would drip from the cursor with a satisfying scintillation, falling to the bottom of the window through a simulated gravitational ﬁeld. When Kent showed me the code, I was struck by how small all the functions were. I was used to func-tions in Swing programs that took up miles of vertical space. Every function in *this* pro-gram was just two, or three, or four lines long. Each was transparently obvious. Each told a story.And each led you to the next in a compelling order. *That’s* how short your functions should be!3  
Một chức năng nên ngắn như thế nào? Năm 1999, tôi đến thăm Kent Beck tại nhà anh ở Ore-gon. Chúng tôi ngồi xuống và lập trình cùng nhau. Tại một thời điểm, anh ấy đã cho tôi xem một chương trình Java/Swing nhỏ dễ thương mà anh ấy gọi là Sparkle. Nó tạo ra hiệu ứng hình ảnh trên màn hình rất giống với cây đũa thần của bà tiên đỡ đầu trong phim Cô bé Lọ Lem. Khi bạn di chuyển chuột, các tia lấp lánh sẽ nhỏ giọt từ con trỏ với ánh sáng nhấp nháy thỏa mãn, rơi xuống cuối cửa sổ thông qua một trường hấp dẫn mô phỏng. Khi Kent cho tôi xem mã, tôi đã bị ấn tượng bởi tất cả các chức năng đều nhỏ như thế nào. Tôi đã quen với các chức năng trong các chương trình Swing chiếm hàng dặm không gian thẳng đứng. Mỗi chức năng trong chương trình này chỉ dài hai, ba hoặc bốn dòng. Mỗi cái đều rõ ràng minh bạch. Mỗi người kể một câu chuyện. Và mỗi người dẫn bạn đến câu chuyện tiếp theo theo thứ tự hấp dẫn. Đó là cách chức năng của bạn nên ngắn!3

2. An open-source unit-testing tool for Java. www.junit.org  
2. Một công cụ kiểm tra đơn vị mã nguồn mở dành cho Java. www.junit.org

3. I asked Kent whether he still had a copy, but he was unable to ﬁnd one. I searched all my old computers too, but to no avail. All that is left now is my memory of that program.  
3. Tôi đã hỏi Kent liệu anh ấy có còn một bản sao không, nhưng anh ấy không thể tìm thấy. Tôi cũng đã tìm kiếm tất cả các máy tính cũ của mình, nhưng vô ích. Tất cả những gì còn lại bây giờ là ký ức của tôi về chương trình đó.

<https://www.facebook.com/groups/phattrienlaptrinh>

**Do One Thing** 35  
Làm Một Việc 35

How short should your functions be? They should usually be shorter than Listing 3-2! Indeed, Listing 3-2 should really be shortened to Listing 3-3.  
Chức năng của bạn nên ngắn như thế nào? Chúng thường phải ngắn hơn Liệt kê 3-2! Thật vậy, Liệt kê 3-2 thực sự nên được rút gọn thành Liệt kê 3-3.

**Listing 3-3**

**HtmlUtil.java (re-refactored)  
HtmlUtil.java (tái cấu trúc lại)**

public static String renderPageWithSetupsAndTeardowns( PageData pageData, boolean isSuite) throws Exception { if (isTestPage(pageData))

includeSetupAndTeardownPages(pageData, isSuite); return pageData.getHtml();

}

[**Blocks and Indenting**](#_page_91_0)  
Khối và thụt lề

This implies that the blocks within ifstatements, elsestatements, whilestatements, and so on should be one line long. Probably that line should be a function call. Not only does this keep the enclosing function small, but it also adds documentary value because the function called within the block can have a nicely descriptive name.  
Điều này ngụ ý rằng các khối bên trong câu lệnh if, câu lệnh khác, câu lệnh trong khi, v.v. phải dài một dòng. Có lẽ dòng đó phải là một lệnh gọi hàm. Điều này không chỉ giữ cho chức năng kèm theo nhỏ mà còn tăng thêm giá trị tài liệu vì chức năng được gọi trong khối có thể có một tên mô tả độc đáo.

This also implies that functions should not be large enough to hold nested structures. Therefore, the indent level of a function should not be greater than one or two. This, of course, makes the functions easier to read and understand.  
Điều này cũng ngụ ý rằng các chức năng không nên đủ lớn để chứa các cấu trúc lồng nhau. Do đó, mức thụt lề của hàm không được lớn hơn một hoặc hai. Tất nhiên, điều này làm cho các chức năng dễ đọc và dễ hiểu hơn.

[**Do One Thing**](#_page_91_0)  
Làm một việc

It should be very clear that Listing 3-1 is doing lots more than one thing. It’s creating buffers, fetching pages, searching for inherited pages, rendering paths, appending arcane strings, and generating HTML, among other things. Listing 3-1 is very busy doing lots of different things. On the other hand, Listing 3-3 is doing one simple thing. It’s including setups and teardowns into test pages.  
Rõ ràng là Liệt kê 3-1 đang làm nhiều hơn một việc. Nó đang tạo bộ đệm, tìm nạp trang, tìm kiếm các trang kế thừa, hiển thị đường dẫn, nối thêm chuỗi phức tạp và tạo HTML, cùng nhiều thứ khác. Danh sách 3-1 rất bận làm nhiều việc khác nhau. Mặt khác, Liệt kê 3-3 đang làm một việc đơn giản. Nó bao gồm các thiết lập và chia nhỏ thành các trang thử nghiệm.

The following advice has appeared in one form or another for 30 years or more.  
Những lời khuyên sau đây đã xuất hiện dưới hình thức này hay hình thức khác từ 30 năm trở lên.

***FUNCTIONS SHOULD DO ONE THING. THEY SHOULD DO IT WELL. THEY SHOULD DO IT ONLY.  
CHỨC NĂNG NÊN LÀM MỘT ĐIỀU. HỌ NÊN LÀM TỐT. HỌ CHỈ NÊN LÀM ĐIỀU ĐÓ.***

The problem with this statement is that it is hard to know what “one thing” is. Does Listing 3-3 do one thing? It’s easy to make the case that it’s doing three things:  
Vấn đề với tuyên bố này là thật khó để biết “một điều” là gì. Liệt kê 3-3 có làm một việc không? Thật dễ dàng để hiểu rằng nó đang làm ba việc:

**1.** Determining whether the page is a test page. **2.** If so, including setups and teardowns.  
1. Xác định xem trang có phải là trang thử nghiệm hay không. 2. Nếu vậy, bao gồm cả các thiết lập và chia nhỏ.

**3.** Rendering the page in HTML.  
3. Kết xuất trang trong HTML.

<https://www.facebook.com/groups/phattrienlaptrinh>

36 **Chapter 3: Functions**  
36 Chương 3: Hàm

So which is it? Is the function doing one thing or three things? Notice that the three steps of the function are one level of abstraction below the stated name of the function. We can describe the function by describing it as a brief *TO*4 paragraph:  
Vậy nó là gì? Là chức năng làm một việc hay ba việc? Lưu ý rằng ba bước của hàm là một mức trừu tượng bên dưới tên hàm đã nêu. Chúng ta có thể mô tả chức năng bằng cách mô tả nó dưới dạng một đoạn TO4 ngắn gọn:

*TO RenderPageWithSetupsAndTeardowns, we check to see whether the page is a test page and if so, we include the setups and teardowns. In either case we render the page in HTML.  
ĐỐI VỚI RenderPageWithSetupsAndTeardowns, chúng tôi kiểm tra xem trang đó có phải là trang thử nghiệm hay không và nếu đúng như vậy, chúng tôi sẽ bao gồm các thiết lập và phân tích. Trong cả hai trường hợp, chúng tôi hiển thị trang bằng HTML.*

If a function does only those steps that are one level below the stated name of the function, then the function is doing one thing. After all, the reason we write functions is to decompose a larger concept (in other words, the name of the function) into a set of steps at the next level of abstraction.  
Nếu một chức năng chỉ thực hiện những bước thấp hơn một cấp so với tên đã nêu của chức năng, thì chức năng đó đang thực hiện một việc. Xét cho cùng, lý do chúng ta viết các hàm là để phân tách một khái niệm lớn hơn (nói cách khác, tên của hàm) thành một tập hợp các bước ở mức trừu tượng tiếp theo.

It should be very clear that Listing 3-1 contains steps at many different levels of abstraction. So it is clearly doing more than one thing. Even Listing 3-2 has two levels of abstraction, as proved by our ability to shrink it down. But it would be very hard to mean-ingfully shrink Listing 3-3. We could extract the if statement into a function named includeSetupsAndTeardownsIfTestPage, but that simply restates the code without changing the level of abstraction.  
Rõ ràng là Liệt kê 3-1 chứa các bước ở nhiều mức độ trừu tượng khác nhau. Vì vậy, nó rõ ràng đang làm nhiều hơn một điều. Ngay cả Liệt kê 3-2 cũng có hai mức độ trừu tượng, như đã được chứng minh bằng khả năng thu nhỏ nó của chúng ta. Nhưng sẽ rất khó để thu nhỏ Liệt kê 3-3 một cách có ý nghĩa. Chúng ta có thể trích xuất câu lệnh if thành một hàm có tên là includeSetupsAndTeardownsIfTestPage, nhưng điều đó chỉ đơn giản là trình bày lại mã mà không thay đổi mức độ trừu tượng.

So, another way to know that a function is doing more than “one thing” is if you can extract another function from it with a name that is not merely a restatement of its imple-mentation [G34].  
Vì vậy, một cách khác để biết rằng một hàm đang làm nhiều hơn “một việc” là nếu bạn có thể trích xuất một hàm khác từ hàm đó với một tên không chỉ đơn thuần là sự trình bày lại quá trình triển khai của nó [G34].

[**Sections within Functions**](#_page_91_0)  
Các phần trong Chức năng

Look at Listing 4-7 on page 71. Notice that the generatePrimesfunction is divided into sections such as *declarations*, *initializations*, and *sieve*. This is an obvious symptom of doing more than one thing. Functions that do one thing cannot be reasonably divided into sections.  
Xem Liệt kê 4-7 trên trang 71. Lưu ý rằng hàm generatePrimes được chia thành các phần như khai báo, khởi tạo và sàng. Đây là một triệu chứng rõ ràng của việc làm nhiều hơn một việc. Các chức năng làm một việc không thể được chia thành các phần một cách hợp lý.

[**One Level of Abstraction per Function**](#_page_91_0)  
Một mức độ trừu tượng cho mỗi chức năng

In order to make sure our functions are doing “one thing,” we need to make sure that the statements within our function are all at the same level of abstraction. It is easy to see how Listing 3-1 violates this rule. There are concepts in there that are at a very high level of abstraction, such as getHtml(); others that are at an intermediate level of abstraction, such as: String pagePathName = PathParser.render(pagePath); and still others that are remark-ably low level, such as: .append("\n").  
Để đảm bảo các hàm của chúng ta đang thực hiện “một việc”, chúng ta cần đảm bảo rằng các câu lệnh trong hàm của chúng ta đều ở cùng một mức độ trừu tượng. Dễ dàng thấy Liệt kê 3-1 vi phạm quy tắc này như thế nào. Có những khái niệm ở mức độ trừu tượng rất cao, chẳng hạn như getHtml(); những thứ khác ở mức độ trừu tượng trung bình, chẳng hạn như: String pagePathName = PathParser.render(pagePath); và vẫn còn những thứ khác ở mức thấp đáng kể, chẳng hạn như: .append("\n").

Mixing levels of abstraction within a function is always confusing. Readers may not be able to tell whether a particular expression is an essential concept or a detail. Worse,  
Trộn các mức độ trừu tượng trong một chức năng luôn gây nhầm lẫn. Người đọc có thể không biết liệu một biểu thức cụ thể là một khái niệm thiết yếu hay một chi tiết. Tệ hơn,

4. The LOGO language used the keyword “TO” in the same way that Ruby and Python use “def.” So every function began with the word “TO.” This had an interesting effect on the way functions were designed.  
4. Ngôn ngữ LOGO đã sử dụng từ khóa “TO” giống như cách mà Ruby và Python sử dụng “def”. Vì vậy, mọi chức năng đều bắt đầu bằng từ “TO”. Điều này có một tác động thú vị đến cách các chức năng được thiết kế.

<https://www.facebook.com/groups/phattrienlaptrinh>

**Switch Statements** 37  
Câu lệnh Switch 37

like broken windows, once details are mixed with essential concepts, more and more details tend to accrete within the function.  
giống như các cửa sổ bị vỡ, một khi các chi tiết được trộn lẫn với các khái niệm thiết yếu, ngày càng nhiều chi tiết có xu hướng tích tụ trong chức năng.

[**Reading Code from Top to Bottom: *The Stepdown Rule***](#_page_91_0)  
Đọc mã từ trên xuống dưới: Quy tắc bước xuống

We want the code to read like a top-down narrative.5 We want every function to be fol-lowed by those at the next level of abstraction so that we can read the program, descending one level of abstraction at a time as we read down the list of functions. I call this *The Step-down Rule*.  
Chúng tôi muốn mã đọc giống như một câu chuyện kể từ trên xuống.5 Chúng tôi muốn mọi chức năng được theo sau bởi những chức năng ở cấp độ trừu tượng tiếp theo để chúng tôi có thể đọc chương trình, giảm dần một cấp độ trừu tượng tại một thời điểm khi chúng tôi đọc xuống danh sách các chức năng. Tôi gọi đây là Quy tắc bước xuống.

To say this differently, we want to be able to read the program as though it were a set of *TO* paragraphs, each of which is describing the current level of abstraction and refer-encing subsequent *TO* paragraphs at the next level down.  
Nói cách khác, chúng tôi muốn có thể đọc chương trình như thể nó là một tập hợp các đoạn TO, mỗi đoạn mô tả mức độ trừu tượng hiện tại và tham chiếu các đoạn TO tiếp theo ở cấp độ tiếp theo.

*To include the setups and teardowns, we include setups, then we include the test page con-tent, and then we include the teardowns.  
Để bao gồm các thiết lập và phân tích, chúng tôi bao gồm các thiết lập, sau đó chúng tôi bao gồm nội dung trang thử nghiệm, sau đó chúng tôi bao gồm các phân tích.*

*To include the setups, we include the suite setup if this is a suite, then we include the regular setup.  
Để bao gồm các thiết lập, chúng tôi bao gồm thiết lập bộ nếu đây là một bộ, sau đó chúng tôi bao gồm thiết lập thông thường.*

*To include the suite setup, we search the parent hierarchy for the “SuiteSetUp” page and add an include statement with the path of that page.  
Để bao gồm thiết lập bộ, chúng tôi tìm kiếm cấu trúc phân cấp gốc cho trang “SuiteSetUp” và thêm một câu lệnh bao gồm với đường dẫn của trang đó.*

*To search the parent. . .  
Để tìm kiếm cha mẹ. . .*

It turns out to be very difﬁcult for programmers to learn to follow this rule and write functions that stay at a single level of abstraction. But learning this trick is also very important. It is the key to keeping functions short and making sure they do “one thing.” Making the code read like a top-down set of *TO* paragraphs is an effective technique for keeping the abstraction level consistent.  
Hóa ra là rất khó để các lập trình viên học cách tuân theo quy tắc này và viết các hàm ở một mức trừu tượng duy nhất. Nhưng học thủ thuật này cũng rất quan trọng. Đó là chìa khóa để giữ cho các chức năng ngắn gọn và đảm bảo rằng chúng thực hiện “một việc”. Làm cho mã được đọc giống như một tập hợp các đoạn TO từ trên xuống là một kỹ thuật hiệu quả để giữ cho mức độ trừu tượng nhất quán.

Take a look at Listing 3-7 at the end of this chapter. It shows the whole testableHtml function refactored according to the principles described here. Notice how each function introduces the next, and each function remains at a consistent level of abstraction.  
Hãy xem Liệt kê 3-7 ở cuối chương này. Nó hiển thị toàn bộ hàm testableHtml được cấu trúc lại theo các nguyên tắc được mô tả ở đây. Lưu ý cách mỗi chức năng giới thiệu chức năng tiếp theo và mỗi chức năng vẫn ở mức trừu tượng nhất quán.

[**Switch Statements**](#_page_91_0)  
Báo cáo chuyển đổi

It’s hard to make a small switchstatement.6 Even a switchstatement with only two cases is larger than I’d like a single block or function to be. It’s also hard to make a switchstate-ment that does one thing. By their nature, switchstatements always do *N* things. Unfortu-nately we can’t always avoid switchstatements, but we *can* make sure that each switch statement is buried in a low-level class and is never repeated. We do this, of course, with polymorphism.  
Thật khó để tạo một câu lệnh chuyển mạch nhỏ.6 Ngay cả một câu lệnh chuyển đổi chỉ có hai trường hợp cũng lớn hơn tôi muốn là một khối hoặc chức năng đơn lẻ. Cũng khó để tạo ra một trạng thái chuyển mạch làm được một việc. Về bản chất, các câu lệnh chuyển mạch luôn thực hiện N việc. Thật không may, không phải lúc nào chúng ta cũng có thể tránh được các câu lệnh chuyển đổi, nhưng chúng ta có thể đảm bảo rằng mỗi câu lệnh chuyển đổi được chôn trong một lớp cấp thấp và không bao giờ lặp lại. Tất nhiên, chúng tôi làm điều này với tính đa hình.

5. [KP78], p. 37.

6. And, of course, I include if/else chains in this.  
6. Và, tất nhiên, tôi bao gồm các chuỗi if/else trong phần này.

<https://www.facebook.com/groups/phattrienlaptrinh>

38 **Chapter 3: Functions**  
38 Chương 3: Hàm

Consider Listing 3-4. It shows just one of the operations that might depend on the type of employee.  
Xem xét Liệt kê 3-4. Nó chỉ hiển thị một trong các hoạt động có thể phụ thuộc vào loại nhân viên.

**Listing 3-4 Payroll.java  
Liệt kê 3-4 Payroll.java**

public Money calculatePay(Employee e) throws InvalidEmployeeType {

switch (e.type) { case COMMISSIONED:

return calculateCommissionedPay(e); case HOURLY:

return calculateHourlyPay(e); case SALARIED:

return calculateSalariedPay(e); default:

throw new InvalidEmployeeType(e.type); }

}

There are several problems with this function. First, it’s large, and when new employee types are added, it will grow. Second, it very clearly does more than one thing. Third, it violates the Single Responsibility Principle7 (SRP) because there is more than one reason for it to change. Fourth, it violates the Open Closed Principle8 (OCP) because it must change whenever new types are added. But possibly the worst problem with this function is that there are an unlimited number of other functions that will have the same structure. For example we could have  
Có một số vấn đề với chức năng này. Đầu tiên, nó lớn và khi các loại nhân viên mới được thêm vào, nó sẽ phát triển. Thứ hai, rõ ràng nó làm được nhiều hơn một việc. Thứ ba, nó vi phạm Nguyên tắc Trách nhiệm duy nhất7 (SRP) vì có nhiều hơn một lý do để nó thay đổi. Thứ tư, nó vi phạm Nguyên tắc Mở Đóng8 (OCP) vì nó phải thay đổi bất cứ khi nào các loại mới được thêm vào. Nhưng có lẽ vấn đề tồi tệ nhất với chức năng này là có vô số chức năng khác sẽ có cùng cấu trúc. Ví dụ chúng ta có thể có

isPayday(Employee e, Date date), or

deliverPay(Employee e, Money pay),

or a host of others. All of which would have the same deleterious structure.  
hoặc một loạt những người khác. Tất cả trong số đó sẽ có cùng một cấu trúc có hại.

The solution to this problem (see Listing 3-5) is to bury the switchstatement in the basement of an ABSTRACT FACTORY,9 and never let anyone see it. The factory will use the switchstatement to create appropriate instances of the derivatives of Employee, and the var-ious functions, such as calculatePay, isPayday, and deliverPay, will be dispatched poly-morphically through the Employee interface.  
Giải pháp cho vấn đề này (xem Liệt kê 3-5) là chôn công tắc trong tầng hầm của NHÀ MÁY TÓM TẮT,9 và không bao giờ để bất kỳ ai nhìn thấy nó. Nhà máy sẽ sử dụng câu lệnh chuyển mạch để tạo các phiên bản thích hợp của các dẫn xuất của Nhân viên và các hàm khác nhau, chẳng hạn như tính toánPay, isPayday và deliverPay, sẽ được gửi đa dạng thông qua giao diện Nhân viên.

My general rule for switchstatements is that they can be tolerated if they appear only once, are used to create polymorphic objects, and are hidden behind an inheritance  
Nguyên tắc chung của tôi đối với các câu lệnh chuyển đổi là chúng có thể được chấp nhận nếu chúng chỉ xuất hiện một lần, được sử dụng để tạo các đối tượng đa hình và được ẩn đằng sau một kế thừa

7. a. <http://en.wikipedia.org/wiki/Single_responsibility_principle> b. <http://www.objectmentor.com/resources/articles/srp.pdf>

8. a. <http://en.wikipedia.org/wiki/Open/closed_principle>

b. <http://www.objectmentor.com/resources/articles/ocp.pdf> 9. [GOF].

<https://www.facebook.com/groups/phattrienlaptrinh>

**Use Descriptive Names** 39  
Sử dụng tên mô tả 39

**Listing 3-5**

**Employee and Factory  
Nhân viên và Nhà máy**

public abstract class Employee { public abstract boolean isPayday(); public abstract Money calculatePay();

public abstract void deliverPay(Money pay); }

-----------------

public interface EmployeeFactory {

public Employee makeEmployee(EmployeeRecord r) throws InvalidEmployeeType; }

-----------------

public class EmployeeFactoryImpl implements EmployeeFactory {

public Employee makeEmployee(EmployeeRecord r) throws InvalidEmployeeType { switch (r.type) {

case COMMISSIONED:

return new CommissionedEmployee(r) ; case HOURLY:

return new HourlyEmployee(r); case SALARIED:

return new SalariedEmploye(r); default:

throw new InvalidEmployeeType(r.type); }

} }

relationship so that the rest of the system can’t see them [G23]. Of course every circum-stance is unique, and there are times when I violate one or more parts of that rule.  
mối quan hệ để phần còn lại của hệ thống không thể nhìn thấy chúng [G23]. Tất nhiên, mỗi trường hợp là duy nhất và có những lúc tôi vi phạm một hoặc nhiều phần của quy tắc đó.

[**Use Descriptive Names**](#_page_91_0)  
Sử dụng tên mô tả

In Listing 3-7 I changed the name of our example function from testableHtml to SetupTeardownIncluder.render. This is a far better name because it better describes what the function does. I also gave each of the private methods an equally descriptive name such as isTestableor includeSetupAndTeardownPages. It is hard to overestimate the value of good names. Remember Ward’s principle: “*You know you are working on clean code when each routine turns out to be pretty much what you expected.*” Half the battle to achieving that principle is choosing good names for small functions that do one thing. The smaller and more focused a function is, the easier it is to choose a descriptive name.  
Trong Liệt kê 3-7, tôi đã đổi tên hàm ví dụ của chúng ta từ testableHtml thành SetupTeardownIncluder.render. Đây là một cái tên hay hơn nhiều vì nó mô tả tốt hơn những gì chức năng làm. Tôi cũng đã đặt cho mỗi phương thức riêng một tên mô tả giống nhau, chẳng hạn như isTestable hoặc bao gồmSetupAndTeardownPages. Thật khó để đánh giá quá cao giá trị của những cái tên hay. Hãy nhớ nguyên tắc của Ward: “Bạn biết bạn đang làm việc với mã sạch khi mỗi quy trình trở nên giống như những gì bạn mong đợi.” Một nửa trận chiến để đạt được nguyên tắc đó là chọn tên hay cho các chức năng nhỏ thực hiện một việc. Chức năng càng nhỏ và càng tập trung thì càng dễ chọn tên mô tả.

Don’t be afraid to make a name long. A long descriptive name is better than a short enigmatic name. A long descriptive name is better than a long descriptive comment. Use a naming convention that allows multiple words to be easily read in the function names, and then make use of those multiple words to give the function a name that says what it does.  
Đừng ngại đặt tên dài. Một cái tên mô tả dài sẽ tốt hơn một cái tên ngắn bí ẩn. Một tên mô tả dài sẽ tốt hơn một nhận xét mô tả dài. Sử dụng quy ước đặt tên cho phép dễ dàng đọc nhiều từ trong tên hàm, sau đó sử dụng nhiều từ đó để đặt cho hàm một cái tên nói lên chức năng của nó.

<https://www.facebook.com/groups/phattrienlaptrinh>

40 **Chapter 3: Functions**  
40 Chương 3: Hàm

Don’t be afraid to spend time choosing a name. Indeed, you should try several differ-ent names and read the code with each in place. Modern IDEs like Eclipse or IntelliJ make it trivial to change names. Use one of those IDEs and experiment with different names until you ﬁnd one that is as descriptive as you can make it.  
Đừng ngại tốn thời gian chọn tên. Thật vậy, bạn nên thử một số tên khác nhau và đọc mã với từng vị trí. Các IDE hiện đại như Eclipse hoặc IntelliJ khiến việc thay đổi tên trở nên đơn giản. Sử dụng một trong những IDE đó và thử nghiệm với các tên khác nhau cho đến khi bạn tìm thấy một tên mô tả phù hợp nhất có thể.

Choosing descriptive names will clarify the design of the module in your mind and help you to improve it. It is not at all uncommon that hunting for a good name results in a favorable restructuring of the code.  
Chọn tên mô tả sẽ làm rõ thiết kế của mô-đun trong tâm trí bạn và giúp bạn cải thiện nó. Không có gì lạ khi việc săn lùng một cái tên hay dẫn đến việc tái cấu trúc mã một cách thuận lợi.

Be consistent in your names. Use the same phrases, nouns, and verbs in the function names you choose for your modules. Consider, for example, the names includeSetup-AndTeardownPages, includeSetupPages, includeSuiteSetupPage, and includeSetupPage. The similar phraseology in those names allows the sequence to tell a story. Indeed, if I showed you just the sequence above, you’d ask yourself: “What happened to includeTeardownPages, includeSuiteTeardownPage, and includeTeardownPage?” How’s that for being “. . . *pretty much what you expected*.”  
Hãy nhất quán trong tên của bạn. Sử dụng cùng một cụm từ, danh từ và động từ trong tên hàm mà bạn chọn cho các mô-đun của mình. Ví dụ, hãy xem xét các tên bao gồmSetup-AndTeardownPages, bao gồmSetupPages, bao gồmSuiteSetupPage và bao gồmSetupPage. Cụm từ tương tự trong những cái tên đó cho phép trình tự kể một câu chuyện. Thật vậy, nếu tôi chỉ cho bạn trình tự ở trên, bạn sẽ tự hỏi: “Điều gì đã xảy ra với includeTeardownPages, includeSuiteTeardownPage, và includeTeardownPage?” Làm thế nào mà cho là “. . . khá nhiều những gì bạn mong đợi.

[**Function Arguments**](#_page_91_0)  
Đối số chức năng

The ideal number of arguments for a function is zero (niladic). Next comes one (monadic), followed closely by two (dyadic). Three arguments (triadic) should be avoided where possible. More than three (polyadic) requires very special justiﬁcation—and then shouldn’t be used anyway.  
Số đối số lý tưởng cho một hàm là số không (niladic). Tiếp theo là một (đơn nguyên), theo sát là hai (dyadic). Ba đối số (bộ ba) nên tránh nếu có thể. Nhiều hơn ba (polyadic) đòi hỏi sự chứng minh rất đặc biệt—và sau đó không nên được sử dụng.

Arguments are hard. They take a lot of con-ceptual power. That’s why I got rid of almost all of them from the example. Consider, for instance, the StringBuffer in the example. We could have passed it around as an argument rather than mak-ing it an instance variable, but then our readers would have had to interpret it each time they saw it. When you are reading the story told by the  
Lập luận là khó. Họ mất rất nhiều sức mạnh khái niệm. Đó là lý do tại sao tôi đã loại bỏ gần như tất cả chúng khỏi ví dụ. Ví dụ, hãy xem xét StringBuffer trong ví dụ. Chúng tôi có thể chuyển nó thành một đối số hơn là biến nó thành một biến thể hiện, nhưng sau đó độc giả của chúng tôi sẽ phải diễn giải nó mỗi khi họ nhìn thấy nó. Khi bạn đang đọc câu chuyện được kể bởi

module, includeSetupPage()is easier to understand than includeSetupPageInto(newPage-Content). The argument is at a different level of abstraction than the function name and forces you to know a detail (in other words, StringBuffer) that isn’t particularly important at that point.  
mô-đun, bao gồmSetupPage() dễ hiểu hơn bao gồmSetupPageInto(newPage-Content). Đối số ở một mức độ trừu tượng khác với tên hàm và buộc bạn phải biết một chi tiết (nói cách khác, StringBuffer) không đặc biệt quan trọng tại thời điểm đó.

Arguments are even harder from a testing point of view. Imagine the difﬁculty of writing all the test cases to ensure that all the various combinations of arguments work properly. If there are no arguments, this is trivial. If there’s one argument, it’s not too hard. With two arguments the problem gets a bit more challenging. With more than two argu-ments, testing every combination of appropriate values can be daunting.  
Đối số thậm chí còn khó hơn từ quan điểm thử nghiệm. Hãy tưởng tượng khó khăn khi viết tất cả các trường hợp thử nghiệm để đảm bảo rằng tất cả các tổ hợp đối số khác nhau hoạt động đúng. Nếu không có đối số, điều này là tầm thường. Nếu có một lý lẽ, nó không quá khó. Với hai đối số, vấn đề trở nên khó khăn hơn một chút. Với nhiều hơn hai đối số, việc kiểm tra mọi kết hợp của các giá trị thích hợp có thể gây khó khăn.

<https://www.facebook.com/groups/phattrienlaptrinh>

**Function Arguments** 41  
Các đối số hàm 41

Output arguments are harder to understand than input arguments. When we read a function, we are used to the idea of information going *in* to the function through arguments and *out* through the return value. We don’t usually expect information to be going out through the arguments. So output arguments often cause us to do a double-take.  
Đối số đầu ra khó hiểu hơn đối số đầu vào. Khi chúng ta đọc một hàm, chúng ta đã quen với ý tưởng thông tin đi vào hàm thông qua các đối số và thông tin đi ra thông qua giá trị trả về. Chúng tôi thường không mong đợi thông tin sẽ được đưa ra thông qua các đối số. Vì vậy, các đối số đầu ra thường khiến chúng ta phải thực hiện hai lần.

One input argument is the next best thing to no arguments. SetupTeardown-Includer.render(pageData)is pretty easy to understand. Clearly we are going to *render* the data in the pageData object.  
Một đối số đầu vào là điều tốt nhất tiếp theo không có đối số. SetupTeardown-Includer.render(pageData) khá dễ hiểu. Rõ ràng là chúng ta sẽ hiển thị dữ liệu trong đối tượng pageData.

[**Common Monadic Forms**](#_page_91_0)  
Các hình thức đơn nguyên phổ biến

There are two very common reasons to pass a single argument into a function. You may be asking a question about that argument, as in booleanfileExists(“MyFile”). Or you may be operating on that argument, transforming it into something else and *returning it*. For example, InputStream fileOpen(“MyFile”) transforms a ﬁle name String into an InputStream return value. These two uses are what readers expect when they see a func-tion. You should choose names that make the distinction clear, and always use the two forms in a consistent context. (See Command Query Separation below.)  
Có hai lý do rất phổ biến để chuyển một đối số vào một hàm. Bạn có thể đặt câu hỏi về lập luận đó, chẳng hạn như trong booleanfileExists(“MyFile”). Hoặc bạn có thể đang vận hành đối số đó, chuyển đổi nó thành một thứ khác và trả lại nó. Ví dụ: InputStream fileOpen(“MyFile”) biến đổi tên tệp Chuỗi thành giá trị trả về của InputStream. Hai cách sử dụng này là những gì người đọc mong đợi khi họ nhìn thấy một chức năng. Bạn nên chọn các tên phân biệt rõ ràng và luôn sử dụng hai biểu mẫu trong một ngữ cảnh nhất quán. (Xem Phân tách truy vấn lệnh bên dưới.)

A somewhat less common, but still very useful form for a single argument function, is an *event*. In this form there is an input argument but no output argument. The overall program is meant to interpret the function call as an event and use the argument to alter the state of the system, for example, void passwordAttemptFailedNtimes(int attempts). Use this form with care. It should be very clear to the reader that this is an event. Choose names and contexts carefully.  
Một dạng hơi ít phổ biến hơn, nhưng vẫn rất hữu ích cho một hàm đối số, là một sự kiện. Ở dạng này có đối số đầu vào nhưng không có đối số đầu ra. Toàn bộ chương trình nhằm diễn giải lời gọi hàm như một sự kiện và sử dụng đối số để thay đổi trạng thái của hệ thống, ví dụ: void passwordAttemptFailedNtimes(int nỗ lực). Sử dụng hình thức này một cách cẩn thận. Người đọc phải rất rõ ràng rằng đây là một sự kiện. Chọn tên và bối cảnh cẩn thận.

Try to avoid any monadic functions that don’t follow these forms, for example, void includeSetupPageInto(StringBuffer pageText). Using an output argument instead of a return value for a transformation is confusing. If a function is going to transform its input argument, the transformation should appear as the return value. Indeed, StringBuffer transform(StringBuffer in)is better than void transform-(StringBuffer out), even if the implementation in the ﬁrst case simply returns the input argument. At least it still follows the form of a transformation.  
Cố gắng tránh bất kỳ chức năng đơn nguyên nào không tuân theo các biểu mẫu này, chẳng hạn như void includeSetupPageInto(StringBuffer pageText). Việc sử dụng đối số đầu ra thay vì giá trị trả về cho phép biến đổi là điều khó hiểu. Nếu một hàm sẽ chuyển đổi đối số đầu vào của nó, thì phép chuyển đổi đó sẽ xuất hiện dưới dạng giá trị trả về. Thật vậy, biến đổi StringBuffer(StringBuffer in) tốt hơn biến đổi void-(StringBuffer out), ngay cả khi việc triển khai trong trường hợp đầu tiên chỉ đơn giản trả về đối số đầu vào. Ít ra thì nó vẫn theo dạng biến hình.

[**Flag Arguments**](#_page_91_0)  
Đối số cờ

Flag arguments are ugly. Passing a boolean into a function is a truly terrible practice. It immediately complicates the signature of the method, loudly proclaiming that this function does more than one thing. It does one thing if the ﬂag is true and another if the ﬂag is false!  
Đối số cờ là xấu xí. Chuyển một boolean vào một hàm là một cách làm thực sự tồi tệ. Nó ngay lập tức làm phức tạp chữ ký của phương thức, lớn tiếng tuyên bố rằng chức năng này thực hiện nhiều hơn một việc. Nó thực hiện một việc nếu cờ là đúng và một việc khác nếu cờ là sai!

In Listing 3-7 we had no choice because the callers were already passing that ﬂag in, and I wanted to limit the scope of refactoring to the function and below. Still, the method call render(true)is just plain confusing to a poor reader. Mousing over the call and seeing render(boolean isSuite)helps a little, but not that much. We should have split the function into two: renderForSuite() and renderForSingleTest().  
Trong Liệt kê 3-7, chúng ta không có lựa chọn nào vì những người gọi đã chuyển cờ đó vào, và tôi muốn giới hạn phạm vi tái cấu trúc đối với hàm và bên dưới. Tuy nhiên, lệnh gọi phương thức render(true) chỉ đơn giản là gây nhầm lẫn cho người đọc kém. Di chuột qua cuộc gọi và xem kết xuất (boolean isSuite) sẽ giúp một chút, nhưng không nhiều lắm. Lẽ ra chúng ta nên chia chức năng thành hai: renderForSuite() và renderForSingleTest().

<https://www.facebook.com/groups/phattrienlaptrinh>

42 **Chapter 3: Functions**  
42 Chương 3: Hàm

[**Dyadic Functions**](#_page_91_0)  
Chức năng Dyadic

A function with two arguments is harder to understand than a monadic function. For exam-ple, writeField(name) is easier to understand than writeField(output-Stream, name).10 Though the meaning of both is clear, the ﬁrst glides past the eye, easily depositing its meaning. The second requires a short pause until we learn to ignore the ﬁrst parameter. And *that*, of course, eventually results in problems because we should never ignore any part of code. The parts we ignore are where the bugs will hide.  
Hàm có hai đối số khó hiểu hơn hàm đơn nguyên. Ví dụ, writeField(name) dễ hiểu hơn writeField(output-Stream, name).10 Mặc dù ý nghĩa của cả hai đều rõ ràng, nhưng ý nghĩa đầu tiên lướt qua mắt, dễ dàng ghi lại ý nghĩa của nó. Thứ hai yêu cầu tạm dừng một thời gian ngắn cho đến khi chúng ta học cách bỏ qua tham số đầu tiên. Và điều đó, tất nhiên, cuối cùng sẽ dẫn đến các vấn đề bởi vì chúng ta không bao giờ nên bỏ qua bất kỳ phần nào của mã. Những phần chúng tôi bỏ qua là nơi mà các lỗi sẽ ẩn.

There are times, of course, where two arguments are appropriate. For example, Point p = new Point(0,0); is perfectly reasonable. Cartesian points naturally take two arguments. Indeed, we’d be very surprised to see new Point(0). However, the two argu-ments in this case *are ordered components of a single value!* Whereas output-Streamand name have neither a natural cohesion, nor a natural ordering.  
Tất nhiên, có những lúc hai lập luận là phù hợp. Ví dụ, Điểm p = Điểm mới(0,0); là hoàn toàn hợp lý. Điểm Descartes tự nhiên có hai đối số. Thật vậy, chúng tôi sẽ rất ngạc nhiên khi thấy Điểm(0) mới. Tuy nhiên, hai đối số trong trường hợp này là các thành phần có thứ tự của một giá trị duy nhất! Trong khi tên luồng đầu ra và tên không có sự gắn kết tự nhiên cũng như thứ tự tự nhiên.

Even obvious dyadic functions like assertEquals(expected, actual)are problematic. How many times have you put the actualwhere the expectedshould be? The two argu-ments have no natural ordering. The expected,actual ordering is a convention that requires practice to learn.  
Ngay cả các chức năng cặp đôi rõ ràng như assertEquals(expected, reality) cũng có vấn đề. Đã bao nhiêu lần bạn đặt cái thực tế vào chỗ cái lẽ ra phải ở? Hai đối số không có thứ tự tự nhiên. Thứ tự dự kiến, thực tế là một quy ước đòi hỏi phải thực hành để tìm hiểu.

Dyads aren’t evil, and you will certainly have to write them. However, you should be aware that they come at a cost and should take advantage of what mechanims may be available to you to convert them into monads. For example, you might make the writeField method a member of outputStream so that you can say outputStream. writeField(name). Or you might make the outputStreama member variable of the current class so that you don’t have to pass it. Or you might extract a new class like FieldWriter that takes the outputStream in its constructor and has a write method.  
Dyads không xấu xa, và bạn chắc chắn sẽ phải viết chúng. Tuy nhiên, bạn nên lưu ý rằng chúng phải trả giá và nên tận dụng những cơ chế nào có thể có sẵn cho bạn để chuyển đổi chúng thành các đơn nguyên. Ví dụ: bạn có thể đặt phương thức writeField thành thành viên của outputStream để bạn có thể nói outputStream. writeField(tên). Hoặc bạn có thể tạo biến thành viên outputStreama của lớp hiện tại để bạn không phải chuyển nó. Hoặc bạn có thể trích xuất một lớp mới như FieldWriter lấy outputStream trong hàm tạo của nó và có một phương thức ghi.

[**Triads**](#_page_91_0)  
Hội Tam Hoàng

Functions that take three arguments are signiﬁcantly harder to understand than dyads. The issues of ordering, pausing, and ignoring are more than doubled. I suggest you think very carefully before creating a triad.  
Các hàm nhận ba đối số khó hiểu hơn đáng kể so với các hàm đôi. Các vấn đề về đặt hàng, tạm dừng và bỏ qua tăng hơn gấp đôi. Tôi đề nghị bạn suy nghĩ thật cẩn thận trước khi tạo một bộ ba.

For example, consider the common overload of assertEqualsthat takes three argu-ments: assertEquals(message, expected, actual). How many times have you read the messageand thought it was the expected? I have stumbled and paused over that particular triad many times. In fact, *every time I see it,* I do a double-take and then learn to ignore the message.  
Ví dụ, xem xét tình trạng quá tải phổ biến của assertEquals có ba đối số: assertEquals(thông báo, dự kiến, thực tế). Đã bao nhiêu lần bạn đọc tin nhắn và nghĩ rằng đó là điều mong đợi? Tôi đã nhiều lần vấp ngã và dừng lại ở bộ ba cụ thể đó. Trên thực tế, mỗi lần tôi nhìn thấy nó, tôi đều thực hiện hai lần và sau đó học cách bỏ qua tin nhắn.

On the other hand, here is a triad that is not quite so insidious: assertEquals(1.0, amount, .001). Although this still requires a double-take, it’s one that’s worth taking. It’s always good to be reminded that equality of ﬂoating point values is a relative thing.  
Mặt khác, đây là một bộ ba không quá quỷ quyệt: assertEquals(1.0,mount,.001). Mặc dù điều này vẫn yêu cầu thực hiện hai lần, nhưng đây là một điều đáng để thực hiện. Thật tốt khi được nhắc nhở rằng sự bằng nhau của các giá trị dấu phẩy động là một điều tương đối.

10. I just ﬁnished refactoring a module that used the dyadic form. I was able to make the outputStream a ﬁeld of the class and convert all the writeField calls to the monadic form. The result was much cleaner.  
10. Tôi vừa hoàn thành việc tái cấu trúc một mô-đun sử dụng dạng cặp đôi. Tôi đã có thể biến outputStream thành một trường của lớp và chuyển đổi tất cả các lệnh gọi writeField sang dạng đơn nguyên. Kết quả là sạch hơn nhiều.

<https://www.facebook.com/groups/phattrienlaptrinh>

**Function Arguments** 43  
Các đối số hàm 43

[**Argument Objects**](#_page_91_0)  
đối số đối tượng

When a function seems to need more than two or three arguments, it is likely that some of those arguments ought to be wrapped into a class of their own. Consider, for example, the difference between the two following declarations:  
Khi một hàm dường như cần nhiều hơn hai hoặc ba đối số, có khả năng một số đối số đó phải được gói gọn trong một lớp của riêng chúng. Ví dụ, xem xét sự khác biệt giữa hai khai báo sau:

Circle makeCircle(double x, double y, double radius); Circle makeCircle(Point center, double radius);

Reducing the number of arguments by creating objects out of them may seem like cheating, but it’s not. When groups of variables are passed together, the way xand yare in the example above, they are likely part of a concept that deserves a name of its own.  
Giảm số lượng đối số bằng cách tạo đối tượng từ chúng có vẻ giống như gian lận, nhưng thực tế không phải vậy. Khi các nhóm biến được truyền cùng nhau, theo cách xand yare trong ví dụ trên, chúng có thể là một phần của khái niệm xứng đáng được đặt tên riêng.

[**Argument Lists**](#_page_91_0)  
Danh sách đối số

Sometimes we want to pass a variable number of arguments into a function. Consider, for example, the String.format method:  
Đôi khi chúng ta muốn chuyển một số lượng đối số khác nhau vào một hàm. Ví dụ, hãy xem xét phương thức String.format:

String.format("%s worked %.2f hours.", name, hours);

If the variable arguments are all treated identically, as they are in the example above, then they are equivalent to a single argument of type List. By that reasoning, String.format is actually dyadic. Indeed, the declaration of String.format as shown below is clearly dyadic.  
Nếu tất cả các đối số biến đều được xử lý giống hệt nhau, như trong ví dụ trên, thì chúng tương đương với một đối số kiểu Danh sách. Theo lý do đó, String.format thực sự là một cặp. Thật vậy, việc khai báo String.format như hình dưới đây rõ ràng là theo cặp.

public String format(String format, Object... args)  
định dạng Chuỗi công khai (Định dạng chuỗi, Đối tượng... args)

So all the same rules apply. Functions that take variable arguments can be monads, dyads, or even triads. But it would be a mistake to give them more arguments than that.  
Vì vậy, tất cả các quy tắc tương tự được áp dụng. Các hàm nhận đối số biến có thể là đơn nguyên, cặp đôi hoặc thậm chí là bộ ba. Nhưng sẽ là một sai lầm nếu cho họ nhiều lý lẽ hơn thế.

void monad(Integer... args);

void dyad(String name, Integer... args);

void triad(String name, int count, Integer... args);

[**Verbs and Keywords**](#_page_91_0)  
Động từ và từ khóa

Choosing good names for a function can go a long way toward explaining the intent of the function and the order and intent of the arguments. In the case of a monad, the function and argument should form a very nice verb/noun pair. For example, write(name)is very evocative. Whatever this “name” thing is, it is being “written.” An even better name might be writeField(name), which tells us that the “name” thing is a “ﬁeld.”  
Việc chọn những cái tên hay cho một hàm có thể giúp ích rất nhiều trong việc giải thích mục đích của hàm cũng như thứ tự và mục đích của các đối số. Trong trường hợp của một đơn nguyên, chức năng và đối số sẽ tạo thành một cặp động từ/danh từ rất hay. Ví dụ, viết (tên) rất gợi. Dù thứ “tên” này là gì, thì nó cũng đang được “viết”. Một cái tên thậm chí còn tốt hơn có thể là writeField(name), cho chúng ta biết rằng thứ “tên” là một “trường”.

This last is an example of the *keyword* form of a function name. Using this form we encode the names of the arguments into the function name. For example, assertEquals might be better written as assertExpectedEqualsActual(expected,actual). This strongly mitigates the problem of having to remember the ordering of the arguments.  
Điều cuối cùng này là một ví dụ về dạng từ khóa của tên hàm. Sử dụng biểu mẫu này, chúng tôi mã hóa tên của các đối số thành tên hàm. Ví dụ: assertEquals có thể được viết tốt hơn là assertExpectedEqualsActual(expected,actual). Điều này giảm thiểu đáng kể vấn đề phải nhớ thứ tự của các đối số.

<https://www.facebook.com/groups/phattrienlaptrinh>

44 **Chapter 3: Functions**  
44 Chương 3: Hàm

[**Have No Side Effects**](#_page_91_0)  
Không có tác dụng phụ

Side effects are lies. Your function promises to do one thing, but it also does other *hidden* things. Sometimes it will make unexpected changes to the variables of its own class. Sometimes it will make them to the parameters passed into the function or to system glo-bals. In either case they are devious and damaging mistruths that often result in strange temporal couplings and order dependencies.  
Tác dụng phụ là dối trá. Chức năng của bạn hứa hẹn sẽ làm một việc, nhưng nó cũng thực hiện những việc ẩn khác. Đôi khi nó sẽ thực hiện những thay đổi không mong muốn đối với các biến của chính lớp của nó. Đôi khi nó sẽ biến chúng thành các tham số được truyền vào hàm hoặc thành các khối toàn cầu của hệ thống. Trong cả hai trường hợp, chúng đều là những sự thật sai lầm và tai hại, thường dẫn đến những mối liên hệ kỳ lạ về thời gian và sự phụ thuộc vào trật tự.

Consider, for example, the seemingly innocuous function in Listing 3-6. This function uses a standard algorithm to match a userNameto a password. It returns trueif they match and false if anything goes wrong. But it also has a side effect. Can you spot it?  
Ví dụ, hãy xem xét hàm dường như vô hại trong Liệt kê 3-6. Hàm này sử dụng thuật toán tiêu chuẩn để khớp tên người dùng với mật khẩu. Nó trả về true nếu chúng khớp và false nếu có gì sai. Nhưng nó cũng có tác dụng phụ. Bạn có thể phát hiện ra nó?

**Listing 3-6 UserValidator.java  
Liệt kê 3-6 UserValidator.java**

public class UserValidator {

private Cryptographer cryptographer;

public boolean checkPassword(String userName, String password) { User user = UserGateway.findByName(userName);

if (user != User.NULL) {

String codedPhrase = user.getPhraseEncodedByPassword(); String phrase = cryptographer.decrypt(codedPhrase, password); if ("Valid Password".equals(phrase)) {

Session.initialize(); return true;

} }

return false; }

}

The side effect is the call to Session.initialize(), of course. The checkPasswordfunc-tion, by its name, says that it checks the password. The name does not imply that it initial-izes the session. So a caller who believes what the name of the function says runs the risk of erasing the existing session data when he or she decides to check the validity of the user.  
Tất nhiên, tác dụng phụ là cuộc gọi đến Session.initialize(). CheckPasswordfunc-tion, theo tên của nó, nói rằng nó kiểm tra mật khẩu. Tên không ngụ ý rằng nó khởi tạo phiên. Vì vậy, một người gọi tin rằng tên của hàm nói gì sẽ có nguy cơ xóa dữ liệu phiên hiện có khi họ quyết định kiểm tra tính hợp lệ của người dùng.

This side effect creates a temporal coupling. That is, checkPasswordcan only be called at certain times (in other words, when it is safe to initialize the session). If it is called out of order, session data may be inadvertently lost. Temporal couplings are con-fusing, especially when hidden as a side effect. If you must have a temporal coupling, you should make it clear in the name of the function. In this case we might rename the function checkPasswordAndInitializeSession, though that certainly violates “Do one thing.”  
Tác dụng phụ này tạo ra một khớp nối tạm thời. Nghĩa là, checkPassword chỉ có thể được gọi vào những thời điểm nhất định (nói cách khác, khi an toàn để khởi tạo phiên). Nếu nó được gọi không đúng thứ tự, dữ liệu phiên có thể vô tình bị mất. Các khớp nối tạm thời gây nhầm lẫn, đặc biệt khi ẩn dưới dạng tác dụng phụ. Nếu bạn phải có khớp nối thời gian, bạn nên làm rõ điều đó trong tên của hàm. Trong trường hợp này, chúng ta có thể đổi tên hàm checkPasswordAndInitializeSession, mặc dù điều đó chắc chắn vi phạm “Làm một việc”.

<https://www.facebook.com/groups/phattrienlaptrinh>

**Command Query Separation** 45  
Phân tách truy vấn lệnh 45

[**Output Arguments**](#_page_91_0)  
Đối số đầu ra

Arguments are most naturally interpreted as *inputs* to a function. If you have been pro-gramming for more than a few years, I’m sure you’ve done a double-take on an argument that was actually an *output* rather than an input. For example:  
Các đối số được hiểu một cách tự nhiên nhất là đầu vào cho một hàm. Nếu bạn đã lập trình được hơn một vài năm, tôi chắc chắn rằng bạn đã thực hiện hai lần đối với một đối số thực sự là đầu ra hơn là đầu vào. Ví dụ:

appendFooter(s);

Does this function append sas the footer to something? Or does it append some footer to s? Is san input or an output? It doesn’t take long to look at the function signature and see:  
Chức năng này có nối thêm chân trang vào thứ gì đó không? Hay nó nối thêm một số chân trang vào s? San là đầu vào hay đầu ra? Không mất nhiều thời gian để nhìn vào chữ ký hàm và thấy:

public void appendFooter(StringBuffer report)

This clariﬁes the issue, but only at the expense of checking the declaration of the function. Anything that forces you to check the function signature is equivalent to a double-take. It’s a cognitive break and should be avoided.  
Điều này làm rõ vấn đề, nhưng chỉ làm mất đi việc kiểm tra khai báo của hàm. Bất cứ điều gì buộc bạn phải kiểm tra chữ ký hàm đều tương đương với việc thực hiện hai lần. Đó là một sự phá vỡ nhận thức và nên tránh.

In the days before object oriented programming it was sometimes necessary to have output arguments. However, much of the need for output arguments disappears in OO lan-guages because thisis *intended* to act as an output argument. In other words, it would be better for appendFooter to be invoked as  
Trong những ngày trước khi lập trình hướng đối tượng, đôi khi cần phải có các đối số đầu ra. Tuy nhiên, phần lớn nhu cầu về các đối số đầu ra biến mất trong ngôn ngữ OO bởi vì điều này được dự định hoạt động như một đối số đầu ra. Nói cách khác, sẽ tốt hơn nếu appendFooter được gọi là

report.appendFooter();

In general output arguments should be avoided. If your function must change the state of something, have it change the state of its owning object.  
Nói chung, nên tránh các đối số đầu ra. Nếu chức năng của bạn phải thay đổi trạng thái của một cái gì đó, hãy để nó thay đổi trạng thái của đối tượng sở hữu nó.

[**Command Query Separation**](#_page_91_0)  
Tách truy vấn lệnh

Functions should either do something or answer something, but not both. Either your function should change the state of an object, or it should return some information about that object. Doing both often leads to confusion. Consider, for example, the following function:  
Các chức năng nên làm một cái gì đó hoặc trả lời một cái gì đó, nhưng không phải cả hai. Hàm của bạn sẽ thay đổi trạng thái của một đối tượng hoặc nó sẽ trả về một số thông tin về đối tượng đó. Làm cả hai thường dẫn đến nhầm lẫn. Ví dụ, xét hàm sau:

public boolean set(String attribute, String value);

This function sets the value of a named attribute and returns trueif it is successful and false if no such attribute exists. This leads to odd statements like this:  
Hàm này đặt giá trị của thuộc tính được đặt tên và trả về true nếu thành công và trả về false nếu không tồn tại thuộc tính đó. Điều này dẫn đến những tuyên bố kỳ lạ như thế này:

if (set("username", "unclebob"))...

Imagine this from the point of view of the reader. What does it mean? Is it asking whether the “username” attribute was previously set to “unclebob”? Or is it asking whether the “username” attribute was successfully set to “unclebob”? It’s hard to infer the meaning from the call because it’s not clear whether the word “set” is a verb or an adjective.  
Hãy tưởng tượng điều này từ quan điểm của người đọc. Nó có nghĩa là gì? Có phải nó đang hỏi liệu thuộc tính “tên người dùng” trước đó đã được đặt thành “unclebob” hay chưa? Hay nó đang hỏi liệu thuộc tính “tên người dùng” đã được đặt thành công thành “unclebob” hay chưa? Thật khó để suy ra ý nghĩa từ cách gọi vì không rõ từ “set” là động từ hay tính từ.

The author intended setto be a verb, but in the context of the ifstatement it *feels* like an adjective. So the statement reads as “If the usernameattribute was previously set to unclebob” and not “set the usernameattribute to uncleboband if that worked then. . . .” We  
Tác giả dự định setto là một động từ, nhưng trong ngữ cảnh của câu lệnh if, nó giống như một tính từ. Vì vậy, câu lệnh có nội dung là “Nếu thuộc tính tên người dùng trước đây được đặt thành chú bob” chứ không phải “đặt thuộc tính tên người dùng thành chúboband nếu điều đó hoạt động sau đó. . . .” Chúng tôi

<https://www.facebook.com/groups/phattrienlaptrinh>

46 **Chapter 3: Functions**  
46 Chương 3: Hàm

could try to resolve this by renaming the setfunction to setAndCheckIfExists, but that doesn’t much help the readability of the ifstatement. The real solution is to separate the command from the query so that the ambiguity cannot occur.  
có thể cố gắng giải quyết vấn đề này bằng cách đổi tên setfunction thành setAndCheckIfExists, nhưng điều đó không giúp ích nhiều cho khả năng đọc của câu lệnh if. Giải pháp thực sự là tách lệnh khỏi truy vấn để không xảy ra sự mơ hồ.

if (attributeExists("username")) { setAttribute("username", "unclebob"); ...

}

[**Prefer Exceptions to Returning Error Codes**](#_page_165_0)  
Ưu tiên ngoại lệ để trả lại mã lỗi

Returning error codes from command functions is a subtle violation of command query separation. It promotes commands being used as expressions in the predicates of ifstate-ments.  
Việc trả lại mã lỗi từ các chức năng lệnh là một sự vi phạm tinh vi đối với việc phân tách truy vấn lệnh. Nó khuyến khích các lệnh được sử dụng làm biểu thức trong các vị từ của ifstate-ments.

if (deletePage(page) == E\_OK)

This does not suffer from verb/adjective confusion but does lead to deeply nested struc-tures. When you return an error code, you create the problem that the caller must deal with the error immediately.  
Điều này không gây nhầm lẫn động từ/tính từ nhưng dẫn đến các cấu trúc được lồng ghép sâu sắc. Khi bạn trả lại mã lỗi, bạn tạo ra sự cố mà người gọi phải xử lý lỗi ngay lập tức.

if (deletePage(page) == E\_OK) {

if (registry.deleteReference(page.name) == E\_OK) {

if (configKeys.deleteKey(page.name.makeKey()) == E\_OK){ logger.log("page deleted");

} else {

logger.log("configKey not deleted"); }

} else {

logger.log("deleteReference from registry failed"); }

} else {

logger.log("delete failed"); return E\_ERROR;

}

On the other hand, if you use exceptions instead of returned error codes, then the error processing code can be separated from the happy path code and can be simpliﬁed:  
Mặt khác, nếu bạn sử dụng các ngoại lệ thay vì mã lỗi được trả về, thì mã xử lý lỗi có thể được tách ra khỏi mã đường dẫn hạnh phúc và có thể được đơn giản hóa:

try { deletePage(page);

registry.deleteReference(page.name); configKeys.deleteKey(page.name.makeKey());

}

catch (Exception e) { logger.log(e.getMessage());

}

[**Extract Try/Catch Blocks**](#_page_165_0)  
Trích xuất khối Try/Catch

Try/catchblocks are ugly in their own right. They confuse the structure of the code and mix error processing with normal processing. So it is better to extract the bodies of the try and catch blocks out into functions of their own.  
Try/catchblocks rất xấu theo đúng nghĩa của chúng. Họ nhầm lẫn cấu trúc của mã và trộn xử lý lỗi với xử lý bình thường. Vì vậy, tốt hơn là trích xuất các nội dung của khối try and catch thành các chức năng của riêng chúng.

<https://www.facebook.com/groups/phattrienlaptrinh>

**Prefer Exceptions to Returning Error Codes** 47  
Ưu tiên ngoại lệ để trả lại mã lỗi 47

public void delete(Page page) { try {

deletePageAndAllReferences(page); }

catch (Exception e) { logError(e);

} }

private void deletePageAndAllReferences(Page page) throws Exception { deletePage(page);

registry.deleteReference(page.name); configKeys.deleteKey(page.name.makeKey());

}

private void logError(Exception e) { logger.log(e.getMessage());

}

In the above, the delete function is all about error processing. It is easy to understand and then ignore. The deletePageAndAllReferencesfunction is all about the processes of fully deleting a page. Error handling can be ignored. This provides a nice separation that makes the code easier to understand and modify.  
Ở trên, chức năng xóa là tất cả về xử lý lỗi. Nó rất dễ hiểu và sau đó bỏ qua. Hàm deletePageAndAllReferences liên quan đến quá trình xóa hoàn toàn một trang. Xử lý lỗi có thể được bỏ qua. Điều này cung cấp một sự tách biệt tốt giúp mã dễ hiểu và dễ sửa đổi hơn.

[**Error Handling Is One Thing**](#_page_165_0)  
Xử lý lỗi là một chuyện

Functions should do one thing. Error handing is one thing. Thus, a function that handles errors should do nothing else. This implies (as in the example above) that if the keyword tryexists in a function, it should be the very ﬁrst word in the function and that there should be nothing after the catch/finally blocks.  
Chức năng nên làm một việc. Bàn giao lỗi là một chuyện. Do đó, một chức năng xử lý lỗi sẽ không làm gì khác. Điều này ngụ ý (như trong ví dụ trên) rằng nếu từ khóa try tồn tại trong một hàm, thì nó phải là từ đầu tiên trong hàm và không được có gì sau khối catch/cuối cùng.

[**The Error.java Dependency Magnet**](#_page_165_0)  
Nam châm phụ thuộc Error.java

Returning error codes usually implies that there is some class or enum in which all the error codes are deﬁned.  
Trả về các mã lỗi thường ám chỉ rằng có một số lớp hoặc enum trong đó tất cả các mã lỗi được xác định.

public enum Error { OK,

INVALID, NO\_SUCH, LOCKED,

OUT\_OF\_RESOURCES, WAITING\_FOR\_EVENT;

}

Classes like this are a *dependency magnet;* many other classes must import and use them. Thus, when the Errorenumchanges, all those other classes need to be recompiled and redeployed.11 This puts a negative pressure on the Errorclass. Programmers don’t want  
Các lớp như thế này là một nam châm phụ thuộc; nhiều lớp khác phải nhập và sử dụng chúng. Vì vậy, khi Errorenumthay đổi, tất cả các lớp khác cần phải được biên dịch lại và triển khai lại.11 Điều này gây áp lực tiêu cực lên Errorclass. Lập trình viên không muốn

11. Those who felt that they could get away without recompiling and redeploying have been found—and dealt with.  
11. Những người cảm thấy rằng họ có thể trốn thoát mà không cần biên dịch lại và triển khai lại đã bị phát hiện—và bị xử lý.

<https://www.facebook.com/groups/phattrienlaptrinh>

48 **Chapter 3: Functions**  
48 Chương 3: Hàm

to add new errors because then they have to rebuild and redeploy everything. So they reuse old error codes instead of adding new ones.  
để thêm các lỗi mới vì sau đó họ phải xây dựng lại và triển khai lại mọi thứ. Vì vậy, họ sử dụng lại mã lỗi cũ thay vì thêm mã lỗi mới.

When you use exceptions rather than error codes, then new exceptions are *derivatives* of the exception class. They can be added without forcing any recompilation or redeployment.12  
Khi bạn sử dụng các ngoại lệ thay vì mã lỗi, thì các ngoại lệ mới là dẫn xuất của lớp ngoại lệ. Chúng có thể được thêm vào mà không buộc phải biên dịch lại hoặc triển khai lại.12

[**Don’t Repeat Yourself13**](#_page_165_0)  
Đừng Lặp Lại Chính Mình13

Look back at Listing 3-1 carefully and you will notice that there is an algorithm that gets repeated four times, once for each of the SetUp, SuiteSetUp, TearDown, and SuiteTearDowncases. It’s not easy to spot this duplication because the four instances are intermixed with other code and aren’t uniformly duplicated. Still, the duplication is a problem because it bloats the code and  
Hãy xem lại Liệt kê 3-1 một cách cẩn thận và bạn sẽ nhận thấy rằng có một thuật toán được lặp lại bốn lần, một lần cho mỗi SetUp, SuiteSetUp, TearDown và SuiteTearDowncase. Không dễ để phát hiện sự trùng lặp này vì bốn trường hợp được trộn lẫn với mã khác và không trùng lặp đồng đều. Tuy nhiên, sự trùng lặp là một vấn đề vì nó làm phình to mã và

will require four-fold modiﬁcation should the algorithm ever have to change. It is also a four-fold opportunity for an error of omission.  
sẽ yêu cầu sửa đổi bốn lần nếu thuật toán phải thay đổi. Đó cũng là một cơ hội gấp bốn lần cho một lỗi thiếu sót.

This duplication was remedied by the includemethod in Listing 3-7. Read through that code again and notice how the readability of the whole module is enhanced by the reduction of that duplication.  
Sự trùng lặp này đã được khắc phục bằng phương pháp bao gồm trong Liệt kê 3-7. Đọc lại mã đó và nhận thấy khả năng đọc của toàn bộ mô-đun được tăng cường như thế nào bằng cách giảm sự trùng lặp đó.

Duplication may be the root of all evil in software. Many principles and practices have been created for the purpose of controlling or eliminating it. Consider, for example, that all of Codd’s database normal forms serve to eliminate duplication in data. Consider also how object-oriented programming serves to concentrate code into base classes that would otherwise be redundant. Structured programming, Aspect Oriented Programming, Compo-nent Oriented Programming, are all, in part, strategies for eliminating duplication. It would appear that since the invention of the subroutine, innovations in software develop-ment have been an ongoing attempt to eliminate duplication from our source code.  
Sao chép có thể là gốc rễ của mọi tội lỗi trong phần mềm. Nhiều nguyên tắc và thực hành đã được tạo ra với mục đích kiểm soát hoặc loại bỏ nó. Ví dụ, hãy xem xét rằng tất cả các biểu mẫu thông thường trong cơ sở dữ liệu của Codd đều dùng để loại bỏ sự trùng lặp trong dữ liệu. Cũng xem xét cách lập trình hướng đối tượng phục vụ để tập trung mã vào các lớp cơ sở mà nếu không sẽ là dư thừa. Lập trình có cấu trúc, Lập trình hướng theo khía cạnh, Lập trình hướng theo hợp chất, tất cả, một phần, là các chiến lược để loại bỏ sự trùng lặp. Có vẻ như kể từ khi phát minh ra chương trình con, những đổi mới trong phát triển phần mềm đã là một nỗ lực không ngừng nhằm loại bỏ sự trùng lặp khỏi mã nguồn của chúng ta.

[**Structured Programming**](#_page_165_0)  
Lập trình có cấu trúc

Some programmers follow Edsger Dijkstra’s rules of structured programming.14 Dijkstra said that every function, and every block within a function, should have one entry and one exit. Following these rules means that there should only be one returnstatement in a func-tion, no break or continue statements in a loop, and never, *ever,* any goto statements.  
Một số lập trình viên tuân theo các quy tắc lập trình có cấu trúc của Edsger Dijkstra.14 Dijkstra nói rằng mọi chức năng và mọi khối bên trong một chức năng nên có một mục nhập và một lối ra. Tuân thủ các quy tắc này có nghĩa là chỉ nên có một câu lệnh trả về trong một hàm, không có câu lệnh break hoặc continue trong một vòng lặp và không bao giờ có bất kỳ câu lệnh goto nào.

12. This is an example of the Open Closed Principle (OCP) [PPP02]. 13. The DRY principle. [PRAG].  
12. Đây là một ví dụ về Nguyên tắc Đóng Mở (OCP) [PPP02]. 13. Nguyên tắc KHÔ. [PRAG].

14. [SP72].

<https://www.facebook.com/groups/phattrienlaptrinh>

**Conclusion** 49  
Kết luận 49

While we are sympathetic to the goals and disciplines of structured programming, those rules serve little beneﬁt when functions are very small. It is only in larger functions that such rules provide signiﬁcant beneﬁt.  
Mặc dù chúng ta đồng cảm với các mục tiêu và nguyên tắc của lập trình có cấu trúc, nhưng các quy tắc đó không mang lại nhiều lợi ích khi các chức năng rất nhỏ. Chỉ trong các chức năng lớn hơn, các quy tắc như vậy mới mang lại lợi ích đáng kể.

So if you keep your functions small, then the occasional multiple return, break, or continuestatement does no harm and can sometimes even be more expressive than the sin-gle-entry, single-exit rule. On the other hand, gotoonly makes sense in large functions, so it should be avoided.  
Vì vậy, nếu bạn giữ cho các chức năng của mình ở mức nhỏ, thì câu lệnh trả về, ngắt hoặc tiếp tục không thường xuyên sẽ không gây hại và đôi khi thậm chí có thể biểu cảm hơn quy tắc một lần nhập, một lần thoát. Mặt khác, gotoon chỉ có ý nghĩa trong các chức năng lớn, vì vậy nên tránh sử dụng nó.

[**How Do You Write Functions Like This?**](#_page_165_0)  
Làm thế nào để bạn viết các chức năng như thế này?

Writing software is like any other kind of writing. When you write a paper or an article, you get your thoughts down ﬁrst, then you massage it until it reads well. The ﬁrst draft might be clumsy and disorganized, so you wordsmith it and restructure it and reﬁne it until it reads the way you want it to read.  
Viết phần mềm cũng giống như bất kỳ loại văn bản nào khác. Khi bạn viết một bài báo hoặc một bài báo, trước tiên bạn phải suy nghĩ kỹ càng, sau đó bạn xoa bóp nó cho đến khi nó đọc tốt. Bản nháp đầu tiên có thể vụng về và lộn xộn, vì vậy bạn sắp xếp lại từ ngữ, cấu trúc lại và tinh chỉnh cho đến khi nó đọc theo cách bạn muốn.

When I write functions, they come out long and complicated. They have lots of indenting and nested loops. They have long argument lists. The names are arbitrary, and there is duplicated code. But I also have a suite of unit tests that cover every one of those clumsy lines of code.  
Khi tôi viết các hàm, chúng rất dài và phức tạp. Chúng có rất nhiều vòng lặp thụt lề và lồng nhau. Họ có danh sách đối số dài. Các tên là tùy ý và có mã trùng lặp. Nhưng tôi cũng có một bộ bài kiểm tra đơn vị bao gồm mọi dòng mã vụng về đó.

So then I massage and reﬁne that code, splitting out functions, changing names, elim-inating duplication. I shrink the methods and reorder them. Sometimes I break out whole classes, all the while keeping the tests passing.  
Vì vậy, sau đó tôi xoa bóp và tinh chỉnh đoạn mã đó, tách các chức năng ra, thay đổi tên, loại bỏ sự trùng lặp. Tôi thu nhỏ các phương pháp và sắp xếp lại chúng. Đôi khi tôi chia nhỏ cả lớp, trong khi vẫn đảm bảo vượt qua các bài kiểm tra.

In the end, I wind up with functions that follow the rules I’ve laid down in this chapter. I don’t write them that way to start. I don’t think anyone could.  
Cuối cùng, tôi kết thúc với các chức năng tuân theo các quy tắc mà tôi đã đặt ra trong chương này. Tôi không viết chúng theo cách đó để bắt đầu. Tôi không nghĩ rằng bất cứ ai có thể.

[**Conclusion**](#_page_165_0)  
Phần kết luận

Every system is built from a domain-speciﬁc language designed by the programmers to describe that system. Functions are the verbs of that language, and classes are the nouns. This is not some throwback to the hideous old notion that the nouns and verbs in a require-ments document are the ﬁrst guess of the classes and functions of a system. Rather, this is a much older truth. The art of programming is, and has always been, the art of language design.  
Mỗi hệ thống được xây dựng từ một ngôn ngữ miền cụ thể do các lập trình viên thiết kế để mô tả hệ thống đó. Hàm là động từ của ngôn ngữ đó và lớp là danh từ. Đây không phải là sự quay ngược trở lại quan niệm cổ hủ gớm ghiếc rằng các danh từ và động từ trong tài liệu yêu cầu là phỏng đoán đầu tiên về các lớp và chức năng của một hệ thống. Thay vào đó, đây là một sự thật cũ hơn nhiều. Nghệ thuật lập trình luôn luôn là nghệ thuật thiết kế ngôn ngữ.

Master programmers think of systems as stories to be told rather than programs to be written. They use the facilities of their chosen programming language to construct a much richer and more expressive language that can be used to tell that story. Part of that domain-speciﬁc language is the hierarchy of functions that describe all the actions that take place within that system. In an artful act of recursion those actions are written to use the very domain-speciﬁc language they deﬁne to tell their own small part of the story.  
Các lập trình viên bậc thầy nghĩ về các hệ thống như những câu chuyện để kể hơn là các chương trình được viết ra. Họ sử dụng các phương tiện của ngôn ngữ lập trình đã chọn để xây dựng một ngôn ngữ phong phú hơn và biểu cảm hơn có thể được sử dụng để kể câu chuyện đó. Một phần của ngôn ngữ dành riêng cho miền đó là hệ thống phân cấp chức năng mô tả tất cả các hành động diễn ra trong hệ thống đó. Trong một hành động đệ quy đầy nghệ thuật, những hành động đó được viết để sử dụng ngôn ngữ rất cụ thể của miền mà chúng xác định để kể phần nhỏ câu chuyện của chính chúng.

This chapter has been about the mechanics of writing functions well. If you follow the rules herein, your functions will be short, well named, and nicely organized. But  
Chương này nói về cơ chế viết hàm tốt. Nếu bạn tuân theo các quy tắc ở đây, các chức năng của bạn sẽ ngắn gọn, được đặt tên rõ ràng và được tổ chức độc đáo. Nhưng

<https://www.facebook.com/groups/phattrienlaptrinh>

50 **Chapter 3: Functions**  
50 Chương 3: Hàm

never forget that your real goal is to tell the story of the system, and that the functions you write need to ﬁt cleanly together into a clear and precise language to help you with that telling.  
đừng bao giờ quên rằng mục tiêu thực sự của bạn là kể câu chuyện về hệ thống và các chức năng bạn viết cần phải khớp với nhau một cách rõ ràng thành một ngôn ngữ rõ ràng và chính xác để giúp bạn kể chuyện đó.

[**SetupTeardownIncluder**](#_page_165_0)  
Thiết lậpTeardownIncluder

**Listing 3-7 SetupTeardownIncluder.java** package fitnesse.html;  
Liệt kê 3-7 gói SetupTeardownIncluder.java fitnesse.html;

import fitnesse.responders.run.SuiteResponder; import fitnesse.wiki.\*;

public class SetupTeardownIncluder { private PageData pageData; private boolean isSuite;

private WikiPage testPage;

private StringBuffer newPageContent; private PageCrawler pageCrawler;

public static String render(PageData pageData) throws Exception { return render(pageData, false);

}

public static String render(PageData pageData, boolean isSuite) throws Exception {

return new SetupTeardownIncluder(pageData).render(isSuite); }

private SetupTeardownIncluder(PageData pageData) { this.pageData = pageData;

testPage = pageData.getWikiPage(); pageCrawler = testPage.getPageCrawler(); newPageContent = new StringBuffer();

}

private String render(boolean isSuite) throws Exception { this.isSuite = isSuite;

if (isTestPage()) includeSetupAndTeardownPages();

return pageData.getHtml(); }

private boolean isTestPage() throws Exception { return pageData.hasAttribute("Test");

}

private void includeSetupAndTeardownPages() throws Exception { includeSetupPages();

includePageContent(); includeTeardownPages(); updatePageContent();

}

<https://www.facebook.com/groups/phattrienlaptrinh>

**SetupTeardownIncluder** 51  
Thiết lậpTeardownIncluder 51

**Listing 3-7 (continued) SetupTeardownIncluder.java**  
Liệt kê 3-7 (tiếp theo) SetupTeardownIncluder.java

private void includeSetupPages() throws Exception { if (isSuite)

includeSuiteSetupPage(); includeSetupPage();

}

private void includeSuiteSetupPage() throws Exception { include(SuiteResponder.SUITE\_SETUP\_NAME, "-setup");

}

private void includeSetupPage() throws Exception { include("SetUp", "-setup");

}

private void includePageContent() throws Exception { newPageContent.append(pageData.getContent());

}

private void includeTeardownPages() throws Exception { includeTeardownPage();

if (isSuite) includeSuiteTeardownPage();

}

private void includeTeardownPage() throws Exception { include("TearDown", "-teardown");

}

private void includeSuiteTeardownPage() throws Exception { include(SuiteResponder.SUITE\_TEARDOWN\_NAME, "-teardown");

}

private void updatePageContent() throws Exception { pageData.setContent(newPageContent.toString());

}

private void include(String pageName, String arg) throws Exception { WikiPage inheritedPage = findInheritedPage(pageName);

if (inheritedPage != null) {

String pagePathName = getPathNameForPage(inheritedPage); buildIncludeDirective(pagePathName, arg);

} }

private WikiPage findInheritedPage(String pageName) throws Exception { return PageCrawlerImpl.getInheritedPage(pageName, testPage);

}

private String getPathNameForPage(WikiPage page) throws Exception { WikiPagePath pagePath = pageCrawler.getFullPath(page);

return PathParser.render(pagePath); }

private void buildIncludeDirective(String pagePathName, String arg) { newPageContent

.append("\n!include ")

<https://www.facebook.com/groups/phattrienlaptrinh>

52 **Chapter 3: Functions**  
52 Chương 3: Hàm

**Listing 3-7 (continued) SetupTeardownIncluder.java  
Liệt kê 3-7 (tiếp theo) SetupTeardownIncluder.java**

.append(arg) .append(" .") .append(pagePathName) .append("\n");

} }

[**Bibliography**](#_page_165_0)  
Thư mục

**[KP78]:** Kernighan and Plaugher, *The Elements of Programming Style*, 2d. ed., McGraw-Hill, 1978.

**[PPP02]:** Robert C. Martin, *Agile Software Development: Principles, Patterns, and Prac-tices*, Prentice Hall, 2002.

**[GOF]:** *Design Patterns: Elements of Reusable Object Oriented Software*, Gamma et al., Addison-Wesley, 1996.

**[PRAG]:** *The Pragmatic Programmer*, Andrew Hunt, Dave Thomas, Addison-Wesley, 2000.

**[SP72]:** *Structured Programming*, O.-J. Dahl, E. W. Dijkstra, C. A. R. Hoare, Academic Press, London, 1972.

<https://www.facebook.com/groups/phattrienlaptrinh>



[**4**](#_page_165_0)

[**Comments**](#_page_165_0)  
Bình luận

*“Don’t comment bad code—rewrite it.”  
“Đừng bình luận mã xấu—hãy viết lại nó.”*

—Brian W. Kernighan and P. J. Plaugher1

Nothing can be quite so helpful as a well-placed comment. Nothing can clutter up a mod-ule more than frivolous dogmatic comments. Nothing can be quite so damaging as an old crufty comment that propagates lies and misinformation.  
Không gì có thể hữu ích bằng một nhận xét được đặt đúng chỗ. Không có gì có thể làm lộn xộn một mô-đun hơn là những nhận xét giáo điều phù phiếm. Không có gì có thể gây tổn hại nghiêm trọng bằng một nhận xét thô lỗ cũ truyền bá những lời dối trá và thông tin sai lệch.

Comments are not like Schindler’s List. They are not “pure good.” Indeed, comments are, at best, a necessary evil. If our programming languages were expressive enough, or if  
Bình luận không giống như Schindler’s List. Chúng không phải là "tốt thuần túy." Thật vậy, các bình luận, tốt nhất, là một điều ác cần thiết. Nếu ngôn ngữ lập trình của chúng tôi đủ biểu cảm, hoặc nếu

1. [KP78], p. 144.

53

<https://www.facebook.com/groups/phattrienlaptrinh>

54 **Chapter 4: Comments**  
54 Chương 4: Bình Luận

we had the talent to subtly wield those languages to express our intent, we would not need comments very much—perhaps not at all.  
chúng tôi có tài sử dụng những ngôn ngữ đó một cách tinh tế để diễn đạt ý định của mình, chúng tôi sẽ không cần bình luận nhiều lắm—có lẽ là không.

The proper use of comments is to compensate for our failure to express ourself in code. Note that I used the word *failure*. I meant it. Comments are always failures. We must have them because we cannot always ﬁgure out how to express ourselves without them, but their use is not a cause for celebration.  
Việc sử dụng bình luận đúng cách là để bù đắp cho việc chúng ta không thể hiện bản thân bằng mã. Lưu ý rằng tôi đã sử dụng từ thất bại. Tôi có ý đó. Nhận xét luôn là thất bại. Chúng ta phải có chúng bởi vì chúng ta không phải lúc nào cũng tìm ra cách thể hiện bản thân mà không có chúng, nhưng việc sử dụng chúng không phải là lý do để ăn mừng.

So when you ﬁnd yourself in a position where you need to write a comment, think it through and see whether there isn’t some way to turn the tables and express yourself in code. Every time you express yourself in code, you should pat yourself on the back. Every time you write a comment, you should grimace and feel the failure of your ability of expression.  
Vì vậy, khi bạn thấy mình ở một vị trí cần viết bình luận, hãy suy nghĩ kỹ và xem liệu có cách nào để lật ngược tình thế và thể hiện bản thân bằng mã hay không. Mỗi khi bạn thể hiện bản thân bằng mã, bạn nên tự vỗ về mình. Mỗi lần viết bình luận, bạn phải nhăn mặt và cảm thấy sự thất bại trong khả năng diễn đạt của mình.

Why am I so down on comments? Because they lie. Not always, and not intentionally, but too often. The older a comment is, and the farther away it is from the code it describes, the more likely it is to be just plain wrong. The reason is simple. Programmers can’t realis-tically maintain them.  
Tại sao tôi rất ít bình luận? Bởi vì họ nói dối. Không phải luôn luôn, và không cố ý, nhưng quá thường xuyên. Một bình luận càng cũ và càng cách xa mã mà nó mô tả, thì càng có nhiều khả năng nó hoàn toàn sai. Lý do rất đơn giản. Các lập trình viên không thể duy trì chúng một cách thực tế.

Code changes and evolves. Chunks of it move from here to there. Those chunks bifur-cate and reproduce and come together again to form chimeras. Unfortunately the com-ments don’t always follow them—*can’t* always follow them. And all too often the comments get separated from the code they describe and become orphaned blurbs of ever-decreasing accuracy. For example, look what has happened to this comment and the line it was intended to describe:  
Mã thay đổi và phát triển. Chunks của nó di chuyển từ đây đến đó. Những khối đó chia đôi, sinh sản và kết hợp lại với nhau để tạo thành chimeras. Thật không may, các nhận xét không phải lúc nào cũng theo sau chúng—không phải lúc nào cũng theo sau chúng. Và tất cả các bình luận thường bị tách khỏi mã mà chúng mô tả và trở thành những điểm mờ mồ côi với độ chính xác ngày càng giảm. Ví dụ: hãy xem điều gì đã xảy ra với nhận xét này và dòng mà nó dự định mô tả:

MockRequest request;

private final String HTTP\_DATE\_REGEXP =

"[SMTWF][a-z]{2}\\,\\s[0-9]{2}\\s[JFMASOND][a-z]{2}\\s"+ "[0-9]{4}\\s[0-9]{2}\\:[0-9]{2}\\:[0-9]{2}\\sGMT";

private Response response; private FitNesseContext context; private FileResponder responder; private Locale saveLocale;

**// Example: "Tue, 02 Apr 2003 22:18:49 GMT"**

Other instance variables that were probably added later were interposed between the HTTP\_DATE\_REGEXP constant and it’s explanatory comment.  
Các biến thể hiện khác có thể được thêm vào sau đó được đặt giữa hằng số HTTP\_DATE\_REGEXP và nhận xét giải thích của nó.

It is possible to make the point that programmers should be disciplined enough to keep the comments in a high state of repair, relevance, and accuracy. I agree, they should. But I would rather that energy go toward making the code so clear and expressive that it does not need the comments in the ﬁrst place.  
Có thể đưa ra quan điểm rằng các lập trình viên phải đủ kỷ luật để giữ cho các bình luận ở trạng thái sửa chữa, phù hợp và chính xác cao. Tôi đồng ý, họ nên. Nhưng tôi muốn năng lượng đó hướng tới việc làm cho mã rõ ràng và biểu cảm đến mức nó không cần bình luận ngay từ đầu.

Inaccurate comments are far worse than no comments at all. They delude and mislead. They set expectations that will never be fulﬁlled. They lay down old rules that need not, or should not, be followed any longer.  
Nhận xét không chính xác còn tệ hơn nhiều so với không có nhận xét nào cả. Họ lừa dối và lừa dối. Họ đặt ra những kỳ vọng sẽ không bao giờ được đáp ứng. Họ đặt ra những quy tắc cũ không cần hoặc không nên tuân theo nữa.

Truth can only be found in one place: the code. Only the code can truly tell you what it does. It is the only source of truly accurate information. Therefore, though comments are sometimes necessary, we will expend signiﬁcant energy to minimize them.  
Sự thật chỉ có thể được tìm thấy ở một nơi: mật mã. Chỉ mã thực sự có thể cho bạn biết những gì nó làm. Đó là nguồn thông tin thực sự chính xác duy nhất. Do đó, mặc dù các bình luận đôi khi là cần thiết, nhưng chúng tôi sẽ tiêu tốn năng lượng đáng kể để giảm thiểu chúng.

<https://www.facebook.com/groups/phattrienlaptrinh>

**Good Comments** 55  
Ý kiến hay 55

[**Comments Do Not Make Up for Bad Code**](#_page_165_0)  
Nhận xét không bù đắp cho mã xấu

One of the more common motivations for writing comments is bad code. We write a mod-ule and we know it is confusing and disorganized. We know it’s a mess. So we say to our-selves, “Ooh, I’d better comment that!” No! You’d better clean it!  
Một trong những động lực phổ biến hơn để viết bình luận là mã xấu. Chúng tôi viết một mô-đun và chúng tôi biết nó khó hiểu và vô tổ chức. Chúng tôi biết đó là một mớ hỗn độn. Vì vậy, chúng tôi tự nói với chính mình, "Ồ, tốt hơn là tôi nên nhận xét điều đó!" KHÔNG! Bạn nên làm sạch nó tốt hơn!

Clear and expressive code with few comments is far superior to cluttered and complex code with lots of comments. Rather than spend your time writing the comments that explain the mess you’ve made, spend it cleaning that mess.  
Mã rõ ràng và biểu cảm với ít bình luận tốt hơn nhiều so với mã lộn xộn và phức tạp với nhiều bình luận. Thay vì dành thời gian viết bình luận giải thích mớ hỗn độn mà bạn đã tạo ra, hãy dành thời gian để dọn dẹp mớ hỗn độn đó.

[**Explain Yourself in Code**](#_page_165_0)  
Giải thích bản thân bằng mã

There are certainly times when code makes a poor vehicle for explanation. Unfortunately, many programmers have taken this to mean that code is seldom, if ever, a good means for explanation. This is patently false. Which would you rather see? This:  
Chắc chắn có những lúc mã tạo ra một phương tiện kém để giải thích. Thật không may, nhiều lập trình viên đã coi điều này có nghĩa là mã hiếm khi, nếu có, là một phương tiện tốt để giải thích. Điều này rõ ràng là sai. Bạn muốn xem cái nào hơn? Cái này:

// Check to see if the employee is eligible for full benefits if ((employee.flags & HOURLY\_FLAG) &&

(employee.age > 65))

Or this?  
Hay cái này?

if (employee.isEligibleForFullBenefits())

It takes only a few seconds of thought to explain most of your intent in code. In many cases it’s simply a matter of creating a function that says the same thing as the comment you want to write.  
Chỉ mất vài giây suy nghĩ để giải thích hầu hết ý định của bạn trong mã. Trong nhiều trường hợp, vấn đề chỉ đơn giản là tạo một chức năng nói điều tương tự như nhận xét mà bạn muốn viết.

[**Good Comments**](#_page_165_0)  
Nhận xét tốt

Some comments are necessary or beneﬁcial. We’ll look at a few that I consider worthy of the bits they consume. Keep in mind, however, that the only truly good comment is the comment you found a way not to write.  
Một số ý kiến ​​là cần thiết hoặc có lợi. Chúng ta sẽ xem xét một số thứ mà tôi cho là xứng đáng với số bit mà chúng tiêu thụ. Tuy nhiên, hãy nhớ rằng nhận xét thực sự tốt duy nhất là nhận xét mà bạn đã tìm ra cách để không viết.

[**Legal Comments**](#_page_165_0)  
Nhận xét pháp lý

Sometimes our corporate coding standards force us to write certain comments for legal reasons. For example, copyright and authorship statements are necessary and reasonable things to put into a comment at the start of each source ﬁle.  
Đôi khi các tiêu chuẩn mã hóa công ty của chúng tôi buộc chúng tôi phải viết một số nhận xét nhất định vì lý do pháp lý. Ví dụ: tuyên bố về bản quyền và quyền tác giả là những điều cần thiết và hợp lý để đưa vào nhận xét ở đầu mỗi tệp nguồn.

Here, for example, is the standard comment header that we put at the beginning of every source ﬁle in FitNesse. I am happy to say that our IDE hides this comment from act-ing as clutter by automatically collapsing it.  
Ví dụ, đây là tiêu đề nhận xét tiêu chuẩn mà chúng tôi đặt ở đầu mỗi tệp nguồn trong FitNesse. Tôi rất vui khi nói rằng IDE của chúng tôi ẩn nhận xét này khỏi hành động lộn xộn bằng cách tự động thu gọn nhận xét đó.

// Copyright (C) 2003,2004,2005 by Object Mentor, Inc. All rights reserved.

// Released under the terms of the GNU General Public License version 2 or later.

<https://www.facebook.com/groups/phattrienlaptrinh>

56 **Chapter 4: Comments**  
56 Chương 4: Bình Luận

Comments like this should not be contracts or legal tomes. Where possible, refer to a stan-dard license or other external document rather than putting all the terms and conditions into the comment.  
Những bình luận như thế này không nên là hợp đồng hoặc sách pháp lý. Nếu có thể, hãy tham khảo giấy phép tiêu chuẩn hoặc tài liệu bên ngoài khác thay vì đưa tất cả các điều khoản và điều kiện vào nhận xét.

[**Informative Comments**](#_page_165_0)  
Nhận xét thông tin

It is sometimes useful to provide basic information with a comment. For example, con-sider this comment that explains the return value of an abstract method:  
Đôi khi nó hữu ích để cung cấp thông tin cơ bản với một bình luận. Ví dụ: hãy xem xét nhận xét này giải thích giá trị trả về của một phương thức trừu tượng:

**// Returns an instance of the Responder being tested.** protected abstract Responder responderInstance();

A comment like this can sometimes be useful, but it is better to use the name of the func-tion to convey the information where possible. For example, in this case the comment could be made redundant by renaming the function: responderBeingTested.  
Một nhận xét như thế này đôi khi có thể hữu ích, nhưng tốt hơn là sử dụng tên của chức năng để truyền đạt thông tin nếu có thể. Ví dụ: trong trường hợp này, nhận xét có thể trở nên dư thừa bằng cách đổi tên hàm: ReplyerBeingTested.

Here’s a case that’s a bit better:  
Đây là một trường hợp tốt hơn một chút:

// format matched kk:mm:ss EEE, MMM dd, yyyy Pattern timeMatcher = Pattern.compile(

"\\d\*:\\d\*:\\d\* \\w\*, \\w\* \\d\*, \\d\*");

In this case the comment lets us know that the regular expression is intended to match a time and date that were formatted with the SimpleDateFormat.formatfunction using the speciﬁed format string. Still, it might have been better, and clearer, if this code had been moved to a special class that converted the formats of dates and times. Then the comment would likely have been superﬂuous.  
Trong trường hợp này, nhận xét cho chúng tôi biết rằng biểu thức chính quy nhằm khớp với thời gian và ngày tháng được định dạng bằng hàm SimpleDateFormat.format bằng cách sử dụng chuỗi định dạng đã chỉ định. Tuy nhiên, nó có thể tốt hơn và rõ ràng hơn nếu mã này được chuyển sang một lớp đặc biệt để chuyển đổi định dạng của ngày tháng và thời gian. Sau đó, bình luận có thể sẽ là thừa.

[**Explanation of Intent**](#_page_165_0)  
Giải thích ý định

Sometimes a comment goes beyond just useful information about the implementation and provides the intent behind a decision. In the following case we see an interesting decision documented by a comment. When comparing two objects, the author decided that he wanted to sort objects of his class higher than objects of any other.  
Đôi khi, một nhận xét vượt ra ngoài thông tin hữu ích về việc triển khai và cung cấp mục đích đằng sau một quyết định. Trong trường hợp sau đây, chúng tôi thấy một quyết định thú vị được ghi lại bằng một bình luận. Khi so sánh hai đối tượng, tác giả quyết định rằng anh ta muốn sắp xếp các đối tượng thuộc lớp của mình cao hơn các đối tượng thuộc bất kỳ đối tượng nào khác.

public int compareTo(Object o) {

if(o instanceof WikiPagePath) {

WikiPagePath p = (WikiPagePath) o;

String compressedName = StringUtil.join(names, ""); String compressedArgumentName = StringUtil.join(p.names, ""); return compressedName.compareTo(compressedArgumentName);

}

return 1; **// we are greater because we are the right type.** }

Here’s an even better example. You might not agree with the programmer’s solution to the problem, but at least you know what he was trying to do.  
Đây là một ví dụ thậm chí còn tốt hơn. Bạn có thể không đồng ý với giải pháp của lập trình viên cho vấn đề, nhưng ít nhất bạn biết anh ta đang cố gắng làm gì.

public void testConcurrentAddWidgets() throws Exception { WidgetBuilder widgetBuilder =

new WidgetBuilder(new Class[]{BoldWidget.class});

<https://www.facebook.com/groups/phattrienlaptrinh>

**Good Comments** 57  
Ý kiến hay 57

String text = "'''bold text'''"; ParentWidget parent =

new BoldWidget(new MockWidgetRoot(), "'''bold text'''"); AtomicBoolean failFlag = new AtomicBoolean(); failFlag.set(false);

**//This is our best attempt to get a race condition //by creating large number of threads.**

for (int i = 0; i < 25000; i++) { WidgetBuilderThread widgetBuilderThread =

new WidgetBuilderThread(widgetBuilder, text, parent, failFlag); Thread thread = new Thread(widgetBuilderThread);

thread.start(); }

assertEquals(false, failFlag.get()); }

[**Clariﬁcation**](#_page_165_0)  
làm rõ

Sometimes it is just helpful to translate the meaning of some obscure argument or return value into something that’s readable. In general it is better to ﬁnd a way to make that argu-ment or return value clear in its own right; but when its part of the standard library, or in code that you cannot alter, then a helpful clarifying comment can be useful.  
Đôi khi, việc dịch nghĩa của một số đối số tối nghĩa hoặc giá trị trả về thành thứ gì đó có thể đọc được sẽ rất hữu ích. Nói chung, tốt hơn hết là tìm cách làm cho đối số đó hoặc giá trị trả về trở nên rõ ràng theo đúng nghĩa của nó; nhưng khi nó là một phần của thư viện chuẩn hoặc trong mã mà bạn không thể thay đổi, thì một nhận xét làm rõ hữu ích có thể hữu ích.

public void testCompareTo() throws Exception {

WikiPagePath a = PathParser.parse("PageA"); WikiPagePath ab = PathParser.parse("PageA.PageB"); WikiPagePath b = PathParser.parse("PageB"); WikiPagePath aa = PathParser.parse("PageA.PageA"); WikiPagePath bb = PathParser.parse("PageB.PageB"); WikiPagePath ba = PathParser.parse("PageB.PageA");

assertTrue(a.compareTo(a) == 0); // a == a assertTrue(a.compareTo(b) != 0); // a != b assertTrue(ab.compareTo(ab) == 0); // ab == ab assertTrue(a.compareTo(b) == -1); // a < b assertTrue(aa.compareTo(ab) == -1); // aa < ab assertTrue(ba.compareTo(bb) == -1); // ba < bb assertTrue(b.compareTo(a) == 1); // b > a assertTrue(ab.compareTo(aa) == 1); // ab > aa assertTrue(bb.compareTo(ba) == 1); // bb > ba

}

There is a substantial risk, of course, that a clarifying comment is incorrect. Go through the previous example and see how difﬁcult it is to verify that they are correct. This explains both why the clariﬁcation is necessary and why it’s risky. So before writing com-ments like this, take care that there is no better way, and then take even more care that they are accurate.  
Tất nhiên, có một rủi ro đáng kể là một nhận xét rõ ràng là không chính xác. Xem lại ví dụ trước và xem việc xác minh rằng chúng đúng là khó như thế nào. Điều này giải thích cả lý do tại sao việc làm rõ là cần thiết và tại sao nó lại rủi ro. Vì vậy, trước khi viết những nhận xét như thế này, hãy cẩn thận rằng không có cách nào tốt hơn, và sau đó hãy cẩn thận hơn nữa để chúng chính xác.

<https://www.facebook.com/groups/phattrienlaptrinh>

58 **Chapter 4: Comments**  
58 Chương 4: Bình Luận

[**Warning of Consequences**](#_page_165_0)  
Cảnh báo hậu quả

Sometimes it is useful to warn other pro-grammers about certain consequences. For example, here is a comment that explains why a particular test case is turned off:  
Đôi khi rất hữu ích khi cảnh báo các lập trình viên khác về những hậu quả nhất định. Ví dụ: đây là một nhận xét giải thích lý do tại sao một trường hợp thử nghiệm cụ thể bị tắt:

**// Don't run unless you // have some time to kill.**

public void \_testWithReallyBigFile() {

writeLinesToFile(10000000);

response.setBody(testFile); response.readyToSend(this);

String responseString = output.toString(); assertSubString("Content-Length: 1000000000", responseString); assertTrue(bytesSent > 1000000000);

}

Nowadays, of course, we’d turn off the test case by using the @Ignore attribute with an appropriate explanatory string. @Ignore("Takes too long to run"). But back in the days before JUnit 4, putting an underscore in front of the method name was a common conven-tion. The comment, while ﬂippant, makes the point pretty well.  
Tất nhiên, ngày nay, chúng tôi sẽ tắt trường hợp thử nghiệm bằng cách sử dụng thuộc tính @Ignore với một chuỗi giải thích thích hợp. @Ignore("Mất quá nhiều thời gian để chạy"). Nhưng vào những ngày trước JUnit 4, việc đặt dấu gạch dưới trước tên phương thức là một quy ước phổ biến. Nhận xét, trong khi ﬂippant, làm cho điểm khá tốt.

Here’s another, more poignant example:  
Đây là một ví dụ khác, sâu sắc hơn:

public static SimpleDateFormat makeStandardHttpDateFormat() {

**//SimpleDateFormat is not thread safe,**

**//so we need to create each instance independently.**

SimpleDateFormat df = new SimpleDateFormat("EEE, dd MMM yyyy HH:mm:ss z"); df.setTimeZone(TimeZone.getTimeZone("GMT"));

return df; }

You might complain that there are better ways to solve this problem. I might agree with you. But the comment, as given here, is perfectly reasonable. It will prevent some overly eager programmer from using a static initializer in the name of efﬁciency.  
Bạn có thể phàn nàn rằng có nhiều cách tốt hơn để giải quyết vấn đề này. Tôi có thể đồng ý với bạn. Nhưng nhận xét, như được đưa ra ở đây, là hoàn toàn hợp lý. Nó sẽ ngăn một số lập trình viên quá háo hức sử dụng trình khởi tạo tĩnh dưới danh nghĩa hiệu quả.

[**TODO Comments**](#_page_165_0)  
Ý kiến TODO

It is sometimes reasonable to leave “To do” notes in the form of //TODOcomments. In the following case, the TODOcomment explains why the function has a degenerate implementa-tion and what that function’s future should be.  
Đôi khi, việc để lại các ghi chú “Việc cần làm” ở dạng //TODOcomments là hợp lý. Trong trường hợp sau đây, TODOcomment giải thích lý do tại sao hàm có cách triển khai suy biến và tương lai của hàm đó sẽ như thế nào.

**//TODO-MdM these are not needed**

**// We expect this to go away when we do the checkout model** protected VersionInfo makeVersion() throws Exception

{

return null; }

<https://www.facebook.com/groups/phattrienlaptrinh>

**Bad Comments** 59  
Bình Luận Xấu 59

TODOs are jobs that the programmer thinks should be done, but for some reason can’t do at the moment. It might be a reminder to delete a deprecated feature or a plea for someone else to look at a problem. It might be a request for someone else to think of a better name or a reminder to make a change that is dependent on a planned event. Whatever else a TODOmight be, it is *not* an excuse to leave bad code in the system.  
TODO là những công việc mà lập trình viên nghĩ là nên làm, nhưng vì lý do nào đó không thể làm vào lúc này. Đó có thể là lời nhắc xóa một tính năng không dùng nữa hoặc lời kêu gọi người khác xem xét vấn đề. Đó có thể là yêu cầu người khác nghĩ ra một cái tên hay hơn hoặc lời nhắc thực hiện thay đổi phụ thuộc vào một sự kiện đã lên kế hoạch. Dù TODO có thể là gì đi chăng nữa, thì đó không phải là cái cớ để để lại mã xấu trong hệ thống.

Nowadays, most good IDEs provide special gestures and features to locate all the TODOcomments, so it’s not likely that they will get lost. Still, you don’t want your code to be littered with TODOs. So scan through them regularly and eliminate the ones you can.  
Ngày nay, hầu hết các IDE tốt đều cung cấp các cử chỉ và tính năng đặc biệt để định vị tất cả các TODOcomments, vì vậy không có khả năng chúng sẽ bị thất lạc. Tuy nhiên, bạn không muốn mã của mình bị lấp đầy bởi TODO. Vì vậy, hãy quét qua chúng thường xuyên và loại bỏ những thứ bạn có thể.

[**Ampliﬁcation**](#_page_165_0)  
khuếch đại

A comment may be used to amplify the importance of something that may otherwise seem inconsequential.  
Một nhận xét có thể được sử dụng để khuếch đại tầm quan trọng của một thứ gì đó có vẻ không quan trọng.

String listItemContent = match.group(3).trim();

**// the trim is real important. It removes the starting // spaces that could cause the item to be recognized // as another list.**

new ListItemWidget(this, listItemContent, this.level + 1); return buildList(text.substring(match.end()));

[**Javadocs in Public APIs**](#_page_165_0)  
Javadocs trong API công khai

There is nothing quite so helpful and satisfying as a well-described public API. The java-docs for the standard Java library are a case in point. It would be difﬁcult, at best, to write Java programs without them.  
Không có gì hữu ích và thỏa mãn bằng một API công khai được mô tả rõ ràng. Tài liệu java cho thư viện Java chuẩn là một trường hợp điển hình. Tốt nhất, sẽ rất khó để viết các chương trình Java mà không có chúng.

If you are writing a public API, then you should certainly write good javadocs for it. But keep in mind the rest of the advice in this chapter. Javadocs can be just as misleading, nonlocal, and dishonest as any other kind of comment.  
Nếu bạn đang viết một API công khai, thì chắc chắn bạn nên viết javadocs tốt cho nó. Nhưng hãy ghi nhớ phần còn lại của lời khuyên trong chương này. Javadocs có thể gây hiểu lầm, không cục bộ và không trung thực như bất kỳ loại nhận xét nào khác.

[**Bad Comments**](#_page_165_0)  
bình luận xấu

Most comments fall into this category. Usually they are crutches or excuses for poor code or justiﬁcations for insufﬁcient decisions, amounting to little more than the programmer talking to himself.  
Hầu hết các bình luận rơi vào loại này. Thông thường, chúng là những cái nạng hoặc lời bào chữa cho mã kém hoặc biện minh cho các quyết định không đầy đủ, chẳng khác gì việc lập trình viên tự nói chuyện với chính mình.

[**Mumbling**](#_page_165_0)  
lầm bầm

Plopping in a comment just because you feel you should or because the process requires it, is a hack. If you decide to write a comment, then spend the time necessary to make sure it is the best comment you can write.  
Viết một bình luận chỉ vì bạn cảm thấy nên làm hoặc vì quy trình yêu cầu điều đó, là một hành vi gian lận. Nếu bạn quyết định viết bình luận, thì hãy dành thời gian cần thiết để đảm bảo rằng đó là bình luận tốt nhất mà bạn có thể viết.

<https://www.facebook.com/groups/phattrienlaptrinh>

60 **Chapter 4: Comments**  
60 Chương 4: Bình Luận

Here, for example, is a case I found in FitNesse, where a comment might indeed have been useful. But the author was in a hurry or just not paying much attention. His mum-bling left behind an enigma:  
Ví dụ, đây là một trường hợp tôi tìm thấy trong FitNesse, trong đó một nhận xét thực sự có thể hữu ích. Nhưng tác giả đã vội vàng hoặc chỉ là không chú ý nhiều. Tiếng lầm bầm của anh ấy để lại một điều bí ẩn:

public void loadProperties() {

try {

String propertiesPath = propertiesLocation + "/" + PROPERTIES\_FILE; FileInputStream propertiesStream = new FileInputStream(propertiesPath); loadedProperties.load(propertiesStream);

}

catch(IOException e) {

**// No properties files means all defaults are loaded** }

}

What does that comment in the catchblock mean? Clearly it meant something to the author, but the meaning does not come through all that well. Apparently, if we get an IOException, it means that there was no properties ﬁle; and in that case all the defaults are loaded. But who loads all the defaults? Were they loaded before the call to loadProperties.load? Or did loadProperties.loadcatch the exception, load the defaults, and then pass the exception on for us to ignore? Or did loadProperties.loadload all the defaults before attempting to load the ﬁle? Was the author trying to comfort himself about the fact that he was leaving the catch block empty? Or—and this is the scary possibility— was the author trying to tell himself to come back here later and write the code that would load the defaults?  
Nhận xét đó trong khối bắt có nghĩa là gì? Rõ ràng nó có ý nghĩa gì đó đối với tác giả, nhưng ý nghĩa không hoàn toàn rõ ràng. Rõ ràng, nếu chúng tôi nhận được một IOException, điều đó có nghĩa là không có tệp thuộc tính nào; và trong trường hợp đó, tất cả các giá trị mặc định đều được tải. Nhưng ai tải tất cả các giá trị mặc định? Chúng có được tải trước lệnh gọi tới loadProperties.load không? Hay loadProperties.loadcatch ngoại lệ, tải các giá trị mặc định, rồi chuyển ngoại lệ cho chúng tôi bỏ qua? Hay loadProperties.load đã tải tất cả các giá trị mặc định trước khi thử tải tệp? Có phải tác giả đang cố tự an ủi mình về việc anh ta đang để trống khối bắt? Hoặc—và đây là một khả năng đáng sợ—có phải tác giả đang cố tự nhủ rằng hãy quay lại đây sau và viết đoạn mã sẽ tải các giá trị mặc định?

Our only recourse is to examine the code in other parts of the system to ﬁnd out what’s going on. Any comment that forces you to look in another module for the meaning of that comment has failed to communicate to you and is not worth the bits it consumes.  
Cách duy nhất của chúng tôi là kiểm tra mã trong các phần khác của hệ thống để tìm hiểu điều gì đang xảy ra. Bất kỳ nhận xét nào buộc bạn phải tìm kiếm trong một mô-đun khác để biết ý nghĩa của nhận xét đó đều không truyền đạt được cho bạn và không đáng để tiêu tốn bit.

[**Redundant Comments**](#_page_165_0)  
Nhận xét dự phòng

Listing 4-1 shows a simple function with a header comment that is completely redundant. The comment probably takes longer to read than the code itself.  
Liệt kê 4-1 cho thấy một chức năng đơn giản với chú thích tiêu đề hoàn toàn dư thừa. Nhận xét có thể mất nhiều thời gian hơn để đọc so với mã.

**Listing 4-1 waitForClose  
Liệt kê 4-1 waitForClose**

**// Utility method that returns when this.closed is true. Throws an exception // if the timeout is reached.**

public synchronized void waitForClose(final long timeoutMillis) throws Exception

{

if(!closed) {

wait(timeoutMillis);

if(!closed)

throw new Exception("MockResponseSender could not be closed"); }

}

<https://www.facebook.com/groups/phattrienlaptrinh>

**Bad Comments** 61  
Bình Luận Xấu 61

What purpose does this comment serve? It’s certainly not more informative than the code. It does not justify the code, or provide intent or rationale. It is not easier to read than the code. Indeed, it is less precise than the code and entices the reader to accept that lack of precision in lieu of true understanding. It is rather like a gladhanding used-car salesman assuring you that you don’t need to look under the hood.  
Bình luận này phục vụ mục đích gì? Nó chắc chắn không nhiều thông tin hơn mã. Nó không biện minh cho mã, hoặc cung cấp mục đích hoặc lý do. Nó không dễ đọc hơn mã. Thật vậy, nó kém chính xác hơn mã và lôi kéo người đọc chấp nhận sự thiếu chính xác đó thay cho sự hiểu biết thực sự. Nó giống như một người bán xe đã qua sử dụng vui vẻ đảm bảo với bạn rằng bạn không cần phải nhìn dưới mui xe.

Now consider the legion of useless and redundant javadocs in Listing 4-2 taken from Tomcat. These comments serve only to clutter and obscure the code. They serve no docu-mentary purpose at all. To make matters worse, I only showed you the ﬁrst few. There are many more in this module.  
Bây giờ hãy xem xét vô số javadoc vô dụng và dư thừa trong Liệt kê 4-2 được lấy từ Tomcat. Những nhận xét này chỉ phục vụ để làm lộn xộn và che khuất mã. Chúng không phục vụ mục đích tài liệu nào cả. Tệ hơn nữa, tôi chỉ cho bạn xem vài bức đầu tiên. Có nhiều hơn nữa trong mô-đun này.

**Listing 4-2**

**ContainerBase.java (Tomcat)  
ContainerBase.java (Tomcat)**

public abstract class ContainerBase implements Container, Lifecycle, Pipeline, MBeanRegistration, Serializable {

/\*\*

\* The processor delay for this component. \*/

protected int backgroundProcessorDelay = -1;

/\*\*

\* The lifecycle event support for this component. \*/

protected LifecycleSupport lifecycle = new LifecycleSupport(this);

/\*\*

\* The container event listeners for this Container. \*/

protected ArrayList listeners = new ArrayList();

/\*\*

\* The Loader implementation with which this Container is \* associated.

\*/

protected Loader loader = null;

/\*\*

\* The Logger implementation with which this Container is

\* associated. \*/

protected Log logger = null;

/\*\*

\* Associated logger name. \*/

protected String logName = null;

<https://www.facebook.com/groups/phattrienlaptrinh>

62 **Chapter 4: Comments**  
62 Chương 4: Bình Luận

**Listing 4-2 (continued) ContainerBase.java (Tomcat)  
Liệt kê 4-2 (tiếp theo) ContainerBase.java (Tomcat)**

/\*\*

\* The Manager implementation with which this Container is \* associated.

\*/

protected Manager manager = null;

/\*\*

\* The cluster with which this Container is associated. \*/

protected Cluster cluster = null;

/\*\*

\* The human-readable name of this Container. \*/

protected String name = null;

/\*\*

\* The parent Container to which this Container is a child. \*/

protected Container parent = null;

/\*\*

\* The parent class loader to be configured when we install a \* Loader.

\*/

protected ClassLoader parentClassLoader = null;

/\*\*

\* The Pipeline object with which this Container is \* associated.

\*/

protected Pipeline pipeline = new StandardPipeline(this);

/\*\*

\* The Realm with which this Container is associated. \*/

protected Realm realm = null;

/\*\*

\* The resources DirContext object with which this Container \* is associated.

\*/

protected DirContext resources = null;

<https://www.facebook.com/groups/phattrienlaptrinh>

**Bad Comments** 63  
Bình Luận Xấu 63

[**Misleading Comments**](#_page_165_0)  
Nhận xét gây hiểu lầm

Sometimes, with all the best intentions, a programmer makes a statement in his comments that isn’t precise enough to be accurate. Consider for another moment the badly redundant but also subtly misleading comment we saw in Listing 4-1.  
Đôi khi, với tất cả những ý định tốt nhất, một lập trình viên đưa ra một tuyên bố trong nhận xét của mình không đủ chính xác để chính xác. Hãy xem xét một khoảnh khắc khác về nhận xét thừa thãi nhưng cũng gây hiểu lầm một cách tinh vi mà chúng ta đã thấy trong Liệt kê 4-1.

Did you discover how the comment was misleading? The method does not return *when* this.closedbecomes true. It returns *if* this.closedis true; otherwise, it waits for a blind time-out and then throws an exception *if* this.closed is still not true.  
Bạn có phát hiện ra cách bình luận gây hiểu nhầm không? Phương thức này không trả về khi this. closedbecome true. Nó trả về nếu this. closedis true; mặt khác, nó đợi hết thời gian mù và sau đó đưa ra một ngoại lệ nếu điều này.đóng vẫn không đúng.

This subtle bit of misinformation, couched in a comment that is harder to read than the body of the code, could cause another programmer to blithely call this function in the expectation that it will return as soon as this.closedbecomes true. That poor programmer would then ﬁnd himself in a debugging session trying to ﬁgure out why his code executed so slowly.  
Một chút thông tin sai lệch tinh vi này, được đặt trong một nhận xét khó đọc hơn phần thân của mã, có thể khiến một lập trình viên khác vô tình gọi hàm này với mong muốn rằng nó sẽ trả về ngay khi điều này.đóng cửa trở thành sự thật. Sau đó, lập trình viên kém cỏi đó sẽ thấy mình đang trong một phiên gỡ lỗi để cố gắng tìm ra lý do tại sao mã của anh ta thực thi quá chậm.

[**Mandated Comments**](#_page_165_0)  
Nhận xét bắt buộc

It is just plain silly to have a rule that says that every function must have a javadoc, or every variable must have a comment. Comments like this just clutter up the code, propa-gate lies, and lend to general confusion and disorganization.  
Thật ngớ ngẩn khi có một quy tắc nói rằng mọi chức năng phải có javadoc hoặc mọi biến phải có nhận xét. Những nhận xét như thế này chỉ làm lộn xộn mã, nói dối về cổng tuyên truyền và dẫn đến sự nhầm lẫn và vô tổ chức chung.

For example, required javadocs for every function lead to abominations such as List-ing 4-3. This clutter adds nothing and serves only to obfuscate the code and create the potential for lies and misdirection.  
Ví dụ: javadocs bắt buộc đối với mọi chức năng dẫn đến những điều ghê tởm như List-ing 4-3. Sự lộn xộn này không bổ sung gì và chỉ phục vụ để làm xáo trộn mã và tạo ra khả năng dối trá và định hướng sai.

**Listing 4-3**

/\*\* \*

\* @param title The title of the CD \* @param author The author of the CD

\* @param tracks The number of tracks on the CD

\* @param durationInMinutes The duration of the CD in minutes \*/

public void addCD(String title, String author,

int tracks, int durationInMinutes) { CD cd = new CD();

cd.title = title; cd.author = author; cd.tracks = tracks; cd.duration = duration; cdList.add(cd);

}

[**Journal Comments**](#_page_165_0)  
Nhận xét tạp chí

Sometimes people add a comment to the start of a module every time they edit it. These comments accumulate as a kind of journal, or log, of every change that has ever been made. I have seen some modules with dozens of pages of these run-on journal entries.  
Đôi khi mọi người thêm nhận xét vào đầu mô-đun mỗi khi họ chỉnh sửa mô-đun đó. Những nhận xét này tích lũy như một loại nhật ký hoặc nhật ký về mọi thay đổi đã từng được thực hiện. Tôi đã thấy một số mô-đun có hàng chục trang gồm các mục nhật ký đang chạy này.

<https://www.facebook.com/groups/phattrienlaptrinh>

64 **Chapter 4: Comments**  
64 Chương 4: Bình Luận

\* Changes (from 11-Oct-2001) \* --------------------------

\* 11-Oct-2001 : Re-organised the class and moved it to new package \* com.jrefinery.date (DG);

\* 05-Nov-2001 : Added a getDescription() method, and eliminated NotableDate \* class (DG);

\* 12-Nov-2001 : IBD requires setDescription() method, now that NotableDate \* class is gone (DG); Changed getPreviousDayOfWeek(),

\* getFollowingDayOfWeek() and getNearestDayOfWeek() to correct \* bugs (DG);

\* 05-Dec-2001 : Fixed bug in SpreadsheetDate class (DG);

\* 29-May-2002 : Moved the month constants into a separate interface \* (MonthConstants) (DG);

\* 27-Aug-2002 : Fixed bug in addMonths() method, thanks to N???levka Petr (DG); \* 03-Oct-2002 : Fixed errors reported by Checkstyle (DG);

\* 13-Mar-2003 : Implemented Serializable (DG);

\* 29-May-2003 : Fixed bug in addMonths method (DG);

\* 04-Sep-2003 : Implemented Comparable. Updated the isInRange javadocs (DG); \* 05-Jan-2005 : Fixed bug in addYears() method (1096282) (DG);

Long ago there was a good reason to create and maintain these log entries at the start of every module. We didn’t have source code control systems that did it for us. Nowadays, however, these long journals are just more clutter to obfuscate the module. They should be completely removed.  
Từ lâu, đã có lý do chính đáng để tạo và duy trì các mục nhật ký này khi bắt đầu mỗi mô-đun. Chúng tôi không có hệ thống kiểm soát mã nguồn làm việc đó cho chúng tôi. Tuy nhiên, ngày nay, những nhật ký dài này chỉ lộn xộn hơn để làm rối mô-đun. Chúng nên được loại bỏ hoàn toàn.

[**Noise Comments**](#_page_165_0)  
Nhận xét về tiếng ồn

Sometimes you see comments that are nothing but noise. They restate the obvious and provide no new information.  
Đôi khi bạn thấy những bình luận không có gì ngoài tiếng ồn. Họ trình bày lại điều hiển nhiên và không cung cấp thông tin mới.

/\*\*

\* Default constructor. \*/

protected AnnualDateRule() { }

No, *really?* Or how about this:  
Không, thực sự? Hoặc làm thế nào về điều này:

/\*\* The day of the month. \*/ private int dayOfMonth;

And then there’s this paragon of redundancy:  
Và sau đó là mô hình dư thừa này:

/\*\*

\* Returns the day of the month. \*

\* @return the day of the month. \*/

public int getDayOfMonth() { return dayOfMonth;

}

<https://www.facebook.com/groups/phattrienlaptrinh>

**Bad Comments** 65  
Bình Luận Xấu 65

These comments are so noisy that we learn to ignore them. As we read through code, our eyes simply skip over them. Eventually the comments begin to lie as the code around them changes.  
Những bình luận này ồn ào đến mức chúng tôi học cách phớt lờ chúng. Khi chúng ta đọc qua mã, mắt của chúng ta chỉ đơn giản là bỏ qua chúng. Cuối cùng, các nhận xét bắt đầu dối trá khi mã xung quanh chúng thay đổi.

The ﬁrst comment in Listing 4-4 seems appropriate.2 It explains why the catchblock is being ignored. But the second comment is pure noise. Apparently the programmer was just so frustrated with writing try/catch blocks in this function that he needed to vent.  
Nhận xét đầu tiên trong Liệt kê 4-4 có vẻ phù hợp.2 Nó giải thích lý do tại sao khối bắt được bỏ qua. Nhưng nhận xét thứ hai là tiếng ồn thuần túy. Rõ ràng lập trình viên đã quá chán nản với việc viết các khối try/catch trong chức năng này nên anh ta cần trút giận.

**Listing 4-4 startSending  
Liệt kê 4-4 startSending**

private void startSending() {

try {

doSending(); }

catch(SocketException e) {

**// normal. someone stopped the request.** }

catch(Exception e) {

try {

response.add(ErrorResponder.makeExceptionString(e)); response.closeAll();

}

catch(Exception e1) {

**//Give me a break!** }

} }

Rather than venting in a worthless and noisy comment, the programmer should have recognized that his frustration could be resolved by improving the structure of his code. He should have redirected his energy to extracting that last try/catchblock into a separate function, as shown in Listing 4-5.  
Thay vì trút một bình luận ồn ào và vô giá trị, lập trình viên nên nhận ra rằng sự thất vọng của anh ta có thể được giải quyết bằng cách cải thiện cấu trúc mã của anh ta. Lẽ ra anh ta nên chuyển hướng năng lượng của mình sang việc giải nén try/catchblock cuối cùng đó thành một hàm riêng biệt, như trong Liệt kê 4-5.

**Listing 4-5**

**startSending (refactored)  
startSending (tái cấu trúc)**

private void startSending() {

try {

doSending(); }

2. The current trend for IDEs to check spelling in comments will be a balm for those of us who read a lot of code.  
2. Xu hướng hiện tại đối với các IDE kiểm tra chính tả trong các bình luận sẽ là một liều thuốc an thần cho những ai đọc nhiều mã như chúng ta.

<https://www.facebook.com/groups/phattrienlaptrinh>

66 **Chapter 4: Comments**  
66 Chương 4: Bình Luận

**Listing 4-5 (continued) startSending (refactored)  
Liệt kê 4-5 (tiếp theo) startSending (đã tái cấu trúc)**

catch(SocketException e) {

// normal. someone stopped the request. }

catch(Exception e) {

addExceptionAndCloseResponse(e); }

}

private void addExceptionAndCloseResponse(Exception e) {

try {

response.add(ErrorResponder.makeExceptionString(e)); response.closeAll();

}

catch(Exception e1) {

} }

Replace the temptation to create noise with the determination to clean your code. You’ll ﬁnd it makes you a better and happier programmer.  
Thay thế sự cám dỗ để tạo ra tiếng ồn bằng quyết tâm làm sạch mã của bạn. Bạn sẽ thấy nó giúp bạn trở thành một lập trình viên tốt hơn và hạnh phúc hơn.

[**Scary Noise**](#_page_165_0)  
tiếng ồn đáng sợ

Javadocs can also be noisy. What purpose do the following Javadocs (from a well-known open-source library) serve? Answer: nothing. They are just redundant noisy comments written out of some misplaced desire to provide documentation.  
Javadocs cũng có thể ồn ào. Các Javadocs sau (từ một thư viện mã nguồn mở nổi tiếng) phục vụ cho mục đích gì? Trả lời: không có gì. Chúng chỉ là những bình luận ồn ào dư thừa được viết ra từ một số mong muốn cung cấp tài liệu không đúng chỗ.

/\*\* The name. \*/ private String name;

/\*\* The version. \*/ private String version;

/\*\* The licenceName. \*/ private String licenceName;

/\*\* The version. \*/ private String info;

Read these comments again more carefully. Do you see the cut-paste error? If authors aren’t paying attention when comments are written (or pasted), why should readers be expected to proﬁt from them?  
Hãy đọc lại những bình luận này một cách cẩn thận hơn. Bạn có thấy lỗi cắt dán không? Nếu các tác giả không chú ý khi các bình luận được viết (hoặc dán), thì tại sao người đọc phải thu lợi từ chúng?

<https://www.facebook.com/groups/phattrienlaptrinh>

**Bad Comments** 67  
Bình Luận Xấu 67

[**Don’t Use a Comment When You Can Use a Function or a Variable**](#_page_165_0)  
Không sử dụng nhận xét khi bạn có thể sử dụng hàm hoặc biến

Consider the following stretch of code:  
Hãy xem xét đoạn mã sau:

// does the module from the global list <mod> depend on the // subsystem we are part of?

if (smodule.getDependSubsystems().contains(subSysMod.getSubSystem()))

This could be rephrased without the comment as  
Điều này có thể được viết lại mà không cần bình luận như

ArrayList moduleDependees = smodule.getDependSubsystems(); String ourSubSystem = subSysMod.getSubSystem();

if (moduleDependees.contains(ourSubSystem))

The author of the original code may have written the comment ﬁrst (unlikely) and then written the code to fulﬁll the comment. However, the author should then have refactored the code, as I did, so that the comment could be removed.  
Tác giả của đoạn mã gốc có thể đã viết chú thích trước (không chắc) và sau đó viết mã để hoàn thành chú thích. Tuy nhiên, tác giả sau đó nên cấu trúc lại mã, như tôi đã làm, để nhận xét có thể bị xóa.

[**Position Markers**](#_page_165_0)  
Đánh dấu vị trí

Sometimes programmers like to mark a particular position in a source ﬁle. For example, I recently found this in a program I was looking through:  
Đôi khi các lập trình viên muốn đánh dấu một vị trí cụ thể trong tệp nguồn. Ví dụ: gần đây tôi đã tìm thấy điều này trong một chương trình mà tôi đang xem qua:

// Actions /

There are rare times when it makes sense to gather certain functions together beneath a banner like this. But in general they are clutter that should be eliminated—especially the noisy train of slashes at the end.  
Hiếm khi có ý nghĩa khi tập hợp các chức năng nhất định lại với nhau bên dưới một biểu ngữ như thế này. Nhưng nói chung, chúng lộn xộn nên được loại bỏ—đặc biệt là chuỗi gạch chéo ồn ào ở cuối.

Think of it this way. A banner is startling and obvious if you don’t see banners very often. So use them very sparingly, and only when the beneﬁt is signiﬁcant. If you overuse banners, they’ll fall into the background noise and be ignored.  
Nghĩ theo cách này. Một biểu ngữ gây sửng sốt và rõ ràng nếu bạn không thường xuyên nhìn thấy các biểu ngữ. Vì vậy, hãy sử dụng chúng thật tiết kiệm và chỉ khi lợi ích mang lại là đáng kể. Nếu bạn lạm dụng các biểu ngữ, chúng sẽ rơi vào tiếng ồn xung quanh và bị bỏ qua.

[**Closing Brace Comments**](#_page_165_0)  
Nhận xét cú đúp đóng

Sometimes programmers will put special comments on closing braces, as in Listing 4-6. Although this might make sense for long functions with deeply nested structures, it serves only to clutter the kind of small and encapsulated functions that we prefer. So if you ﬁnd yourself wanting to mark your closing braces, try to shorten your functions instead.  
Đôi khi các lập trình viên sẽ đưa ra các nhận xét đặc biệt về việc đóng dấu ngoặc nhọn, như trong Liệt kê 4-6. Mặc dù điều này có thể hợp lý đối với các hàm dài có cấu trúc lồng sâu, nhưng nó chỉ dùng để làm lộn xộn loại hàm nhỏ và được đóng gói mà chúng ta thích. Vì vậy, nếu bạn thấy mình muốn đánh dấu các dấu ngoặc nhọn đóng của mình, thay vào đó hãy cố gắng rút ngắn các hàm của bạn.

**Listing 4-6 wc.java**  
Liệt kê 4-6 wc.java

public class wc {

public static void main(String[] args) {

BufferedReader in = new BufferedReader(new InputStreamReader(System.in)); String line;

int lineCount = 0; int charCount = 0; int wordCount = 0; try {

<https://www.facebook.com/groups/phattrienlaptrinh>

68 **Chapter 4: Comments**  
68 Chương 4: Bình Luận

**Listing 4-6 (continued) wc.java  
Liệt kê 4-6 (tiếp theo) wc.java**

while ((line = in.readLine()) != null) { lineCount++;

charCount += line.length();

String words[] = line.split("\\W"); wordCount += words.length;

**} //while**

System.out.println("wordCount = " + wordCount); System.out.println("lineCount = " + lineCount); System.out.println("charCount = " + charCount);

**} // try**

catch (IOException e) { System.err.println("Error:" + e.getMessage());

**} //catch** } //main

}

[**Attributions and Bylines**](#_page_165_0)  
Thuộc tính và Bylines

/\* Added by Rick \*/

Source code control systems are very good at remembering who added what, when. There is no need to pollute the code with little bylines. You might think that such com-ments would be useful in order to help others know who to talk to about the code. But the reality is that they tend to stay around for years and years, getting less and less accurate and relevant.  
Các hệ thống kiểm soát mã nguồn rất giỏi trong việc ghi nhớ ai đã thêm cái gì, khi nào. Không cần phải làm ô nhiễm mã với các dòng nhỏ. Bạn có thể nghĩ rằng những nhận xét như vậy sẽ hữu ích để giúp những người khác biết nên nói chuyện với ai về mã. Nhưng thực tế là chúng có xu hướng tồn tại hết năm này qua năm khác, ngày càng ít chính xác và ít liên quan hơn.

Again, the source code control system is a better place for this kind of information.  
Một lần nữa, hệ thống kiểm soát mã nguồn là nơi tốt hơn cho loại thông tin này.

[**Commented-Out Code**](#_page_237_0)  
Mã nhận xét

Few practices are as odious as commenting-out code. Don’t do this!  
Một số thực hành cũng đáng ghét như mã bình luận. Đừng làm điều này!

InputStreamResponse response = new InputStreamResponse(); response.setBody(formatter.getResultStream(), formatter.getByteCount());

**// InputStream resultsStream = formatter.getResultStream(); // StreamReader reader = new StreamReader(resultsStream);**

**// response.setContent(reader.read(formatter.getByteCount()));**

Others who see that commented-out code won’t have the courage to delete it. They’ll think it is there for a reason and is too important to delete. So commented-out code gathers like dregs at the bottom of a bad bottle of wine.  
Những người khác nhìn thấy mã nhận xét đó sẽ không đủ can đảm để xóa nó. Họ sẽ nghĩ rằng nó có lý do và quá quan trọng để xóa. Vì vậy, mã nhận xét tập hợp lại như cặn dưới đáy một chai rượu dở.

Consider this from apache commons:  
Hãy xem xét điều này từ apache commons:

this.bytePos = writeBytes(pngIdBytes, 0); **//hdrPos = bytePos;**

writeHeader(); writeResolution(); **//dataPos = bytePos;** if (writeImageData()) {

writeEnd();

this.pngBytes = resizeByteArray(this.pngBytes, this.maxPos); }

<https://www.facebook.com/groups/phattrienlaptrinh>

**Bad Comments** 69  
Bình Luận Xấu 69

else {

this.pngBytes = null; }

return this.pngBytes;

Why are those two lines of code commented? Are they important? Were they left as reminders for some imminent change? Or are they just cruft that someone commented-out years ago and has simply not bothered to clean up.  
Tại sao hai dòng mã đó được nhận xét? Chúng có quan trọng không? Có phải họ để lại như lời nhắc nhở cho một số thay đổi sắp xảy ra? Hay chúng chỉ là thứ rác rưởi mà ai đó đã nhận xét từ nhiều năm trước và đơn giản là không thèm dọn dẹp.

There was a time, back in the sixties, when commenting-out code might have been useful. But we’ve had good source code control systems for a very long time now. Those systems will remember the code for us. We don’t have to comment it out any more. Just delete the code. We won’t lose it. Promise.  
Đã có lúc, vào những năm sáu mươi, khi mã nhận xét có thể hữu ích. Nhưng chúng ta đã có những hệ thống kiểm soát mã nguồn tốt từ rất lâu rồi. Những hệ thống đó sẽ ghi nhớ mã cho chúng ta. Chúng ta không cần phải bình luận nó ra nữa. Chỉ cần xóa mã. Chúng tôi sẽ không để mất nó. Hứa.

[**HTML Comments**](#_page_237_0)  
Nhận xét HTML

HTML in source code comments is an abomination, as you can tell by reading the code below. It makes the comments hard to read in the one place where they should be easy to read—the editor/IDE. If comments are going to be extracted by some tool (like Javadoc) to appear in a Web page, then it should be the responsibility of that tool, and not the program-mer, to adorn the comments with appropriate HTML.  
HTML trong các nhận xét về mã nguồn là một điều đáng ghê tởm, như bạn có thể biết bằng cách đọc đoạn mã bên dưới. Nó làm cho các bình luận trở nên khó đọc ở một nơi mà lẽ ra chúng phải dễ đọc—trình soạn thảo/IDE. Nếu các bình luận sẽ được trích xuất bởi một công cụ nào đó (như Javadoc) để xuất hiện trong một trang Web, thì trách nhiệm của công cụ đó, chứ không phải của người lập trình, là tô điểm cho các bình luận bằng HTML thích hợp.

/\*\*

\* Task to run fit tests.

\* This task runs fitnesse tests and publishes the results. \* <p/>

\* <pre> \* Usage:

\* &lt;taskdef name=&quot;execute-fitnesse-tests&quot;

\* classname=&quot;fitnesse.ant.ExecuteFitnesseTestsTask&quot; \* classpathref=&quot;classpath&quot; /&gt;

\* OR

\* &lt;taskdef classpathref=&quot;classpath&quot;

\* resource=&quot;tasks.properties&quot; /&gt; \* <p/>

\* &lt;execute-fitnesse-tests

\* suitepage=&quot;FitNesse.SuiteAcceptanceTests&quot; \* fitnesseport=&quot;8082&quot;

\* resultsdir=&quot;${results.dir}&quot;

\* resultshtmlpage=&quot;fit-results.html&quot; \* classpathref=&quot;classpath&quot; /&gt;

\* </pre> \*/

[**Nonlocal Information**](#_page_237_0)  
Thông tin phi cục bộ

If you must write a comment, then make sure it describes the code it appears near. Don’t offer systemwide information in the context of a local comment. Consider, for example, the javadoc comment below. Aside from the fact that it is horribly redundant, it also offers information about the default port. And yet the function has absolutely no control over what that default is. The comment is not describing the function, but some other, far dis-tant part of the system. Of course there is no guarantee that this comment will be changed when the code containing the default is changed.  
Nếu bạn phải viết bình luận, thì hãy đảm bảo rằng nó mô tả mã mà nó xuất hiện gần đó. Không cung cấp thông tin toàn hệ thống trong bối cảnh nhận xét cục bộ. Ví dụ, hãy xem xét nhận xét javadoc bên dưới. Bên cạnh thực tế là nó quá dư thừa, nó còn cung cấp thông tin về cổng mặc định. Tuy nhiên, chức năng này hoàn toàn không kiểm soát được mặc định đó là gì. Nhận xét không mô tả chức năng, mà là một phần khác, rất xa của hệ thống. Tất nhiên không có gì đảm bảo rằng nhận xét này sẽ bị thay đổi khi mã chứa mặc định bị thay đổi.

<https://www.facebook.com/groups/phattrienlaptrinh>

70 **Chapter 4: Comments**  
70 Chương 4: Bình Luận

/\*\*

\* Port on which fitnesse would run. Defaults to <b>8082</b>. \*

\* @param fitnessePort \*/

public void setFitnessePort(int fitnessePort) {

this.fitnessePort = fitnessePort; }

[**Too Much Information**](#_page_237_0)  
Quá nhiều thông tin

Don’t put interesting historical discussions or irrelevant descriptions of details into your comments. The comment below was extracted from a module designed to test that a func-tion could encode and decode base64. Other than the RFC number, someone reading this code has no need for the arcane information contained in the comment.  
Đừng đưa các cuộc thảo luận lịch sử thú vị hoặc mô tả chi tiết không liên quan vào nhận xét của bạn. Nhận xét bên dưới được trích xuất từ một mô-đun được thiết kế để kiểm tra xem chức năng có thể mã hóa và giải mã base64 hay không. Ngoài số RFC, ai đó đọc mã này không cần thông tin phức tạp có trong nhận xét.

/\*

RFC 2045 - Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies

section 6.8. Base64 Content-Transfer-Encoding

The encoding process represents 24-bit groups of input bits as output strings of 4 encoded characters. Proceeding from left to right, a

24-bit input group is formed by concatenating 3 8-bit input groups. These 24 bits are then treated as 4 concatenated 6-bit groups, each of which is translated into a single digit in the base64 alphabet. When encoding a bit stream via the base64 encoding, the bit stream must be presumed to be ordered with the most-significant-bit first. That is, the first bit in the stream will be the high-order bit in the first 8-bit byte, and the eighth bit will be the low-order bit in the first 8-bit byte, and so on.

\*/

[**Inobvious Connection**](#_page_237_0)  
kết nối không rõ ràng

The connection between a comment and the code it describes should be obvious. If you are going to the trouble to write a comment, then at least you’d like the reader to be able to look at the comment and the code and understand what the comment is talking about.  
Mối liên hệ giữa một nhận xét và mã mà nó mô tả phải rõ ràng. Nếu bạn gặp khó khăn khi viết bình luận, thì ít nhất bạn muốn người đọc có thể xem bình luận và mã và hiểu bình luận đó đang nói về điều gì.

Consider, for example, this comment drawn from apache commons:  
Ví dụ, hãy xem xét nhận xét này được rút ra từ apache commons:

/\*

\* start with an array that is big enough to hold all the pixels \* (plus filter bytes), and an extra 200 bytes for header info \*/

this.pngBytes = new byte[((this.width + 1) \* this.height \* 3) + 200];

What is a ﬁlter byte? Does it relate to the +1? Or to the \*3? Both? Is a pixel a byte? Why 200? The purpose of a comment is to explain code that does not explain itself. It is a pity when a comment needs its own explanation.  
Byte bộ lọc là gì? Nó có liên quan đến +1 không? Hay đến \*3? Cả hai? Là một pixel một byte? Tại sao lại là 200? Mục đích của một bình luận là để giải thích mã không giải thích chính nó. Thật đáng tiếc khi một bình luận cần lời giải thích riêng.

[**Function Headers**](#_page_237_0)  
Tiêu đề chức năng

Short functions don’t need much description. A well-chosen name for a small function that does one thing is usually better than a comment header.  
Các chức năng ngắn không cần mô tả nhiều. Một cái tên được lựa chọn tốt cho một chức năng nhỏ thực hiện một việc thường tốt hơn một tiêu đề bình luận.

<https://www.facebook.com/groups/phattrienlaptrinh>

**Bad Comments** 71  
Bình Luận Xấu 71

[**Javadocs in Nonpublic Code**](#_page_237_0)  
Javadocs trong mã không công khai

As useful as javadocs are for public APIs, they are anathema to code that is not intended for public consumption. Generating javadoc pages for the classes and functions inside a system is not generally useful, and the extra formality of the javadoc comments amounts to little more than cruft and distraction.  
Cũng hữu ích như javadocs dành cho các API công khai, chúng không phù hợp với mã không dành cho mục đích sử dụng chung. Việc tạo các trang javadoc cho các lớp và chức năng bên trong một hệ thống nói chung không hữu ích và hình thức bổ sung của các nhận xét javadoc chỉ hơn một chút so với hành trình và sự phân tâm.

[**Example**](#_page_237_0)  
Ví dụ

I wrote the module in Listing 4-7 for the ﬁrst *XP Immersion*. It was intended to be an example of bad coding and commenting style. Kent Beck then refactored this code into a much more pleasant form in front of several dozen enthusiastic students. Later I adapted the example for my book *Agile Software Development, Principles, Patterns, and Practices* and the ﬁrst of my *Craftsman* articles published in *Software Development* magazine.  
Tôi đã viết mô-đun trong Liệt kê 4-7 cho lần ngâm XP đầu tiên. Nó được dự định là một ví dụ về phong cách viết mã và bình luận tồi. Kent Beck sau đó đã cấu trúc lại đoạn mã này thành một dạng dễ chịu hơn nhiều trước sự chứng kiến của hàng chục sinh viên nhiệt tình. Sau đó, tôi đã điều chỉnh ví dụ này cho cuốn sách Agile Software Development, Principles, Patterns, and Practices và bài báo đầu tiên về Craftsman của tôi đăng trên tạp chí Software Development.

What I ﬁnd fascinating about this module is that there was a time when many of us would have considered it “well documented.” Now we see it as a small mess. See how many different comment problems you can ﬁnd.  
Điều tôi thấy thú vị về mô-đun này là đã có lúc nhiều người trong chúng ta coi nó là “tài liệu tốt”. Bây giờ chúng tôi thấy nó như một mớ hỗn độn nhỏ. Xem có bao nhiêu vấn đề bình luận khác nhau mà bạn có thể tìm thấy.

**Listing 4-7 GeneratePrimes.java**  
Liệt kê 4-7 GeneratePrimes.java

/\*\*

\* This class Generates prime numbers up to a user specified \* maximum. The algorithm used is the Sieve of Eratosthenes. \* <p>

\* Eratosthenes of Cyrene, b. c. 276 BC, Cyrene, Libya --\* d. c. 194, Alexandria. The first man to calculate the \* circumference of the Earth. Also known for working on

\* calendars with leap years and ran the library at Alexandria. \* <p>

\* The algorithm is quite simple. Given an array of integers \* starting at 2. Cross out all multiples of 2. Find the next \* uncrossed integer, and cross out all of its multiples.

\* Repeat untilyou have passed the square root of the maximum \* value.

\*

\* @author Alphonse

\* @version 13 Feb 2002 atp \*/

import java.util.\*;

public class GeneratePrimes {

/\*\*

\* @param maxValue is the generation limit. \*/

public static int[] generatePrimes(int maxValue) {

if (maxValue >= 2) // the only valid case {

// declarations

int s = maxValue + 1; // size of array boolean[] f = new boolean[s];

int i;

<https://www.facebook.com/groups/phattrienlaptrinh>

72 **Chapter 4: Comments**  
72 Chương 4: Bình Luận

**Listing 4-7 (continued) GeneratePrimes.java  
Liệt kê 4-7 (tiếp theo) GeneratePrimes.java**

// initialize array to true. for (i = 0; i < s; i++)

f[i] = true;

// get rid of known non-primes f[0] = f[1] = false;

// sieve int j;

for (i = 2; i < Math.sqrt(s) + 1; i++) {

if (f[i]) // if i is uncrossed, cross its multiples. {

for (j = 2 \* i; j < s; j += i)

f[j] = false; // multiple is not prime }

}

// how many primes are there? int count = 0;

for (i = 0; i < s; i++) {

if (f[i])

count++; // bump count. }

int[] primes = new int[count];

// move the primes into the result for (i = 0, j = 0; i < s; i++)

{

if (f[i]) // if prime primes[j++] = i;

}

return primes; // return the primes }

else // maxValue < 2

return new int[0]; // return null array if bad input. }

}

In Listing 4-8 you can see a refactored version of the same module. Note that the use of comments is signiﬁcantly restrained. There are just two comments in the whole module. Both comments are explanatory in nature.  
Trong Liệt kê 4-8, bạn có thể thấy một phiên bản tái cấu trúc của cùng một mô-đun. Lưu ý rằng việc sử dụng các bình luận bị hạn chế đáng kể. Chỉ có hai nhận xét trong toàn bộ mô-đun. Cả hai ý kiến ​​là giải thích trong tự nhiên.

**Listing 4-8**

**PrimeGenerator.java (refactored)  
PrimeGenerator.java (tái cấu trúc)**

/\*\*

\* This class Generates prime numbers up to a user specified \* maximum. The algorithm used is the Sieve of Eratosthenes. \* Given an array of integers starting at 2:

\* Find the first uncrossed integer, and cross out all its

<https://www.facebook.com/groups/phattrienlaptrinh>

**Bad Comments** 73  
Bình Luận Xấu 73

**Listing 4-8 (continued) PrimeGenerator.java (refactored)  
Liệt kê 4-8 (tiếp theo) PrimeGenerator.java (đã tái cấu trúc)**

\* multiples. Repeat until there are no more multiples \* in the array.

\*/

public class PrimeGenerator {

private static boolean[] crossedOut; private static int[] result;

public static int[] generatePrimes(int maxValue) {

if (maxValue < 2) return new int[0];

else {

uncrossIntegersUpTo(maxValue); crossOutMultiples(); putUncrossedIntegersIntoResult(); return result;

} }

private static void uncrossIntegersUpTo(int maxValue) {

crossedOut = new boolean[maxValue + 1]; for (int i = 2; i < crossedOut.length; i++)

crossedOut[i] = false; }

private static void crossOutMultiples() {

int limit = determineIterationLimit(); for (int i = 2; i <= limit; i++)

if (notCrossed(i)) crossOutMultiplesOf(i);

}

private static int determineIterationLimit() {

// Every multiple in the array has a prime factor that // is less than or equal to the root of the array size, // so we don't have to cross out multiples of numbers // larger than that root.

double iterationLimit = Math.sqrt(crossedOut.length); return (int) iterationLimit;

}

private static void crossOutMultiplesOf(int i) {

for (int multiple = 2\*i;

multiple < crossedOut.length; multiple += i)

crossedOut[multiple] = true; }

<https://www.facebook.com/groups/phattrienlaptrinh>

74 **Chapter 4: Comments**  
74 Chương 4: Bình Luận

**Listing 4-8 (continued) PrimeGenerator.java (refactored)**  
Liệt kê 4-8 (tiếp theo) PrimeGenerator.java (đã tái cấu trúc)

private static boolean notCrossed(int i) {

return crossedOut[i] == false; }

private static void putUncrossedIntegersIntoResult() {

result = new int[numberOfUncrossedIntegers()]; for (int j = 0, i = 2; i < crossedOut.length; i++)

if (notCrossed(i)) result[j++] = i;

}

private static int numberOfUncrossedIntegers() {

int count = 0;

for (int i = 2; i < crossedOut.length; i++) if (notCrossed(i))

count++;

return count; }

}

It is easy to argue that the ﬁrst comment is redundant because it reads very much like the generatePrimesfunction itself. Still, I think the comment serves to ease the reader into the algorithm, so I’m inclined to leave it.  
Thật dễ dàng để lập luận rằng bình luận đầu tiên là dư thừa bởi vì nó đọc rất giống với bản thân hàm generatePrimes. Tuy nhiên, tôi nghĩ rằng nhận xét này giúp người đọc dễ dàng tiếp cận với thuật toán, vì vậy tôi có xu hướng bỏ nó đi.

The second argument is almost certainly necessary. It explains the rationale behind the use of the square root as the loop limit. I could ﬁnd no simple variable name, nor any different coding structure that made this point clear. On the other hand, the use of the square root might be a conceit. Am I really saving that much time by limiting the iteration to the square root? Could the calculation of the square root take more time than I’m saving?  
Đối số thứ hai gần như chắc chắn là cần thiết. Nó giải thích lý do đằng sau việc sử dụng căn bậc hai làm giới hạn vòng lặp. Tôi không thể tìm thấy tên biến đơn giản nào, cũng như bất kỳ cấu trúc mã hóa nào khác làm rõ điểm này. Mặt khác, việc sử dụng căn bậc hai có thể là một sự tự phụ. Tôi có thực sự tiết kiệm được ngần ấy thời gian bằng cách giới hạn phép lặp đến căn bậc hai không? Việc tính căn bậc hai có thể mất nhiều thời gian hơn tôi tiết kiệm được không?

It’s worth thinking about. Using the square root as the iteration limit satisﬁes the old C and assembly language hacker in me, but I’m not convinced it’s worth the time and effort that everyone else will expend to understand it.  
Thật đáng để suy nghĩ. Việc sử dụng căn bậc hai làm giới hạn lặp lại thỏa mãn bản tính hacker ngôn ngữ hợp ngữ và C cũ trong tôi, nhưng tôi không tin rằng nó xứng đáng với thời gian và công sức mà những người khác sẽ bỏ ra để hiểu nó.

[**Bibliography**](#_page_237_0)  
Thư mục

**[KP78]:** Kernighan and Plaugher, *The Elements of Programming Style*, 2d. ed., McGraw-Hill, 1978.

<https://www.facebook.com/groups/phattrienlaptrinh>

[**5**](#_page_237_0)

[**Formatting**](#_page_237_0)  
định dạng

When people look under the hood, we want them to be impressed with the neatness, con-sistency, and attention to detail that they perceive. We want them to be struck by the orderliness. We want their eyebrows to rise as they scroll through the modules. We want them to perceive that professionals have been at work. If instead they see a scrambled mass of code that looks like it was written by a bevy of drunken sailors, then they are likely to conclude that the same inattention to detail pervades every other aspect of the project.  
Khi mọi người nhìn kỹ, chúng tôi muốn họ ấn tượng với sự gọn gàng, nhất quán và sự chú ý đến từng chi tiết mà họ cảm nhận được. Chúng tôi muốn họ bị ấn tượng bởi sự ngăn nắp. Chúng tôi muốn lông mày của họ nhướng lên khi họ cuộn qua các mô-đun. Chúng tôi muốn họ hiểu rằng các chuyên gia đã làm việc. Nếu thay vào đó, họ nhìn thấy một khối mã lộn xộn trông giống như được viết bởi một nhóm thủy thủ say xỉn, thì họ có khả năng kết luận rằng sự thiếu chú ý đến từng chi tiết cũng lan tràn khắp mọi khía cạnh khác của dự án.

75

<https://www.facebook.com/groups/phattrienlaptrinh>

76 **Chapter 5: Formatting**  
76 Chương 5: Định dạng

You should take care that your code is nicely formatted. You should choose a set of simple rules that govern the format of your code, and then you should consistently apply those rules. If you are working on a team, then the team should agree to a single set of formatting rules and all members should comply. It helps to have an automated tool that can apply those formatting rules for you.  
Bạn nên cẩn thận rằng mã của bạn được định dạng độc đáo. Bạn nên chọn một bộ quy tắc đơn giản chi phối định dạng mã của mình và sau đó bạn nên áp dụng các quy tắc đó một cách nhất quán. Nếu bạn đang làm việc trong một nhóm, thì nhóm phải đồng ý với một bộ quy tắc định dạng duy nhất và tất cả các thành viên phải tuân thủ. Sẽ rất hữu ích khi có một công cụ tự động có thể áp dụng các quy tắc định dạng đó cho bạn.

[**The Purpose of Formatting**](#_page_237_0)  
Mục đích của việc định dạng

First of all, let’s be clear. Code formatting is *important*. It is too important to ignore and it is too important to treat religiously. Code formatting is about communication, and communication is the professional developer’s ﬁrst order of business.  
Trước hết, hãy rõ ràng. Định dạng mã là quan trọng. Quá quan trọng để bỏ qua và quá quan trọng để đối xử một cách tôn giáo. Định dạng mã là về giao tiếp và giao tiếp là yêu cầu kinh doanh đầu tiên của nhà phát triển chuyên nghiệp.

Perhaps you thought that “getting it working” was the ﬁrst order of business for a professional developer. I hope by now, however, that this book has disabused you of that idea. The functionality that you create today has a good chance of changing in the next release, but the readability of your code will have a profound effect on all the changes that will ever be made. The coding style and readability set precedents that continue to affect maintainability and extensibility long after the original code has been changed beyond recognition. Your style and discipline survives, even though your code does not.  
Có lẽ bạn nghĩ rằng “làm cho nó hoạt động” là nhiệm vụ đầu tiên của một nhà phát triển chuyên nghiệp. Tuy nhiên, đến giờ tôi hy vọng rằng cuốn sách này đã giúp bạn loại bỏ ý tưởng đó. Chức năng mà bạn tạo hôm nay có nhiều khả năng thay đổi trong bản phát hành tiếp theo, nhưng khả năng đọc mã của bạn sẽ có ảnh hưởng sâu sắc đến tất cả những thay đổi sẽ được thực hiện. Phong cách viết mã và khả năng đọc đã đặt ra các tiền lệ tiếp tục ảnh hưởng đến khả năng bảo trì và khả năng mở rộng rất lâu sau khi mã gốc đã bị thay đổi ngoài sự công nhận. Phong cách và kỷ luật của bạn tồn tại, mặc dù mã của bạn thì không.

So what are the formatting issues that help us to communicate best?  
Vậy các vấn đề về định dạng giúp chúng ta giao tiếp tốt nhất là gì?

[**Vertical Formatting**](#_page_237_0)  
Định dạng dọc

Let’s start with vertical size. How big should a source ﬁle be? In Java, ﬁle size is closely related to class size. We’ll talk about class size when we talk about classes. For the moment let’s just consider ﬁle size.  
Hãy bắt đầu với kích thước dọc. Một tập tin nguồn nên lớn như thế nào? Trong Java, kích thước tệp có liên quan chặt chẽ với kích thước lớp. Chúng ta sẽ nói về quy mô lớp học khi chúng ta nói về các lớp học. Hiện tại, chúng ta hãy xem xét kích thước tệp.

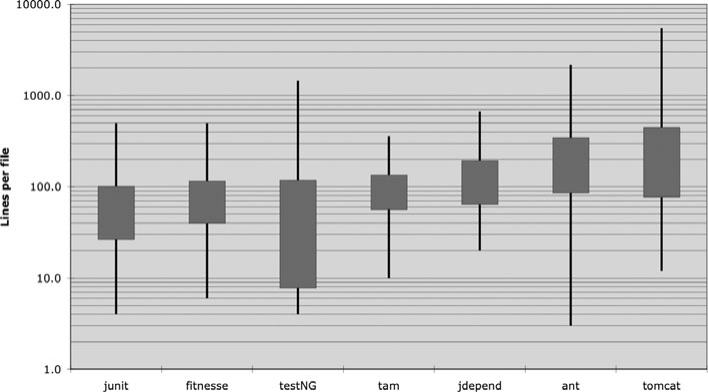
How big are most Java source ﬁles? It turns out that there is a huge range of sizes and some remarkable differences in style. Figure 5-1 shows some of those differences.  
Hầu hết các tệp mã nguồn Java lớn đến mức nào? Nó chỉ ra rằng có rất nhiều kích cỡ và một số khác biệt đáng chú ý về kiểu dáng. Hình 5-1 cho thấy một số khác biệt đó.

Seven different projects are depicted. Junit, FitNesse, testNG, Time and Money, JDepend, Ant, and Tomcat. The lines through the boxes show the minimum and maxi-mum ﬁle lengths in each project. The box shows approximately one-third (one standard deviation1) of the ﬁles. The middle of the box is the mean. So the average ﬁle size in the FitNesse project is about 65 lines, and about one-third of the ﬁles are between 40 and 100+ lines. The largest ﬁle in FitNesse is about 400 lines and the smallest is 6 lines. Note that this is a log scale, so the small difference in vertical position implies a very large difference in absolute size.  
Bảy dự án khác nhau được mô tả. Junit, FitNesse, testNG, Thời gian và Tiền bạc, JDepend, Ant và Tomcat. Các dòng thông qua các hộp hiển thị độ dài tệp tối thiểu và tối đa trong mỗi dự án. Hộp hiển thị khoảng một phần ba (một độ lệch chuẩn1) của các tệp. Giữa hộp là giá trị trung bình. Vì vậy, kích thước tệp trung bình trong dự án FitNesse là khoảng 65 dòng và khoảng một phần ba số tệp nằm trong khoảng từ 40 đến hơn 100 dòng. Tệp lớn nhất trong FitNesse là khoảng 400 dòng và nhỏ nhất là 6 dòng. Lưu ý rằng đây là thang đo log, vì vậy sự khác biệt nhỏ ở vị trí thẳng đứng ngụ ý sự khác biệt rất lớn về kích thước tuyệt đối.

1. The box shows sigma/2 above and below the mean. Yes, I know that the ﬁle length distribution is not normal, and so the stan-dard deviation is not mathematically precise. But we’re not trying for precision here. We’re just trying to get a feel.  
1. Hộp hiển thị sigma/2 trên và dưới giá trị trung bình. Vâng, tôi biết rằng phân phối độ dài tệp không bình thường và do đó độ lệch chuẩn không chính xác về mặt toán học. Nhưng chúng tôi không cố gắng cho độ chính xác ở đây. Chúng tôi chỉ đang cố gắng để có được một cảm giác.

<https://www.facebook.com/groups/phattrienlaptrinh>

**Vertical Formatting** 77  
Định dạng Dọc 77



**Figure 5-1  
Hình 5-1**

File length distributions LOG scale (box height = sigma)  
Phân phối độ dài tệp Tỷ lệ LOG (chiều cao hộp = sigma)

Junit, FitNesse, and Time and Money are composed of relatively small ﬁles. None are over 500 lines and most of those ﬁles are less than 200 lines. Tomcat and Ant, on the other hand, have some ﬁles that are several thousand lines long and close to half are over 200 lines.  
Junit, FitNesse và Time and Money bao gồm các tập tin tương đối nhỏ. Không có tệp nào vượt quá 500 dòng và hầu hết các tệp đó đều dưới 200 dòng. Mặt khác, Tomcat và Ant có một số tệp dài vài nghìn dòng và gần một nửa là trên 200 dòng.

What does that mean to us? It appears to be possible to build signiﬁcant systems (FitNesse is close to 50,000 lines) out of ﬁles that are typically 200 lines long, with an upper limit of 500. Although this should not be a hard and fast rule, it should be considered very desirable. Small ﬁles are usually easier to understand than large ﬁles are.  
Điều đó có ý nghĩa gì đối với chúng ta? Dường như có thể xây dựng các hệ thống quan trọng (FitNesse có gần 50.000 dòng) từ các tệp thường dài 200 dòng, với giới hạn trên là 500. Mặc dù đây không phải là một quy tắc khó và nhanh, nhưng nó nên được xem xét rất kỹ lưỡng. mong muốn. Các tệp nhỏ thường dễ hiểu hơn các tệp lớn.

[**The Newspaper Metaphor**](#_page_237_0)  
Phép ẩn dụ trên báo

Think of a well-written newspaper article. You read it vertically. At the top you expect a headline that will tell you what the story is about and allows you to decide whether it is something you want to read. The ﬁrst paragraph gives you a synopsis of the whole story, hiding all the details while giving you the broad-brush concepts. As you continue down-ward, the details increase until you have all the dates, names, quotes, claims, and other minutia.  
Hãy nghĩ về một bài báo được viết tốt. Bạn đọc nó theo chiều dọc. Ở đầu, bạn mong đợi một tiêu đề sẽ cho bạn biết câu chuyện nói về cái gì và cho phép bạn quyết định xem đó có phải là thứ bạn muốn đọc hay không. Đoạn đầu tiên cung cấp cho bạn một bản tóm tắt của toàn bộ câu chuyện, che giấu tất cả các chi tiết trong khi cung cấp cho bạn các khái niệm chung. Khi bạn tiếp tục đi xuống, các chi tiết sẽ tăng lên cho đến khi bạn có tất cả ngày tháng, tên, trích dẫn, tuyên bố và các chi tiết vụn vặt khác.

We would like a source ﬁle to be like a newspaper article. The name should be simple but explanatory. The name, by itself, should be sufﬁcient to tell us whether we are in the right module or not. The topmost parts of the source ﬁle should provide the high-level  
Chúng tôi muốn một tập tin nguồn giống như một bài báo. Tên nên đơn giản nhưng giải thích. Bản thân cái tên đã đủ để cho chúng ta biết liệu chúng ta có đang ở đúng mô-đun hay không. Các phần trên cùng của tệp nguồn phải cung cấp mức độ cao

<https://www.facebook.com/groups/phattrienlaptrinh>

78 **Chapter 5: Formatting**  
78 Chương 5: Định dạng

concepts and algorithms. Detail should increase as we move downward, until at the end we ﬁnd the lowest level functions and details in the source ﬁle.  
khái niệm và thuật toán. Chi tiết sẽ tăng lên khi chúng ta di chuyển xuống dưới, cho đến khi kết thúc, chúng ta tìm thấy các chức năng và chi tiết mức thấp nhất trong tệp nguồn.

A newspaper is composed of many articles; most are very small. Some are a bit larger. Very few contain as much text as a page can hold. This makes the newspaper *usable*. If the newspaper were just one long story containing a disorganized agglomeration of facts, dates, and names, then we simply would not read it.  
Một tờ báo bao gồm nhiều bài viết; hầu hết đều rất nhỏ. Một số lớn hơn một chút. Rất ít chứa nhiều văn bản như một trang có thể chứa. Điều này làm cho tờ báo có thể sử dụng được. Nếu tờ báo chỉ là một câu chuyện dài chứa đựng sự kết hợp lộn xộn của các sự kiện, ngày tháng và tên tuổi, thì đơn giản là chúng ta sẽ không đọc nó.

[**Vertical Openness Between Concepts**](#_page_237_0)  
Độ mở dọc giữa các khái niệm

Nearly all code is read left to right and top to bottom. Each line represents an expression or a clause, and each group of lines represents a complete thought. Those thoughts should be separated from each other with blank lines.  
Gần như tất cả mã được đọc từ trái sang phải và từ trên xuống dưới. Mỗi dòng đại diện cho một biểu thức hoặc một mệnh đề và mỗi nhóm dòng đại diện cho một ý nghĩ hoàn chỉnh. Những suy nghĩ đó nên được ngăn cách với nhau bằng những dòng trống.

Consider, for example, Listing 5-1. There are blank lines that separate the package declaration, the import(s), and each of the functions. This extremely simple rule has a pro-found effect on the visual layout of the code. Each blank line is a visual cue that identiﬁes a new and separate concept. As you scan down the listing, your eye is drawn to the ﬁrst line that follows a blank line.  
Ví dụ, xem xét Liệt kê 5-1. Có các dòng trống phân tách phần khai báo gói, (các) phần nhập và từng chức năng. Quy tắc cực kỳ đơn giản này có ảnh hưởng sâu sắc đến bố cục trực quan của mã. Mỗi dòng trống là một gợi ý trực quan xác định một khái niệm mới và riêng biệt. Khi bạn lướt qua danh sách, mắt của bạn sẽ bị thu hút vào dòng đầu tiên tiếp theo một dòng trống.

**Listing 5-1 BoldWidget.java  
Liệt kê 5-1 BoldWidget.java**

package fitnesse.wikitext.widgets;

import java.util.regex.\*;

public class BoldWidget extends ParentWidget { public static final String REGEXP = "'''.+?'''";

private static final Pattern pattern = Pattern.compile("'''(.+?)'''", Pattern.MULTILINE + Pattern.DOTALL

);

public BoldWidget(ParentWidget parent, String text) throws Exception { super(parent);

Matcher match = pattern.matcher(text); match.find(); addChildWidgets(match.group(1));

}

public String render() throws Exception { StringBuffer html = new StringBuffer("<b>"); html.append(childHtml()).append("</b>"); return html.toString();

} }

Taking those blank lines out, as in Listing 5-2, has a remarkably obscuring effect on the readability of the code.  
Loại bỏ những dòng trống đó, như trong Liệt kê 5-2, có tác động che khuất đáng kể khả năng đọc mã.

<https://www.facebook.com/groups/phattrienlaptrinh>

**Vertical Formatting** 79  
Định dạng Dọc 79

**Listing 5-2 BoldWidget.java**  
Liệt kê 5-2 BoldWidget.java

package fitnesse.wikitext.widgets; import java.util.regex.\*;

public class BoldWidget extends ParentWidget { public static final String REGEXP = "'''.+?'''";

private static final Pattern pattern = Pattern.compile("'''(.+?)'''", Pattern.MULTILINE + Pattern.DOTALL);

public BoldWidget(ParentWidget parent, String text) throws Exception { super(parent);

Matcher match = pattern.matcher(text); match.find(); addChildWidgets(match.group(1));}

public String render() throws Exception { StringBuffer html = new StringBuffer("<b>"); html.append(childHtml()).append("</b>"); return html.toString();

} }

This effect is even more pronounced when you unfocus your eyes. In the ﬁrst example the different groupings of lines pop out at you, whereas the second example looks like a muddle. The difference between these two listings is a bit of vertical openness.  
Hiệu ứng này thậm chí còn rõ rệt hơn khi bạn không tập trung vào mắt. Trong ví dụ đầu tiên, các nhóm đường kẻ khác nhau hiện ra trước mắt bạn, trong khi ví dụ thứ hai trông giống như một mớ hỗn độn. Sự khác biệt giữa hai danh sách này là một chút độ mở theo chiều dọc.

[**Vertical Density**](#_page_237_0)  
Mật độ dọc

If openness separates concepts, then vertical density implies close association. So lines of code that are tightly related should appear vertically dense. Notice how the useless comments in Listing 5-3 break the close association of the two instance variables.  
Nếu tính mở ngăn cách các khái niệm, thì mật độ dọc bao hàm sự liên kết chặt chẽ. Vì vậy, các dòng mã có liên quan chặt chẽ sẽ xuất hiện dày đặc theo chiều dọc. Lưu ý cách các nhận xét vô dụng trong Liệt kê 5-3 phá vỡ sự liên kết chặt chẽ của hai biến thể hiện.

**Listing 5-3  
Liệt kê 5-3**

public class ReporterConfig {

/\*\*

\* The class name of the reporter listener \*/

private String m\_className;

/\*\*

\* The properties of the reporter listener \*/

private List<Property> m\_properties = new ArrayList<Property>();

public void addProperty(Property property) { m\_properties.add(property);

}

Listing 5-4 is much easier to read. It ﬁts in an “eye-full,” or at least it does for me. I can look at it and see that this is a class with two variables and a method, without having to move my head or eyes much. The previous listing forces me to use much more eye and head motion to achieve the same level of comprehension.  
Liệt kê 5-4 dễ đọc hơn nhiều. Nó phù hợp với một "mắt đầy", hoặc ít nhất là nó phù hợp với tôi. Tôi có thể nhìn vào nó và thấy rằng đây là một lớp có hai biến và một phương thức mà không cần phải di chuyển đầu hoặc mắt nhiều. Việc liệt kê trước đó buộc tôi phải sử dụng nhiều chuyển động của mắt và đầu hơn để đạt được mức độ hiểu tương tự.

<https://www.facebook.com/groups/phattrienlaptrinh>

80 **Chapter 5: Formatting**  
80 Chương 5: Định dạng

**Listing 5-4**

public class ReporterConfig { private String m\_className;

private List<Property> m\_properties = new ArrayList<Property>();

public void addProperty(Property property) { m\_properties.add(property);

}

[**Vertical Distance**](#_page_237_0)  
Khoảng cách dọc

Have you ever chased your tail through a class, hopping from one function to the next, scrolling up and down the source ﬁle, trying to divine how the functions relate and operate, only to get lost in a rat’s nest of confusion? Have you ever hunted up the chain of inheritance for the deﬁnition of a variable or function? This is frustrating because you are trying to understand *what* the system does, but you are spending your time and mental energy on trying to locate and remember *where* the pieces are.  
Bạn đã bao giờ đuổi theo cái đuôi của mình trong một lớp học, nhảy từ chức năng này sang chức năng tiếp theo, cuộn lên và xuống tệp nguồn, cố gắng đoán xem các chức năng đó liên quan và hoạt động như thế nào, chỉ để bị lạc trong một mớ hỗn độn chưa? Bạn đã bao giờ tìm kiếm chuỗi kế thừa để tìm định nghĩa của một biến hoặc hàm chưa? Điều này thật khó chịu vì bạn đang cố gắng hiểu những gì hệ thống làm, nhưng bạn đang dành thời gian và năng lượng tinh thần của mình để cố gắng xác định vị trí và ghi nhớ vị trí của các mảnh ghép.

Concepts that are closely related should be kept vertically close to each other [G10]. Clearly this rule doesn’t work for concepts that belong in separate ﬁles. But then closely related concepts should not be separated into different ﬁles unless you have a very good reason. Indeed, this is one of the reasons that protected variables should be avoided.  
Các khái niệm có liên quan chặt chẽ nên được giữ gần nhau theo chiều dọc [G10]. Rõ ràng quy tắc này không hoạt động đối với các khái niệm thuộc về các tệp riêng biệt. Tuy nhiên, không nên tách các khái niệm có liên quan chặt chẽ thành các tệp khác nhau trừ khi bạn có lý do chính đáng. Thật vậy, đây là một trong những lý do nên tránh các biến được bảo vệ.

For those concepts that are so closely related that they belong in the same source ﬁle, their vertical separation should be a measure of how important each is to the understand-ability of the other. We want to avoid forcing our readers to hop around through our source ﬁles and classes.  
Đối với những khái niệm có liên quan chặt chẽ đến mức chúng thuộc về cùng một tệp nguồn, sự phân tách theo chiều dọc của chúng sẽ là thước đo mức độ quan trọng của từng khái niệm đối với khả năng hiểu của khái niệm kia. Chúng tôi muốn tránh buộc độc giả của chúng tôi phải nhảy qua các lớp và tệp nguồn của chúng tôi.

**Variable Declarations.** Variables should be declared as close to their usage as possi-ble. Because our functions are very short, local variables should appear a the top of each function, as in this longish function from Junit4.3.1.  
Khai báo biến. Các biến nên được khai báo càng gần với cách sử dụng của chúng càng tốt. Vì các hàm của chúng ta rất ngắn nên các biến cục bộ sẽ xuất hiện ở đầu mỗi hàm, như trong hàm dài này từ Junit4.3.1.

private static void readPreferences() { **InputStream is= null;**

try {

is= new FileInputStream(getPreferencesFile()); setPreferences(new Properties(getPreferences())); getPreferences().load(is);

} catch (IOException e) { try {

if (is != null) is.close();

} catch (IOException e1) { }

} }

Control variables for loops should usually be declared within the loop statement, as in this cute little function from the same source.  
Các biến điều khiển cho các vòng lặp thường phải được khai báo trong câu lệnh vòng lặp, như trong hàm nhỏ dễ thương này từ cùng một nguồn.

<https://www.facebook.com/groups/phattrienlaptrinh>

**Vertical Formatting** 81  
Định dạng Dọc 81

public int countTestCases() { int count= 0;

for (**Test each** : tests)

count += each.countTestCases(); return count;

}

In rare cases a variable might be declared at the top of a block or just before a loop in a long-ish function. You can see such a variable in this snippet from the midst of a very long function in TestNG.  
Trong một số ít trường hợp, một biến có thể được khai báo ở đầu khối hoặc ngay trước một vòng lặp trong hàm long-ish. Bạn có thể thấy một biến như vậy trong đoạn mã này ở giữa một hàm rất dài trong TestNG.

...

for (XmlTest test : m\_suite.getTests()) {

**TestRunner tr = m\_runnerFactory.newTestRunner(this, test);** tr.addListener(m\_textReporter);

m\_testRunners.add(tr);

invoker = tr.getInvoker();

for (ITestNGMethod m : tr.getBeforeSuiteMethods()) { beforeSuiteMethods.put(m.getMethod(), m);

}

for (ITestNGMethod m : tr.getAfterSuiteMethods()) { afterSuiteMethods.put(m.getMethod(), m);

} }

...

**Instance variables,** on the other hand, should be declared at the top of the class. This should not increase the vertical distance of these variables, because in a well-designed class, they are used by many, if not all, of the methods of the class.  
Mặt khác, các biến thể hiện phải được khai báo ở đầu lớp. Điều này sẽ không làm tăng khoảng cách theo chiều dọc của các biến này, bởi vì trong một lớp được thiết kế tốt, chúng được sử dụng bởi nhiều, nếu không muốn nói là tất cả, các phương thức của lớp.

There have been many debates over where instance variables should go. In C++ we commonly practiced the so-called *scissors rule*, which put all the instance variables at the bottom. The common convention in Java, however, is to put them all at the top of the class. I see no reason to follow any other convention. The important thing is for the instance vari-ables to be declared in one well-known place. Everybody should know where to go to see the declarations.  
Đã có nhiều cuộc tranh luận về việc nên đặt các biến thể hiện ở đâu. Trong C++, chúng ta thường thực hành cái gọi là quy tắc cái kéo, quy tắc này đặt tất cả các biến thể hiện ở dưới cùng. Tuy nhiên, quy ước chung trong Java là đặt tất cả chúng ở đầu lớp. Tôi thấy không có lý do gì để tuân theo bất kỳ quy ước nào khác. Điều quan trọng là các biến thể hiện phải được khai báo ở một nơi nổi tiếng. Mọi người nên biết nơi để đi để xem các tuyên bố.

Consider, for example, the strange case of the TestSuiteclass in JUnit 4.3.1. I have greatly attenuated this class to make the point. If you look about halfway down the listing, you will see two instance variables declared there. It would be hard to hide them in a better place. Someone reading this code would have to stumble across the declarations by acci-dent (as I did).  
Ví dụ, hãy xem xét trường hợp kỳ lạ của TestSuiteclass trong JUnit 4.3.1. Tôi đã làm suy yếu rất nhiều lớp này để làm cho quan điểm. Nếu bạn nhìn vào giữa danh sách, bạn sẽ thấy hai biến thể hiện được khai báo ở đó. Thật khó để giấu chúng ở một nơi tốt hơn. Một người nào đó đọc mã này sẽ phải tình cờ đọc được các khai báo (như tôi đã làm).

public class TestSuite implements Test {

static public Test createTest(Class<? extends TestCase> theClass, String name) {

... }

<https://www.facebook.com/groups/phattrienlaptrinh>

82 **Chapter 5: Formatting**  
82 Chương 5: Định dạng

public static Constructor<? extends TestCase> getTestConstructor(Class<? extends TestCase> theClass) throws NoSuchMethodException {

... }

public static Test warning(final String message) { ...

}

private static String exceptionToString(Throwable t) { ...

}

**private String fName;**

**private Vector<Test> fTests= new Vector<Test>(10);**

public TestSuite() { }

public TestSuite(final Class<? extends TestCase> theClass) { ...

}

public TestSuite(Class<? extends TestCase> theClass, String name) { ...

}

... ... ... ... ... }

**Dependent Functions.** If one function calls another, they should be vertically close, and the caller should be above the callee, if at all possible. This gives the program a natural ﬂow. If the convention is followed reliably, readers will be able to trust that function deﬁni-tions will follow shortly after their use. Consider, for example, the snippet from FitNesse in Listing 5-5. Notice how the topmost function calls those below it and how they in turn call those below them. This makes it easy to ﬁnd the called functions and greatly enhances the readability of the whole module.  
Chức năng phụ thuộc. Nếu một chức năng gọi một chức năng khác, chúng phải được đóng theo chiều dọc và người gọi phải ở trên callee, nếu có thể. Điều này mang lại cho chương trình một dòng chảy tự nhiên. Nếu quy ước được tuân theo một cách đáng tin cậy, người đọc sẽ có thể tin tưởng rằng các định nghĩa hàm sẽ tuân theo ngay sau khi sử dụng. Ví dụ, xem xét đoạn trích từ FitNesse trong Liệt kê 5-5. Lưu ý cách hàm trên cùng gọi những hàm bên dưới nó và cách chúng lần lượt gọi những hàm bên dưới chúng. Điều này giúp dễ dàng tìm thấy các chức năng được gọi và nâng cao đáng kể khả năng đọc của toàn bộ mô-đun.

**Listing 5-5 WikiPageResponder.java  
Liệt kê 5-5 WikiPageResponder.java**

public class WikiPageResponder implements SecureResponder { protected WikiPage page;

protected PageData pageData; protected String pageTitle; protected Request request; protected PageCrawler crawler;

public Response makeResponse(FitNesseContext context, Request request) throws Exception {

String pageName = getPageNameOrDefault(request, "FrontPage");

<https://www.facebook.com/groups/phattrienlaptrinh>

**Vertical Formatting** 83  
Định dạng Dọc 83

**Listing 5-5 (continued) WikiPageResponder.java**  
Liệt kê 5-5 (tiếp theo) WikiPageResponder.java

loadPage(pageName, context); if (page == null)

return notFoundResponse(context, request); else

return makePageResponse(context); }

private String getPageNameOrDefault(Request request, String defaultPageName) {

String pageName = request.getResource(); if (StringUtil.isBlank(pageName))

pageName = defaultPageName;

return pageName; }

protected void loadPage(String resource, FitNesseContext context) throws Exception {

WikiPagePath path = PathParser.parse(resource); crawler = context.root.getPageCrawler();

crawler.setDeadEndStrategy(new VirtualEnabledPageCrawler()); page = crawler.getPage(context.root, path);

if (page != null)

pageData = page.getData(); }

private Response notFoundResponse(FitNesseContext context, Request request) throws Exception {

return new NotFoundResponder().makeResponse(context, request); }

private SimpleResponse makePageResponse(FitNesseContext context) throws Exception {

pageTitle = PathParser.render(crawler.getFullPath(page)); String html = makeHtml(context);

SimpleResponse response = new SimpleResponse(); response.setMaxAge(0); response.setContent(html);

return response; }

...

As an aside, this snippet provides a nice example of keeping constants at the appropri-ate level [G35]. The "FrontPage" constant could have been buried in the getPageNameOrDefaultfunction, but that would have hidden a well-known and expected constant in an inappropriately low-level function. It was better to pass that constant down from the place where it makes sense to know it to the place that actually uses it.  
Bên cạnh đó, đoạn mã này cung cấp một ví dụ hay về việc giữ các hằng số ở mức phù hợp [G35]. Hằng số "FrontPage" có thể đã bị chôn vùi trong hàm getPageNameOrDefault, nhưng điều đó sẽ ẩn một hằng số nổi tiếng và được mong đợi trong một hàm cấp thấp không phù hợp. Tốt hơn hết là chuyển hằng số đó từ nơi có ý nghĩa để biết nó sang nơi thực sự sử dụng nó.

<https://www.facebook.com/groups/phattrienlaptrinh>

84 **Chapter 5: Formatting**  
84 Chương 5: Định dạng

**Conceptual Afﬁnity.** Certain bits of code *want* to be near other bits. They have a certain conceptual afﬁnity. The stronger that afﬁnity, the less vertical distance there should be between them.  
Mối quan hệ khái niệm. Một số bit mã muốn ở gần các bit khác. Họ có một mối quan hệ khái niệm nhất định. Ái lực đó càng mạnh thì khoảng cách thẳng đứng giữa chúng càng ít.

As we have seen, this afﬁnity might be based on a direct dependence, such as one function call-ing another, or a function using a variable. But there are other possible causes of afﬁnity. Afﬁnity might be caused because a group of functions per-form a similar operation. Consider this snippet of code from Junit 4.3.1:  
Như chúng ta đã thấy, mối quan hệ này có thể dựa trên sự phụ thuộc trực tiếp, chẳng hạn như một hàm gọi một hàm khác, hoặc một hàm sử dụng một biến. Nhưng có thể có những nguyên nhân khác của mối quan hệ. Mối quan hệ có thể được gây ra bởi vì một nhóm các chức năng thực hiện một hoạt động tương tự. Hãy xem xét đoạn mã này từ Junit 4.3.1:

public class Assert {

static public void assertTrue(String message, boolean condition) { if (!condition)

fail(message); }

static public void assertTrue(boolean condition) { assertTrue(null, condition);

}

static public void assertFalse(String message, boolean condition) { assertTrue(message, !condition);

}

static public void assertFalse(boolean condition) { assertFalse(null, condition);

} ...

These functions have a strong conceptual afﬁnity because they share a common naming scheme and perform variations of the same basic task. The fact that they call each other is secondary. Even if they didn’t, they would still want to be close together.  
Các chức năng này có mối quan hệ khái niệm mạnh mẽ vì chúng chia sẻ một sơ đồ đặt tên chung và thực hiện các biến thể của cùng một nhiệm vụ cơ bản. Việc họ gọi cho nhau chỉ là thứ yếu. Ngay cả khi họ không, họ vẫn muốn được gần nhau.

[**Vertical Ordering**](#_page_237_0)  
Thứ tự dọc

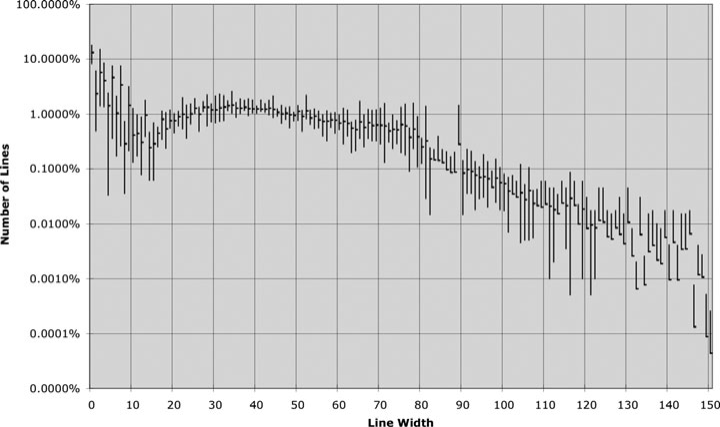
In general we want function call dependencies to point in the downward direction. That is, a function that is called should be below a function that does the calling.2 This creates a nice ﬂow down the source code module from high level to low level.  
Nói chung, chúng tôi muốn các phụ thuộc cuộc gọi hàm trỏ theo hướng đi xuống. Nghĩa là, một chức năng được gọi phải ở dưới một chức năng thực hiện việc gọi.2 Điều này tạo ra một luồng tốt đẹp cho mô-đun mã nguồn từ mức cao xuống mức thấp.

As in newspaper articles, we expect the most important concepts to come ﬁrst, and we expect them to be expressed with the least amount of polluting detail. We expect the low-level details to come last. This allows us to skim source ﬁles, getting the gist from the  
Như trong các bài báo, chúng tôi mong đợi những khái niệm quan trọng nhất sẽ xuất hiện trước và chúng tôi mong đợi chúng được thể hiện với ít chi tiết gây ô nhiễm nhất. Chúng tôi hy vọng các chi tiết cấp thấp sẽ xuất hiện cuối cùng. Điều này cho phép chúng tôi đọc lướt qua các tệp nguồn, lấy ý chính từ

2. This is the exact opposite of languages like Pascal, C, and C++ that enforce functions to be deﬁned, or at least declared, *before* they are used.  
2. Điều này hoàn toàn ngược lại với các ngôn ngữ như Pascal, C và C++, bắt buộc các hàm phải được xác định hoặc ít nhất là được khai báo trước khi chúng được sử dụng.

<https://www.facebook.com/groups/phattrienlaptrinh>

**Horizontal Formatting** 85  
Định dạng ngang 85



ﬁrst few functions, without having to immerse ourselves in the details. Listing 5-5 is organized this way. Perhaps even better examples are Listing 15-5 on page 263, and List-ing 3-7 on page 50.  
một số chức năng đầu tiên mà không cần phải tìm hiểu chi tiết. Liệt kê 5-5 được tổ chức theo cách này. Có lẽ các ví dụ tốt hơn nữa là Liệt kê 15-5 trên trang 263 và Liệt kê 3-7 trên trang 50.

[**Horizontal Formatting**](#_page_237_0)  
Định dạng ngang

How wide should a line be? To answer that, let’s look at how wide lines are in typical pro-grams. Again, we examine the seven different projects. Figure 5-2 shows the distribution of line lengths of all seven projects. The regularity is impressive, especially right around 45 characters. Indeed, every size from 20 to 60 represents about 1 percent of the total number of lines. That’s 40 percent! Perhaps another 30 percent are less than 10 characters wide. Remember this is a log scale, so the linear appearance of the drop-off above 80 char-acters is really very signiﬁcant. Programmers clearly prefer short lines.  
Một đường nên rộng bao nhiêu? Để trả lời điều đó, chúng ta hãy xem độ rộng của các dòng trong các chương trình điển hình. Một lần nữa, chúng tôi kiểm tra bảy dự án khác nhau. Hình 5-2 cho thấy sự phân bổ độ dài đường của tất cả bảy dự án. Mức độ đều đặn rất ấn tượng, đặc biệt là khoảng 45 ký tự. Thật vậy, mỗi kích thước từ 20 đến 60 đại diện cho khoảng 1 phần trăm tổng số dòng. Đó là 40 phần trăm! Có lẽ 30 phần trăm khác có chiều rộng dưới 10 ký tự. Hãy nhớ rằng đây là một thang đo nhật ký, do đó, sự xuất hiện tuyến tính của việc bỏ qua trên 80 ký tự thực sự rất quan trọng. Các lập trình viên rõ ràng thích các dòng ngắn hơn.

**Figure 5-2  
Hình 5-2**

Java line width distribution  
Phân phối chiều rộng dòng Java

This suggests that we should strive to keep our lines short. The old Hollerith limit of 80 is a bit arbitrary, and I’m not opposed to lines edging out to 100 or even 120. But beyond that is probably just careless.  
Điều này gợi ý rằng chúng ta nên cố gắng giữ cho lời thoại của mình ngắn gọn. Giới hạn cũ của Hollerith là 80 hơi tùy tiện và tôi không phản đối việc vạch ra tới 100 hoặc thậm chí 120. Nhưng vượt quá mức đó có lẽ chỉ là bất cẩn.

I used to follow the rule that you should never have to scroll to the right. But monitors are too wide for that nowadays, and younger programmers can shrink the font so small  
Tôi đã từng tuân theo quy tắc rằng bạn không bao giờ phải cuộn sang phải. Nhưng màn hình ngày nay quá rộng so với điều đó và các lập trình viên trẻ hơn có thể thu nhỏ phông chữ quá nhỏ

<https://www.facebook.com/groups/phattrienlaptrinh>

86 **Chapter 5: Formatting**  
86 Chương 5: Định dạng

that they can get 200 characters across the screen. Don’t do that. I personally set my limit at 120.  
rằng họ có thể nhận được 200 ký tự trên màn hình. Đừng làm vậy. Cá nhân tôi đặt giới hạn của mình ở mức 120.

[**Horizontal Openness and Density**](#_page_237_0)  
Độ mở và mật độ ngang

We use horizontal white space to associate things that are strongly related and disassociate things that are more weakly related. Consider the following function:  
Chúng tôi sử dụng khoảng trắng ngang để liên kết những thứ có liên quan chặt chẽ và tách rời những thứ có liên quan yếu hơn. Xét hàm sau:

private void measureLine(String line) { lineCount++;

int lineSize = line.length(); totalChars += lineSize;

lineWidthHistogram.addLine(lineSize, lineCount); recordWidestLine(lineSize);

}

I surrounded the assignment operators with white space to accentuate them. Assignment statements have two distinct and major elements: the left side and the right side. The spaces make that separation obvious.  
Tôi bao quanh các toán tử gán bằng khoảng trắng để làm nổi bật chúng. Câu lệnh gán có hai phần tử chính và khác biệt: vế trái và vế phải. Các không gian làm cho sự tách biệt rõ ràng.

On the other hand, I didn’t put spaces between the function names and the opening parenthesis. This is because the function and its arguments are closely related. Separat-ing them makes them appear disjoined instead of conjoined. I separate arguments within the function call parenthesis to accentuate the comma and show that the arguments are separate.  
Mặt khác, tôi không đặt dấu cách giữa tên hàm và dấu ngoặc đơn mở. Điều này là do hàm và các đối số của nó có liên quan chặt chẽ với nhau. Việc tách chúng ra khiến chúng có vẻ rời rạc thay vì dính liền. Tôi tách các đối số trong dấu ngoặc đơn gọi hàm để làm nổi bật dấu phẩy và cho thấy rằng các đối số là riêng biệt.

Another use for white space is to accentuate the precedence of operators.  
Một cách sử dụng khác cho khoảng trắng là làm nổi bật thứ tự ưu tiên của các toán tử.

public class Quadratic {

public static double root1(double a, double b, double c) { double determinant = determinant(a, b, c);

return (-b + Math.sqrt(determinant)) / (2\*a); }

public static double root2(int a, int b, int c) { double determinant = determinant(a, b, c); return (-b - Math.sqrt(determinant)) / (2\*a);

}

private static double determinant(double a, double b, double c) { return b\*b - 4\*a\*c;

} }

Notice how nicely the equations read. The factors have no white space between them because they are high precedence. The terms are separated by white space because addi-tion and subtraction are lower precedence.  
Chú ý các phương trình đọc độc đáo như thế nào. Các yếu tố không có khoảng trắng giữa chúng vì chúng có mức độ ưu tiên cao. Các thuật ngữ được phân tách bằng khoảng trắng vì phép cộng và phép trừ có mức độ ưu tiên thấp hơn.

Unfortunately, most tools for reformatting code are blind to the precedence of operators and impose the same spacing throughout. So subtle spacings like those shown above tend to get lost after you reformat the code.  
Thật không may, hầu hết các công cụ để định dạng lại mã đều mù quáng về thứ tự ưu tiên của các toán tử và áp đặt cùng một khoảng cách trong suốt. Vì vậy, các khoảng cách nhỏ như được hiển thị ở trên có xu hướng bị mất sau khi bạn định dạng lại mã.

<https://www.facebook.com/groups/phattrienlaptrinh>

**Horizontal Formatting** 87  
Định dạng ngang 87

[**Horizontal Alignment**](#_page_237_0)  
Căn ngang

When I was an assembly language programmer,3 I used horizontal alignment to accentuate certain structures. When I started coding in C, C++, and eventually Java, I continued to try to line up all the variable names in a set of declarations, or all the rvalues in a set of assign-ment statements. My code might have looked like this:  
Khi tôi còn là một lập trình viên hợp ngữ,3 tôi đã sử dụng căn lề ngang để làm nổi bật các cấu trúc nhất định. Khi tôi bắt đầu viết mã bằng C, C++ và cuối cùng là Java, tôi tiếp tục cố gắng sắp xếp tất cả các tên biến trong một tập hợp các khai báo hoặc tất cả các giá trị trong một tập hợp các câu lệnh gán. Mã của tôi có thể trông như thế này:

public class FitNesseExpediter implements ResponseSender {

private Socket socket; private InputStream input; private OutputStream output; private Request request; private Response response; private FitNesseContext context;

protected long requestParsingTimeLimit; private long requestProgress; private long requestParsingDeadline; private boolean hasError;

public FitNesseExpediter(Socket s,

FitNesseContext context) throws Exception {

this.context = context; socket = s;

input = s.getInputStream(); output = s.getOutputStream(); requestParsingTimeLimit = 10000;

}

I have found, however, that this kind of alignment is not useful. The alignment seems to emphasize the wrong things and leads my eye away from the true intent. For example, in the list of declarations above you are tempted to read down the list of variable names with-out looking at their types. Likewise, in the list of assignment statements you are tempted to look down the list of rvalues without ever seeing the assignment operator. To make matters worse, automatic reformatting tools usually eliminate this kind of alignment.  
Tuy nhiên, tôi nhận thấy rằng kiểu sắp xếp này không hữu ích. Sự liên kết dường như nhấn mạnh những điều sai trái và khiến tôi không chú ý đến mục đích thực sự. Ví dụ, trong danh sách khai báo ở trên, bạn nên đọc danh sách tên biến mà không cần nhìn vào kiểu của chúng. Tương tự như vậy, trong danh sách các câu lệnh gán, bạn muốn xem danh sách các giá trị mà không bao giờ nhìn thấy toán tử gán. Tệ hơn nữa, các công cụ định dạng lại tự động thường loại bỏ kiểu căn chỉnh này.

So, in the end, I don’t do this kind of thing anymore. Nowadays I prefer unaligned declarations and assignments, as shown below, because they point out an important deﬁ-ciency. If I have long lists that need to be aligned, *the problem is the length of the lists*, not the lack of alignment. The length of the list of declarations in FitNesseExpediterbelow suggests that this class should be split up.  
Vì vậy, cuối cùng, tôi không làm những việc như thế này nữa. Ngày nay, tôi thích các khai báo và gán không được phân bổ, như được hiển thị bên dưới, bởi vì chúng chỉ ra một thiếu sót quan trọng. Nếu tôi có các danh sách dài cần được căn chỉnh, thì vấn đề là độ dài của danh sách chứ không phải thiếu sự liên kết. Độ dài của danh sách khai báo trong FitNesseExpediterbelow gợi ý rằng lớp này nên được tách ra.

public class FitNesseExpediter implements ResponseSender {

private Socket socket; private InputStream input; private OutputStream output; private Request request;

3. Who am I kidding? I still am an assembly language programmer. You can take the boy away from the metal, but you can’t take the metal out of the boy!  
3. Tôi đang đùa với ai vậy? Tôi vẫn là một lập trình viên hợp ngữ. Bạn có thể lấy cậu bé ra khỏi kim loại, nhưng bạn không thể lấy kim loại ra khỏi cậu bé!

<https://www.facebook.com/groups/phattrienlaptrinh>

88 **Chapter 5: Formatting**  
88 Chương 5: Định dạng

private Response response; private FitNesseContext context;

protected long requestParsingTimeLimit; private long requestProgress;

private long requestParsingDeadline; private boolean hasError;

public FitNesseExpediter(Socket s, FitNesseContext context) throws Exception {

this.context = context; socket = s;

input = s.getInputStream(); output = s.getOutputStream(); requestParsingTimeLimit = 10000;

}

[**Indentation**](#_page_237_0)  
thụt đầu dòng

A source ﬁle is a hierarchy rather like an outline. There is information that pertains to the ﬁle as a whole, to the individual classes within the ﬁle, to the methods within the classes, to the blocks within the methods, and recursively to the blocks within the blocks. Each level of this hierarchy is a scope into which names can be declared and in which declara-tions and executable statements are interpreted.  
Tệp nguồn là một cấu trúc phân cấp giống như một phác thảo. Có thông tin liên quan đến toàn bộ tệp, đến các lớp riêng lẻ trong tệp, đến các phương thức bên trong các lớp, đến các khối bên trong các phương thức và đệ quy đến các khối bên trong các khối. Mỗi cấp độ của hệ thống phân cấp này là một phạm vi trong đó các tên có thể được khai báo và trong đó các khai báo và câu lệnh thực thi được diễn giải.

To make this hierarchy of scopes visible, we indent the lines of source code in pro-portion to their position in the hiearchy. Statements at the level of the ﬁle, such as most class declarations, are not indented at all. Methods within a class are indented one level to the right of the class. Implementations of those methods are implemented one level to the right of the method declaration. Block implementations are implemented one level to the right of their containing block, and so on.  
Để hiển thị hệ thống phân cấp phạm vi này, chúng tôi thụt lề các dòng mã nguồn theo tỷ lệ tương ứng với vị trí của chúng trong hệ thống phân cấp. Các câu lệnh ở mức tệp, chẳng hạn như hầu hết các khai báo lớp, hoàn toàn không được thụt lề. Các phương thức trong một lớp được thụt vào một cấp ở bên phải của lớp. Việc triển khai các phương thức đó được triển khai ở một cấp độ bên phải của khai báo phương thức. Việc triển khai khối được triển khai một cấp ở bên phải của khối chứa chúng, v.v.

Programmers rely heavily on this indentation scheme. They visually line up lines on the left to see what scope they appear in. This allows them to quickly hop over scopes, such as implementations of ifor whilestatements, that are not relevant to their current situation. They scan the left for new method declarations, new variables, and even new classes. Without indentation, programs would be virtually unreadable by humans.  
Các lập trình viên phụ thuộc rất nhiều vào sơ đồ thụt đầu dòng này. Họ trực quan sắp xếp các dòng ở bên trái để xem chúng xuất hiện trong phạm vi nào. Điều này cho phép họ nhanh chóng chuyển qua các phạm vi, chẳng hạn như triển khai các câu lệnh if for while, không liên quan đến tình huống hiện tại của họ. Họ quét bên trái để tìm các khai báo phương thức mới, các biến mới và thậm chí cả các lớp mới. Nếu không có thụt đầu dòng, các chương trình sẽ hầu như không thể đọc được bởi con người.

Consider the following programs that are syntactically and semantically identical:  
Hãy xem xét các chương trình sau giống hệt nhau về mặt cú pháp và ngữ nghĩa:

public class FitNesseServer implements SocketServer { private FitNesseContext context; public FitNesseServer(FitNesseContext context) { this.context = context; } public void serve(Socket s) { serve(s, 10000); } public void serve(Socket s, long requestTimeout) { try { FitNesseExpediter sender = new FitNesseExpediter(s, context); sender.setRequestParsingTimeLimit(requestTimeout); sender.start(); } catch(Exception e) { e.printStackTrace(); } } }

-----

public class FitNesseServer implements SocketServer { private FitNesseContext context;

<https://www.facebook.com/groups/phattrienlaptrinh>

**Horizontal Formatting** 89  
Định dạng ngang 89

public FitNesseServer(FitNesseContext context) { this.context = context;

}

public void serve(Socket s) { serve(s, 10000);

}

public void serve(Socket s, long requestTimeout) { try {

FitNesseExpediter sender = new FitNesseExpediter(s, context); sender.setRequestParsingTimeLimit(requestTimeout); sender.start();

}

catch (Exception e) { e.printStackTrace();

} }

}

Your eye can rapidly discern the structure of the indented ﬁle. You can almost instantly spot the variables, constructors, accessors, and methods. It takes just a few seconds to real-ize that this is some kind of simple front end to a socket, with a time-out. The unindented version, however, is virtually impenetrable without intense study.  
Mắt bạn có thể nhanh chóng nhận ra cấu trúc của tệp được thụt vào. Bạn gần như có thể ngay lập tức phát hiện ra các biến, hàm tạo, bộ truy cập và phương thức. Chỉ mất vài giây để nhận ra rằng đây là một loại giao diện người dùng đơn giản nào đó cho ổ cắm, có thời gian chờ. Tuy nhiên, phiên bản không thụt lề hầu như không thể thâm nhập được nếu không nghiên cứu kỹ lưỡng.

**Breaking Indentation.** It is sometimes tempting to break the indentation rule for short ifstatements, short whileloops, or short functions. Whenever I have succumbed to this temptation, I have almost always gone back and put the indentation back in. So I avoid col-lapsing scopes down to one line like this:  
Phá vỡ thụt đầu dòng. Đôi khi, việc phá vỡ quy tắc thụt đầu dòng cho các câu lệnh if ngắn, vòng lặp while ngắn hoặc các hàm ngắn đôi khi rất hấp dẫn. Bất cứ khi nào tôi không chịu nổi sự cám dỗ này, tôi hầu như luôn quay lại và đặt lại vết lõm vào. Vì vậy, tôi tránh thu gọn phạm vi xuống còn một dòng như thế này:

public class CommentWidget extends TextWidget {

public static final String REGEXP = "^#[^\r\n]\*(?:(?:\r\n)|\n|\r)?";

public CommentWidget(ParentWidget parent, String text){super(parent, text);} public String render() throws Exception {return ""; }

}

I prefer to expand and indent the scopes instead, like this:

public class CommentWidget extends TextWidget {

public static final String REGEXP = "^#[^\r\n]\*(?:(?:\r\n)|\n|\r)?";

public CommentWidget(ParentWidget parent, String text) { super(parent, text);

}

public String render() throws Exception { return "";

} }

<https://www.facebook.com/groups/phattrienlaptrinh>

90 **Chapter 5: Formatting**  
90 Chương 5: Định dạng

[**Dummy Scopes**](#_page_237_0)  
ống ngắm giả

Sometimes the body of a whileor forstatement is a dummy, as shown below. I don’t like these kinds of structures and try to avoid them. When I can’t avoid them, I make sure that the dummy body is properly indented and surrounded by braces. I can’t tell you how many times I’ve been fooled by a semicolon silently sitting at the end of a whileloop on the same line. Unless you make that semicolon *visible* by indenting it on it’s own line, it’s just too hard to see.  
Đôi khi phần thân của một while hoặc forstatement là một hình nộm, như minh họa bên dưới. Tôi không thích những kiểu cấu trúc này và cố gắng tránh chúng. Khi tôi không thể tránh chúng, tôi đảm bảo rằng cơ thể giả được thụt vào đúng cách và được bao quanh bởi các thanh giằng. Tôi không thể nói cho bạn biết đã bao nhiêu lần tôi bị đánh lừa bởi một dấu chấm phẩy âm thầm nằm ở cuối vòng lặp while trên cùng một dòng. Trừ khi bạn làm cho dấu chấm phẩy đó hiển thị bằng cách thụt vào dòng riêng của nó, nếu không thì nó quá khó để nhìn thấy.

while (dis.read(buf, 0, readBufferSize) != -1) ;

[**Team Rules**](#_page_237_0)  
Nội quy đội

The title of this section is a play on words. Every programmer has his own favorite formatting rules, but if he works in a team, then the team rules.  
Tiêu đề của phần này là một cách chơi chữ. Mỗi lập trình viên đều có quy tắc định dạng yêu thích của riêng mình, nhưng nếu anh ta làm việc theo nhóm, thì quy tắc của nhóm.

A team of developers should agree upon a single formatting style, and then every member of that team should use that style. We want the software to have a  
Một nhóm các nhà phát triển nên thống nhất về một kiểu định dạng duy nhất và sau đó mọi thành viên của nhóm đó nên sử dụng kiểu đó. Chúng tôi muốn phần mềm có một

consistent style. We don’t want it to appear to have been written by a bunch of disagreeing individuals.  
phong cách nhất quán. Chúng tôi không muốn nó có vẻ như được viết bởi một nhóm người không đồng ý.

When I started the FitNesse project back in 2002, I sat down with the team to work out a coding style. This took about 10 minutes. We decided where we’d put our braces, what our indent size would be, how we would name classes, variables, and methods, and so forth. Then we encoded those rules into the code formatter of our IDE and have stuck with them ever since. These were not the rules that I prefer; they were rules decided by the team. As a member of that team I followed them when writing code in the FitNesse project.  
Khi tôi bắt đầu dự án FitNesse vào năm 2002, tôi đã ngồi lại với nhóm để tìm ra một phong cách viết mã. Quá trình này mất khoảng 10 phút. Chúng tôi đã quyết định nơi chúng tôi sẽ đặt dấu ngoặc nhọn, kích thước thụt lề của chúng tôi sẽ là bao nhiêu, cách chúng tôi đặt tên cho các lớp, biến và phương thức, v.v. Sau đó, chúng tôi đã mã hóa các quy tắc đó vào trình định dạng mã của IDE và gắn bó với chúng kể từ đó. Đây không phải là những quy tắc mà tôi thích; chúng là những quy tắc do đội quyết định. Là một thành viên của nhóm đó, tôi đã theo dõi họ khi viết mã trong dự án FitNesse.

Remember, a good software system is composed of a set of documents that read nicely. They need to have a consistent and smooth style. The reader needs to be able to trust that the formatting gestures he or she has seen in one source ﬁle will mean the same thing in others. The last thing we want to do is add more complexity to the source code by writing it in a jumble of different individual styles.  
Hãy nhớ rằng, một hệ thống phần mềm tốt bao gồm một bộ tài liệu đọc tốt. Họ cần phải có một phong cách nhất quán và trơn tru. Người đọc cần có khả năng tin tưởng rằng các cử chỉ định dạng mà họ đã thấy trong một tệp nguồn sẽ có ý nghĩa tương tự ở những tệp khác. Điều cuối cùng chúng tôi muốn làm là tăng thêm độ phức tạp cho mã nguồn bằng cách viết nó theo một mớ hỗn độn các phong cách riêng lẻ khác nhau.

[**Uncle Bob’s Formatting Rules**](#_page_237_0)  
Quy tắc định dạng của chú Bob

The rules I use personally are very simple and are illustrated by the code in Listing 5-6. Consider this an example of how code makes the best coding standard document.  
Cá nhân tôi sử dụng các quy tắc rất đơn giản và được minh họa bằng đoạn mã trong Liệt kê 5-6. Hãy coi đây là một ví dụ về cách mã tạo ra tài liệu tiêu chuẩn viết mã tốt nhất.

<https://www.facebook.com/groups/phattrienlaptrinh>

**Uncle Bob’s Formatting Rules** 91  
Quy tắc định dạng của chú Bob 91

**Listing 5-6 CodeAnalyzer.java**  
Liệt kê 5-6 CodeAnalyzer.java

public class CodeAnalyzer implements JavaFileAnalysis { private int lineCount;

private int maxLineWidth; private int widestLineNumber;

private LineWidthHistogram lineWidthHistogram; private int totalChars;

public CodeAnalyzer() {

lineWidthHistogram = new LineWidthHistogram(); }

public static List<File> findJavaFiles(File parentDirectory) { List<File> files = new ArrayList<File>(); findJavaFiles(parentDirectory, files);

return files; }

private static void findJavaFiles(File parentDirectory, List<File> files) { for (File file : parentDirectory.listFiles()) {

if (file.getName().endsWith(".java")) files.add(file);

else if (file.isDirectory()) findJavaFiles(file, files);

} }

public void analyzeFile(File javaFile) throws Exception { BufferedReader br = new BufferedReader(new FileReader(javaFile)); String line;

while ((line = br.readLine()) != null) measureLine(line);

}

private void measureLine(String line) { lineCount++;

int lineSize = line.length(); totalChars += lineSize;

lineWidthHistogram.addLine(lineSize, lineCount); recordWidestLine(lineSize);

}

private void recordWidestLine(int lineSize) { if (lineSize > maxLineWidth) {

maxLineWidth = lineSize; widestLineNumber = lineCount;

} }

public int getLineCount() { return lineCount;

}

public int getMaxLineWidth() { return maxLineWidth;

}

<https://www.facebook.com/groups/phattrienlaptrinh>

92 **Chapter 5: Formatting**  
92 Chương 5: Định dạng

**Listing 5-6 (continued) CodeAnalyzer.java  
Liệt kê 5-6 (tiếp theo) CodeAnalyzer.java**

public int getWidestLineNumber() { return widestLineNumber;

}

public LineWidthHistogram getLineWidthHistogram() { return lineWidthHistogram;

}

public double getMeanLineWidth() { return (double)totalChars/lineCount;

}

public int getMedianLineWidth() {

Integer[] sortedWidths = getSortedWidths(); int cumulativeLineCount = 0;

for (int width : sortedWidths) { cumulativeLineCount += lineCountForWidth(width); if (cumulativeLineCount > lineCount/2)

return width; }

throw new Error("Cannot get here"); }

private int lineCountForWidth(int width) {

return lineWidthHistogram.getLinesforWidth(width).size(); }

private Integer[] getSortedWidths() {

Set<Integer> widths = lineWidthHistogram.getWidths(); Integer[] sortedWidths = (widths.toArray(new Integer[0])); Arrays.sort(sortedWidths);

return sortedWidths; }

}

<https://www.facebook.com/groups/phattrienlaptrinh>



[**6**](#_page_237_0)

[**Objects and Data Structures**](#_page_237_0)  
Đối tượng và cấu trúc dữ liệu

There is a reason that we keep our variables private. We don’t want anyone else to depend on them. We want to keep the freedom to change their type or implementation on a whim or an impulse. Why, then, do so many programmers automatically add getters and setters to their objects, exposing their private variables as if they were public?  
Có một lý do mà chúng tôi giữ các biến của mình ở chế độ riêng tư. Chúng tôi không muốn bất cứ ai khác phụ thuộc vào họ. Chúng tôi muốn giữ quyền tự do thay đổi loại hoặc triển khai của họ theo ý thích hoặc sự thúc đẩy. Vậy thì tại sao nhiều lập trình viên lại tự động thêm getters và setters vào đối tượng của họ, để lộ các biến riêng tư của họ như thể chúng là công khai?

[**Data Abstraction**](#_page_237_0)  
Trừu tượng dữ liệu

Consider the difference between Listing 6-1 and Listing 6-2. Both represent the data of a point on the Cartesian plane. And yet one exposes its implementation and the other com-pletely hides it.  
Hãy xem xét sự khác biệt giữa Liệt kê 6-1 và Liệt kê 6-2. Cả hai đều biểu diễn dữ liệu của một điểm trên mặt phẳng Descartes. Tuy nhiên, một cái phơi bày việc triển khai nó và cái kia hoàn toàn che giấu nó.

93

<https://www.facebook.com/groups/phattrienlaptrinh>

94 **Chapter 6: Objects and Data Structures**  
94 Chương 6: Đối tượng và cấu trúc dữ liệu

**Listing 6-1 Concrete Point  
Liệt kê 6-1 Điểm cụ thể**

public class Point { public double x; public double y;

}

**Listing 6-2 Abstract Point**  
Liệt kê 6-2 Điểm trừu tượng

public interface Point { double getX();

double getY();

void setCartesian(double x, double y); double getR();

double getTheta();

void setPolar(double r, double theta); }

The beautiful thing about Listing 6-2 is that there is no way you can tell whether the implementation is in rectangular or polar coordinates. It might be neither! And yet the interface still unmistakably represents a data structure.  
Điều thú vị về Liệt kê 6-2 là không có cách nào để bạn có thể biết liệu việc triển khai là theo tọa độ vuông góc hay cực. Nó có thể là không! Tuy nhiên, giao diện vẫn thể hiện rõ ràng một cấu trúc dữ liệu.

But it represents more than just a data structure. The methods enforce an access policy.You can read the individual coordinates independently, but you must set the coordi-nates together as an atomic operation.  
Nhưng nó không chỉ đại diện cho một cấu trúc dữ liệu. Các phương thức thực thi chính sách truy cập. Bạn có thể đọc các tọa độ riêng lẻ một cách độc lập, nhưng bạn phải đặt các tọa độ cùng nhau dưới dạng thao tác nguyên tử.

Listing 6-1, on the other hand, is very clearly implemented in rectangular coordinates, and it forces us to manipulate those coordinates independently. This exposes implementa-tion. Indeed, it would expose implementation even if the variables were private and we were using single variable getters and setters.  
Mặt khác, Liệt kê 6-1 được triển khai rất rõ ràng trong các tọa độ hình chữ nhật và nó buộc chúng ta phải thao tác các tọa độ đó một cách độc lập. Điều này cho thấy việc thực hiện. Thật vậy, nó sẽ hiển thị việc triển khai ngay cả khi các biến là riêng tư và chúng tôi đang sử dụng các trình khởi tạo và trình đặt biến đơn lẻ.

Hiding implementation is not just a matter of putting a layer of functions between the variables. Hiding implementation is about abstractions! A class does not simply push its variables out through getters and setters. Rather it exposes abstract interfaces that allow its users to manipulate the *essence* of the data, without having to know its implementation.  
Ẩn việc triển khai không chỉ là vấn đề đặt một lớp chức năng giữa các biến. Ẩn thực hiện là về trừu tượng! Một lớp không chỉ đơn giản là đẩy các biến của nó ra thông qua getters và setters. Thay vào đó, nó hiển thị các giao diện trừu tượng cho phép người dùng thao tác với bản chất của dữ liệu mà không cần phải biết việc triển khai nó.

Consider Listing 6-3 and Listing 6-4. The ﬁrst uses concrete terms to communicate the fuel level of a vehicle, whereas the second does so with the abstraction of percentage. In the concrete case you can be pretty sure that these are just accessors of variables. In the abstract case you have no clue at all about the form of the data.  
Xem xét Liệt kê 6-3 và Liệt kê 6-4. Cái đầu tiên sử dụng các thuật ngữ cụ thể để truyền đạt mức nhiên liệu của một chiếc xe, trong khi cái thứ hai làm như vậy với phần trăm trừu tượng. Trong trường hợp cụ thể, bạn có thể khá chắc chắn rằng đây chỉ là những bộ truy cập của các biến. Trong trường hợp trừu tượng, bạn không có manh mối nào về dạng dữ liệu.

**Listing 6-3 Concrete Vehicle  
Liệt kê 6-3 Xe bê tông**

public interface Vehicle {

double getFuelTankCapacityInGallons(); double getGallonsOfGasoline();

}

<https://www.facebook.com/groups/phattrienlaptrinh>

**Data/Object Anti-Symmetry** 95  
Chống đối xứng dữ liệu/đối tượng 95

**Listing 6-4 Abstract Vehicle**  
Liệt kê 6-4 Phương tiện Trừu tượng

public interface Vehicle {

double getPercentFuelRemaining(); }

In both of the above cases the second option is preferable. We do not want to expose the details of our data. Rather we want to express our data in abstract terms. This is not merely accomplished by using interfaces and/or getters and setters. Serious thought needs to be put into the best way to represent the data that an object contains. The worst option is to blithely add getters and setters.  
Trong cả hai trường hợp trên, tùy chọn thứ hai là thích hợp hơn. Chúng tôi không muốn tiết lộ chi tiết dữ liệu của mình. Thay vào đó, chúng tôi muốn thể hiện dữ liệu của mình dưới dạng trừu tượng. Điều này không chỉ được thực hiện bằng cách sử dụng giao diện và/hoặc getters và setters. Suy nghĩ nghiêm túc cần được đưa vào cách tốt nhất để biểu diễn dữ liệu mà một đối tượng chứa. Tùy chọn tồi tệ nhất là vô tình thêm getters và setters.

[**Data/Object Anti-Symmetry**](#_page_237_0)  
Chống đối xứng dữ liệu/đối tượng

These two examples show the difference between objects and data structures. Objects hide their data behind abstractions and expose functions that operate on that data. Data struc-ture expose their data and have no meaningful functions. Go back and read that again. Notice the complimentary nature of the two deﬁnitions. They are virtual opposites. This difference may seem trivial, but it has far-reaching implications.  
Hai ví dụ này cho thấy sự khác biệt giữa đối tượng và cấu trúc dữ liệu. Các đối tượng ẩn dữ liệu của chúng đằng sau các trừu tượng và hiển thị các chức năng hoạt động trên dữ liệu đó. Cấu trúc dữ liệu phơi bày dữ liệu của họ và không có chức năng có ý nghĩa. Quay trở lại và đọc nó một lần nữa. Lưu ý bản chất miễn phí của hai định nghĩa. Họ là đối lập ảo. Sự khác biệt này có vẻ tầm thường, nhưng nó có ý nghĩa sâu rộng.

Consider, for example, the procedural shape example in Listing 6-5. The Geometry class operates on the three shape classes. The shape classes are simple data structures without any behavior. All the behavior is in the Geometry class.  
Ví dụ, hãy xem xét ví dụ về hình dạng thủ tục trong Liệt kê 6-5. Lớp Hình học hoạt động trên ba lớp hình dạng. Các lớp hình dạng là các cấu trúc dữ liệu đơn giản mà không có bất kỳ hành vi nào. Tất cả các hành vi là trong lớp Hình học.

**Listing 6-5 Procedural Shape  
Liệt kê 6-5 Hình dạng thủ tục**

public class Square { public Point topLeft; public double side;

}

public class Rectangle { public Point topLeft; public double height; public double width;

}

public class Circle { public Point center; public double radius;

}

public class Geometry {

public final double PI = 3.141592653589793;

public double area(Object shape) throws NoSuchShapeException {

if (shape instanceof Square) { Square s = (Square)shape; return s.side \* s.side;

}

<https://www.facebook.com/groups/phattrienlaptrinh>

96 **Chapter 6: Objects and Data Structures**  
96 Chương 6: Đối tượng và cấu trúc dữ liệu

**Listing 6-5 (continued) Procedural Shape**  
Liệt kê 6-5 (tiếp theo) Hình dạng thủ tục

else if (shape instanceof Rectangle) { Rectangle r = (Rectangle)shape; return r.height \* r.width;

}

else if (shape instanceof Circle) { Circle c = (Circle)shape;

return PI \* c.radius \* c.radius; }

throw new NoSuchShapeException(); }

}

Object-oriented programmers might wrinkle their noses at this and complain that it is procedural—and they’d be right. But the sneer may not be warranted. Consider what would happen if a perimeter() function were added to Geometry. The shape classes would be unaffected! Any other classes that depended upon the shapes would also be unaffected! On the other hand, if I add a new shape, I must change all the functions in Geometryto deal with it. Again, read that over. Notice that the two conditions are diametrically opposed.  
Các lập trình viên hướng đối tượng có thể nhăn mũi trước điều này và phàn nàn rằng đó là thủ tục—và họ đã đúng. Nhưng sự chế nhạo có thể không được bảo hành. Hãy xem điều gì sẽ xảy ra nếu một hàm Perimeter() được thêm vào Geometry. Các lớp hình dạng sẽ không bị ảnh hưởng! Bất kỳ lớp nào khác phụ thuộc vào hình dạng cũng sẽ không bị ảnh hưởng! Mặt khác, nếu tôi thêm một hình dạng mới, tôi phải thay đổi tất cả các chức năng trong Hình học để xử lý nó. Một lần nữa, đọc nó qua. Lưu ý rằng hai điều kiện hoàn toàn trái ngược nhau.

Now consider the object-oriented solution in Listing 6-6. Here the area()method is polymorphic. No Geometry class is necessary. So if I add a new shape, none of the existing *functions* are affected, but if I add a new function all of the *shapes* must be changed!1  
Bây giờ hãy xem xét giải pháp hướng đối tượng trong Liệt kê 6-6. Ở đây, phương thức area() là đa hình. Không có lớp Hình học là cần thiết. Vì vậy, nếu tôi thêm một hình dạng mới, không có chức năng hiện có nào bị ảnh hưởng, nhưng nếu tôi thêm một chức năng mới thì tất cả các hình dạng đó phải được thay đổi!1

**Listing 6-6 Polymorphic Shapes  
Liệt kê 6-6 Hình dạng đa hình**

public class Square implements Shape { private Point topLeft;

private double side;

public double area() { return side\*side;

} }

public class Rectangle implements Shape { private Point topLeft;

private double height; private double width;

public double area() { return height \* width;

} }

1. There are ways around this that are well known to experienced object-oriented designers: VISITOR, or dual-dispatch, for example. But these techniques carry costs of their own and generally return the structure to that of a procedural program.  
1. Có nhiều cách giải quyết vấn đề này mà các nhà thiết kế hướng đối tượng có kinh nghiệm đã biết rõ: VISITOR hoặc công văn kép chẳng hạn. Nhưng những kỹ thuật này có chi phí của riêng chúng và thường đưa cấu trúc trở lại cấu trúc của một chương trình thủ tục.

<https://www.facebook.com/groups/phattrienlaptrinh>

**The Law of Demeter** 97  
Luật Demeter 97

**Listing 6-6 (continued)**  
Liệt kê 6-6 (tiếp theo)

**Polymorphic Shapes  
Hình dạng đa hình**

public class Circle implements Shape { private Point center;

private double radius;

public final double PI = 3.141592653589793;

public double area() {

return PI \* radius \* radius; }

}

Again, we see the complimentary nature of these two deﬁnitions; they are virtual opposites! This exposes the fundamental dichotomy between objects and data structures:  
Một lần nữa, chúng ta thấy bản chất miễn phí của hai định nghĩa này; họ là đối lập ảo! Điều này cho thấy sự phân đôi cơ bản giữa các đối tượng và cấu trúc dữ liệu:

*Procedural code (code using data structures) makes it easy to add new functions without changing the existing data structures. OO code, on the other hand, makes it easy to add new classes without changing existing functions.  
Mã thủ tục (mã sử dụng cấu trúc dữ liệu) giúp dễ dàng thêm chức năng mới mà không thay đổi cấu trúc dữ liệu hiện có. Mặt khác, mã OO giúp dễ dàng thêm các lớp mới mà không thay đổi các chức năng hiện có.*

The complement is also true:  
Phần bổ sung cũng đúng:

*Procedural code makes it hard to add new data structures because all the functions must change. OO code makes it hard to add new functions because all the classes must change.  
Mã thủ tục khiến việc thêm cấu trúc dữ liệu mới trở nên khó khăn vì tất cả các hàm phải thay đổi. Mã OO khiến việc thêm các chức năng mới trở nên khó khăn vì tất cả các lớp phải thay đổi.*

So, the things that are hard for OO are easy for procedures, and the things that are hard for procedures are easy for OO!  
Vì vậy, những điều khó đối với OO thì dễ về thủ tục, và những điều khó về thủ tục thì dễ đối với OO!

In any complex system there are going to be times when we want to add new data types rather than new functions. For these cases objects and OO are most appropriate. On the other hand, there will also be times when we’ll want to add new functions as opposed to data types. In that case procedural code and data structures will be more appropriate.  
Trong bất kỳ hệ thống phức tạp nào, sẽ có lúc chúng ta muốn thêm các loại dữ liệu mới thay vì các chức năng mới. Đối với những trường hợp này, các đối tượng và OO là thích hợp nhất. Mặt khác, cũng sẽ có lúc chúng ta muốn thêm các hàm mới thay vì các kiểu dữ liệu. Trong trường hợp đó, mã thủ tục và cấu trúc dữ liệu sẽ phù hợp hơn.

Mature programmers know that the idea that everything is an object *is a myth*. Some-times you really *do* want simple data structures with procedures operating on them.  
Các lập trình viên trưởng thành biết rằng ý tưởng rằng mọi thứ đều là một đối tượng là một huyền thoại. Đôi khi bạn thực sự muốn cấu trúc dữ liệu đơn giản với các thủ tục hoạt động trên chúng.

[**The Law of Demeter**](#_page_237_0)  
Định luật Demeter

There is a well-known heuristic called the *Law of Demeter*2 that says a module should not know about the innards of the *objects* it manipulates. As we saw in the last section, objects hide their data and expose operations. This means that an object should not expose its internal structure through accessors because to do so is to expose, rather than to hide, its internal structure.  
Có một kinh nghiệm nổi tiếng được gọi là Định luật Demeter2 nói rằng một mô-đun không nên biết về các bộ phận bên trong của các đối tượng mà nó thao tác. Như chúng ta đã thấy trong phần trước, các đối tượng ẩn dữ liệu của chúng và để lộ các hoạt động. Điều này có nghĩa là một đối tượng không nên để lộ cấu trúc bên trong của nó thông qua các bộ truy cập vì làm như vậy là để lộ ra, chứ không phải để che giấu, cấu trúc bên trong của nó.

More precisely, the Law of Demeter says that a method *f* of a class *C* should only call the methods of these:  
Chính xác hơn, Định luật Demeter nói rằng một phương thức f của lớp C chỉ nên gọi các phương thức sau:

*• C  
• C*

• An object created by *f*  
• Một đối tượng được tạo bởi f

2. <http://en.wikipedia.org/wiki/Law_of_Demeter>

<https://www.facebook.com/groups/phattrienlaptrinh>

98 **Chapter 6: Objects and Data Structures**  
98 Chương 6: Đối tượng và cấu trúc dữ liệu

• An object passed as an argument to *f*  
• Một đối tượng được truyền làm đối số cho f

• An object held in an instance variable of *C*  
• Một đối tượng được giữ trong một biến thể hiện của C

The method should *not* invoke methods on objects that are returned by any of the allowed functions. In other words, talk to friends, not to strangers.  
Phương thức này không được gọi các phương thức trên các đối tượng được trả về bởi bất kỳ hàm nào được phép. Nói cách khác, hãy nói chuyện với bạn bè, không phải với người lạ.

The following code3 appears to violate the Law of Demeter (among other things) because it calls the getScratchDir()function on the return value of getOptions()and then calls getAbsolutePath() on the return value of getScratchDir().  
Code3 sau đây dường như vi phạm Luật Demeter (trong số những thứ khác) vì nó gọi hàm getScratchDir() trên giá trị trả về của getOptions() và sau đó gọi getAbsolutePath() trên giá trị trả về của getScratchDir().

final String outputDir = ctxt.getOptions().getScratchDir().getAbsolutePath();

[**Train Wrecks**](#_page_237_0)  
Xác tàu bị đắm

This kind of code is often called a *train wreck* because it look like a bunch of coupled train cars. Chains of calls like this are generally considered to be sloppy style and should be avoided [G36]. It is usually best to split them up as follows:  
Loại mã này thường được gọi là xác tàu hỏa vì nó trông giống như một loạt các toa tàu được ghép nối. Chuỗi cuộc gọi như thế này thường được coi là phong cách cẩu thả và nên tránh [G36]. Thông thường tốt nhất là chia chúng ra như sau:

Options opts = ctxt.getOptions();

File scratchDir = opts.getScratchDir();

final String outputDir = scratchDir.getAbsolutePath();

Are these two snippets of code viola-tions of the Law of Demeter? Certainly the containing module knows that the ctxtobject contains options, which con-tain a scratch directory, which has an absolute path. That’s a lot of knowledge for one function to know. The calling function knows how to navigate through a lot of different objects.  
Đây có phải là hai đoạn mã vi phạm Luật Demeter không? Chắc chắn mô-đun chứa biết rằng ctxtobject chứa các tùy chọn, chứa một thư mục đầu, có một đường dẫn tuyệt đối. Đó là rất nhiều kiến thức cho một chức năng để biết. Hàm gọi biết cách điều hướng qua rất nhiều đối tượng khác nhau.

Whether this is a violation of Demeter depends on whether or not ctxt, Options, and ScratchDir are objects or data structures. If they are objects, then their internal structure should be hidden rather than exposed, and so knowledge of their innards is a clear viola-tion of the Law of Demeter. On the other hand, if ctxt, Options, and ScratchDirare just data structures with no behavior, then they naturally expose their internal structure, and so Demeter does not apply.  
Việc này có vi phạm Demeter hay không tùy thuộc vào việc ctxt, Options và ScratchDir có phải là đối tượng hoặc cấu trúc dữ liệu hay không. Nếu chúng là đồ vật, thì cấu trúc bên trong của chúng nên được ẩn đi thay vì để lộ ra ngoài, và vì vậy việc biết về các bộ phận bên trong của chúng rõ ràng là vi phạm Định luật Demeter. Mặt khác, nếu ctxt, Options và ScratchDirare chỉ là cấu trúc dữ liệu không có hành vi, thì chúng sẽ tự nhiên để lộ cấu trúc bên trong và do đó Demeter không áp dụng.

The use of accessor functions confuses the issue. If the code had been written as fol-lows, then we probably wouldn’t be asking about Demeter violations.  
Việc sử dụng các chức năng truy cập gây nhầm lẫn vấn đề. Nếu mã đã được viết dưới dạng các thông tin sau, thì có lẽ chúng tôi sẽ không hỏi về các vi phạm của Demeter.

final String outputDir = ctxt.options.scratchDir.absolutePath;

This issue would be a lot less confusing if data structures simply had public variables and no functions, whereas objects had private variables and public functions. However,  
Vấn đề này sẽ ít gây nhầm lẫn hơn nếu cấu trúc dữ liệu chỉ có các biến chung và không có hàm, trong khi các đối tượng có các biến riêng và hàm chung. Tuy nhiên,

3. Found somewhere in the apache framework.  
3. Tìm thấy ở đâu đó trong khung apache.

<https://www.facebook.com/groups/phattrienlaptrinh>

**The Law of Demeter** 99  
Luật Demeter 99

there are frameworks and standards (e.g., “beans”) that demand that even simple data structures have accessors and mutators.  
có các khung và tiêu chuẩn (ví dụ: “bean”) yêu cầu ngay cả các cấu trúc dữ liệu đơn giản cũng phải có bộ truy cập và bộ biến đổi.

[**Hybrids**](#_page_237_0)  
giống lai

This confusion sometimes leads to unfortunate hybrid structures that are half object and half data structure. They have functions that do signiﬁcant things, and they also have either public variables or public accessors and mutators that, for all intents and purposes, make the private variables public, tempting other external functions to use those variables the way a procedural program would use a data structure.4  
Sự nhầm lẫn này đôi khi dẫn đến các cấu trúc lai đáng tiếc là cấu trúc nửa đối tượng và nửa cấu trúc dữ liệu. Chúng có các hàm thực hiện những việc quan trọng và chúng cũng có các biến công khai hoặc bộ truy cập và bộ biến đổi công khai, với mọi ý định và mục đích, đặt các biến riêng tư thành công khai, cám dỗ các hàm bên ngoài khác sử dụng các biến đó theo cách mà một chương trình thủ tục sẽ sử dụng một cấu trúc dữ liệu.4

Such hybrids make it hard to add new functions but also make it hard to add new data structures. They are the worst of both worlds. Avoid creating them. They are indicative of a muddled design whose authors are unsure of—or worse, ignorant of—whether they need protection from functions or types.  
Các kết hợp như vậy gây khó khăn cho việc thêm các chức năng mới nhưng cũng gây khó khăn cho việc thêm các cấu trúc dữ liệu mới. Họ là những người tồi tệ nhất của cả hai thế giới. Tránh tạo ra chúng. Chúng là dấu hiệu của một thiết kế lộn xộn mà các tác giả không chắc chắn—hoặc tệ hơn là không biết—liệu chúng có cần được bảo vệ khỏi các chức năng hoặc loại hay không.

[**Hiding Structure**](#_page_237_0)  
cấu trúc ẩn

What if ctxt, options, and scratchDir are objects with real behavior? Then, because objects are supposed to hide their internal structure, we should not be able to navigate through them. How then would we get the absolute path of the scratch directory?  
Điều gì sẽ xảy ra nếu ctxt, tùy chọn và scratchDir là các đối tượng có hành vi thực sự? Sau đó, vì các đối tượng được cho là che giấu cấu trúc bên trong của chúng, nên chúng ta không thể điều hướng qua chúng. Làm thế nào sau đó chúng ta sẽ có được đường dẫn tuyệt đối của thư mục đầu?

ctxt.getAbsolutePathOfScratchDirectoryOption(); or

ctx.getScratchDirectoryOption().getAbsolutePath()

The ﬁrst option could lead to an explosion of methods in the ctxt object. The second pre-sumes that getScratchDirectoryOption()returns a data structure, not an object. Neither option feels good.  
Tùy chọn đầu tiên có thể dẫn đến sự bùng nổ các phương thức trong đối tượng ctxt. Tiền giả định thứ hai getScratchDirectoryOption() trả về cấu trúc dữ liệu, không phải đối tượng. Cả hai tùy chọn đều không cảm thấy tốt.

If ctxtis an object, we should be telling it to *do something;* we should not be asking it about its internals. So why did we want the absolute path of the scratch directory? What were we going to do with it? Consider this code from (many lines farther down in) the same module:  
Nếu ctxt là một đối tượng, chúng ta nên bảo nó làm gì đó; chúng ta không nên hỏi nó về nội bộ của nó. Vậy tại sao chúng ta lại muốn đường dẫn tuyệt đối của thư mục đầu? Chúng ta sẽ làm gì với nó? Xem xét mã này từ (nhiều dòng xa hơn trong) cùng một mô-đun:

String outFile = outputDir + "/" + className.replace('.', '/') + ".class"; FileOutputStream fout = new FileOutputStream(outFile); BufferedOutputStream bos = new BufferedOutputStream(fout);

The admixture of different levels of detail [G34][G6] is a bit troubling. Dots, slashes, ﬁle extensions, and Fileobjects should not be so carelessly mixed together, and mixed with the enclosing code. Ignoring that, however, we see that the intent of getting the abso-lute path of the scratch directory was to create a scratch ﬁle of a given name.  
Sự kết hợp của các mức độ chi tiết khác nhau [G34][G6] hơi khó hiểu. Các dấu chấm, dấu gạch chéo, phần mở rộng tệp và Đối tượng tệp không được trộn lẫn với nhau một cách bất cẩn và trộn lẫn với mã kèm theo. Tuy nhiên, bỏ qua điều đó, chúng ta thấy rằng mục đích của việc lấy đường dẫn tuyệt đối của thư mục scratch là để tạo một tệp cào có tên nhất định.

4. This is sometimes called Feature Envy from [Refactoring].  
4. Điều này đôi khi được gọi là Đố kỵ tính năng từ [Tái cấu trúc].

<https://www.facebook.com/groups/phattrienlaptrinh>

100 **Chapter 6: Objects and Data Structures**  
100 Chương 6: Đối tượng và cấu trúc dữ liệu

So, what if we told the ctxt object to do this?  
Vì vậy, điều gì sẽ xảy ra nếu chúng ta yêu cầu đối tượng ctxt thực hiện việc này?

BufferedOutputStream bos = ctxt.createScratchFileStream(classFileName);

That seems like a reasonable thing for an object to do! This allows ctxtto hide its internals and prevents the current function from having to violate the Law of Demeter by navigating through objects it shouldn’t know about.  
Đó có vẻ là một điều hợp lý cho một đối tượng để làm! Điều này cho phép ctxt ẩn phần bên trong của nó và ngăn chức năng hiện tại vi phạm Luật Demeter bằng cách điều hướng qua các đối tượng mà nó không nên biết.

[**Data Transfer Objects**](#_page_237_0)  
Đối tượng truyền dữ liệu

The quintessential form of a data structure is a class with public variables and no func-tions. This is sometimes called a data transfer object, or DTO. DTOs are very useful struc-tures, especially when communicating with databases or parsing messages from sockets, and so on. They often become the ﬁrst in a series of translation stages that convert raw data in a database into objects in the application code.  
Hình thức tinh túy của cấu trúc dữ liệu là một lớp có các biến công khai và không có chức năng. Điều này đôi khi được gọi là đối tượng truyền dữ liệu hoặc DTO. DTO là cấu trúc rất hữu ích, đặc biệt là khi giao tiếp với cơ sở dữ liệu hoặc phân tích cú pháp tin nhắn từ ổ cắm, v.v. Chúng thường trở thành giai đoạn đầu tiên trong một loạt các giai đoạn dịch chuyển đổi dữ liệu thô trong cơ sở dữ liệu thành các đối tượng trong mã ứng dụng.

Somewhat more common is the “bean” form shown in Listing 6-7. Beans have private variables manipulated by getters and setters. The quasi-encapsulation of beans seems to make some OO purists feel better but usually provides no other beneﬁt.  
Phổ biến hơn một chút là dạng “bean” được hiển thị trong Liệt kê 6-7. Đậu có các biến riêng được thao tác bởi getters và setters. Việc bán đóng gói đậu dường như làm cho một số người theo chủ nghĩa thuần túy OO cảm thấy tốt hơn nhưng thường không mang lại lợi ích nào khác.

**Listing 6-7 address.java**  
Liệt kê 6-7 address.java

public class Address { private String street; private String streetExtra; private String city; private String state; private String zip;

public Address(String street, String streetExtra, String city, String state, String zip) {

this.street = street; this.streetExtra = streetExtra; this.city = city;

this.state = state; this.zip = zip;

}

public String getStreet() { return street;

}

public String getStreetExtra() { return streetExtra;

}

public String getCity() { return city;

}

<https://www.facebook.com/groups/phattrienlaptrinh>

**Bibliography** 101  
Thư mục 101

**Listing 6-7 (continued) address.java  
Liệt kê 6-7 (tiếp theo) address.java**

public String getState() { return state;

}

public String getZip() { return zip;

} }

[**Active Record**](#_page_237_0)  
Bản ghi hoạt động

Active Records are special forms of DTOs. They are data structures with public (or bean-accessed) variables; but they typically have navigational methods like save and find. Typi-cally these Active Records are direct translations from database tables, or other data sources.  
Bản ghi Hoạt động là các dạng DTO đặc biệt. Chúng là các cấu trúc dữ liệu với các biến công khai (hoặc được truy cập bằng bean); nhưng chúng thường có các phương thức điều hướng như lưu và tìm. Thông thường, các Bản ghi Hoạt động này là bản dịch trực tiếp từ các bảng cơ sở dữ liệu hoặc các nguồn dữ liệu khác.

Unfortunately we often ﬁnd that developers try to treat these data structures as though they were objects by putting business rule methods in them. This is awkward because it creates a hybrid between a data structure and an object.  
Thật không may, chúng tôi thường thấy rằng các nhà phát triển cố gắng xử lý các cấu trúc dữ liệu này như thể chúng là các đối tượng bằng cách đưa các phương thức quy tắc kinh doanh vào chúng. Điều này thật khó xử vì nó tạo ra sự kết hợp giữa cấu trúc dữ liệu và đối tượng.

The solution, of course, is to treat the Active Record as a data structure and to create separate objects that contain the business rules and that hide their internal data (which are probably just instances of the Active Record).  
Tất nhiên, giải pháp là coi Bản ghi Hoạt động như một cấu trúc dữ liệu và tạo các đối tượng riêng biệt chứa các quy tắc kinh doanh và ẩn dữ liệu nội bộ của chúng (có thể chỉ là các phiên bản của Bản ghi Hoạt động).

[**Conclusion**](#_page_237_0)  
Phần kết luận

Objects expose behavior and hide data. This makes it easy to add new kinds of objects without changing existing behaviors. It also makes it hard to add new behaviors to existing objects. Data structures expose data and have no signiﬁcant behavior. This makes it easy to add new behaviors to existing data structures but makes it hard to add new data structures to existing functions.  
Các đối tượng phơi bày hành vi và ẩn dữ liệu. Điều này giúp dễ dàng thêm các loại đối tượng mới mà không thay đổi các hành vi hiện có. Nó cũng gây khó khăn cho việc thêm các hành vi mới vào các đối tượng hiện có. Cấu trúc dữ liệu phơi bày dữ liệu và không có hành vi quan trọng. Điều này giúp dễ dàng thêm các hành vi mới vào cấu trúc dữ liệu hiện có nhưng lại khó thêm cấu trúc dữ liệu mới vào các chức năng hiện có.

In any given system we will sometimes want the ﬂexibility to add new data types, and so we prefer objects for that part of the system. Other times we will want the ﬂexibility to add new behaviors, and so in that part of the system we prefer data types and procedures. Good software developers understand these issues without prejudice and choose the approach that is best for the job at hand.  
Trong bất kỳ hệ thống cụ thể nào, đôi khi chúng tôi sẽ muốn tính linh hoạt để thêm các loại dữ liệu mới và vì vậy chúng tôi thích các đối tượng cho phần đó của hệ thống. Những lần khác, chúng tôi sẽ muốn tính linh hoạt để thêm các hành vi mới và do đó, trong phần đó của hệ thống, chúng tôi thích các loại dữ liệu và thủ tục hơn. Các nhà phát triển phần mềm giỏi hiểu những vấn đề này mà không có thành kiến và chọn cách tiếp cận tốt nhất cho công việc hiện tại.

[**Bibliography**](#_page_237_0)  
Thư mục

**[Refactoring]:** *Refactoring: Improving the Design of Existing Code*, Martin Fowler et al., Addison-Wesley, 1999.  
[Tái cấu trúc]: Tái cấu trúc: Cải thiện thiết kế của mã hiện có, Martin Fowler và cộng sự, Addison-Wesley, 1999.

<https://www.facebook.com/groups/phattrienlaptrinh>

*This page intentionally left blank*

<https://www.facebook.com/groups/phattrienlaptrinh>



[**7**](#_page_311_0)

[**Error Handling**](#_page_311_0)  
Xử lý lỗi

by Michael Feathers  
bởi Michael Feathers

It might seem odd to have a section about error handling in a book about clean code. Error handling is just one of those things that we all have to do when we program. Input can be abnormal and devices can fail. In short, things can go wrong, and when they do, we as pro-grammers are responsible for making sure that our code does what it needs to do.  
Có vẻ lạ khi có một phần về xử lý lỗi trong một cuốn sách về mã sạch. Xử lý lỗi chỉ là một trong những điều mà tất cả chúng ta phải làm khi lập trình. Đầu vào có thể bất thường và thiết bị có thể bị lỗi. Nói tóm lại, mọi thứ có thể sai và khi chúng xảy ra, chúng tôi với tư cách là những người lập trình chuyên nghiệp có trách nhiệm đảm bảo rằng mã của chúng tôi thực hiện những gì nó cần làm.

The connection to clean code, however, should be clear. Many code bases are com-pletely dominated by error handling. When I say dominated, I don’t mean that error han-dling is all that they do. I mean that it is nearly impossible to see what the code does because of all of the scattered error handling. Error handling is important, *but if it obscures logic, it’s wrong*.  
Tuy nhiên, kết nối với mã sạch phải rõ ràng. Nhiều cơ sở mã bị chi phối hoàn toàn bởi việc xử lý lỗi. Khi tôi nói bị chi phối, ý tôi không phải là xử lý lỗi là tất cả những gì họ làm. Ý tôi là gần như không thể xem mã này làm gì vì tất cả các xử lý lỗi rải rác. Xử lý lỗi là quan trọng, nhưng nếu nó che khuất logic, thì đó là sai.

In this chapter I’ll outline a number of techniques and considerations that you can use to write code that is both clean and robust—code that handles errors with grace and style.  
Trong chương này, tôi sẽ phác thảo một số kỹ thuật và cân nhắc mà bạn có thể sử dụng để viết mã rõ ràng và mạnh mẽ—mã xử lý lỗi một cách khéo léo và phong cách.

103

<https://www.facebook.com/groups/phattrienlaptrinh>

104 **Chapter 7: Error Handling**  
104 Chương 7: Xử lý lỗi

[**Use Exceptions Rather Than Return Codes**](#_page_311_0)  
Sử dụng ngoại lệ thay vì trả lại mã

Back in the distant past there were many languages that didn’t have exceptions. In those languages the techniques for handling and reporting errors were limited. You either set an error ﬂag or returned an error code that the caller could check. The code in Listing 7-1 illustrates these approaches.  
Quay trở lại quá khứ xa xôi, có nhiều ngôn ngữ không có ngoại lệ. Trong các ngôn ngữ đó, các kỹ thuật xử lý và báo cáo lỗi bị hạn chế. Bạn có thể đặt một cờ báo lỗi hoặc trả về một mã lỗi mà người gọi có thể kiểm tra. Đoạn mã trong Liệt kê 7-1 minh họa các cách tiếp cận này.

**Listing 7-1 DeviceController.java  
Liệt kê 7-1 DeviceController.java**

public class DeviceController { ...

public void sendShutDown() { DeviceHandle handle = getHandle(DEV1); // Check the state of the device

if (handle != DeviceHandle.INVALID) {

// Save the device status to the record field retrieveDeviceRecord(handle);

// If not suspended, shut down

if (record.getStatus() != DEVICE\_SUSPENDED) { pauseDevice(handle); clearDeviceWorkQueue(handle); closeDevice(handle);

} else {

logger.log("Device suspended. Unable to shut down"); }

} else {

logger.log("Invalid handle for: " + DEV1.toString()); }

} ...

}

The problem with these approaches is that they clutter the caller. The caller must check for errors immediately after the call. Unfortunately, it’s easy to forget. For this rea-son it is better to throw an exception when you encounter an error. The calling code is cleaner. Its logic is not obscured by error handling.  
Vấn đề với những cách tiếp cận này là chúng làm lộn xộn người gọi. Người gọi phải kiểm tra lỗi ngay sau cuộc gọi. Thật không may, nó rất dễ quên. Vì lý do này, tốt hơn hết là ném một ngoại lệ khi bạn gặp lỗi. Mã gọi sạch hơn. Logic của nó không bị che khuất bởi việc xử lý lỗi.

Listing 7-2 shows the code after we’ve chosen to throw exceptions in methods that can detect errors.  
Liệt kê 7-2 hiển thị mã sau khi chúng ta đã chọn đưa ra các ngoại lệ trong các phương thức có thể phát hiện lỗi.

**Listing 7-2**

**DeviceController.java (with exceptions)  
DeviceController.java (có ngoại lệ)**

public class DeviceController { ...

public void sendShutDown() { try {

tryToShutDown();

} catch (DeviceShutDownError e) { logger.log(e);

} }

<https://www.facebook.com/groups/phattrienlaptrinh>

**Write Your Try-Catch-Finally Statement First** 105  
Viết câu lệnh Try-Catch-Finally đầu tiên 105

**Listing 7-2 (continued)**

**DeviceController.java (with exceptions)  
DeviceController.java (có ngoại lệ)**

private void tryToShutDown() throws DeviceShutDownError { DeviceHandle handle = getHandle(DEV1);

DeviceRecord record = retrieveDeviceRecord(handle);

pauseDevice(handle); clearDeviceWorkQueue(handle); closeDevice(handle);

}

private DeviceHandle getHandle(DeviceID id) { ...

throw new DeviceShutDownError("Invalid handle for: " + id.toString()); ...

}

... }

Notice how much cleaner it is. This isn’t just a matter of aesthetics. The code is better because two concerns that were tangled, the algorithm for device shutdown and error han-dling, are now separated. You can look at each of those concerns and understand them independently.  
Chú ý nó sạch hơn bao nhiêu. Đây không chỉ là vấn đề thẩm mỹ. Mã tốt hơn vì hai mối quan tâm bị rối, thuật toán tắt thiết bị và xử lý lỗi, giờ đã được tách biệt. Bạn có thể xem xét từng mối quan tâm đó và hiểu chúng một cách độc lập.

[**Write Your Try-Catch-Finally Statement First**](#_page_311_0)  
Viết câu lệnh Try-Catch-Finally của bạn trước

One of the most interesting things about exceptions is that they *deﬁne a scope* within your program. When you execute code in the tryportion of a try-catch-finallystatement, you are stating that execution can abort at any point and then resume at the catch.  
Một trong những điều thú vị nhất về ngoại lệ là chúng xác định phạm vi trong chương trình của bạn. Khi bạn thực thi mã trong phần thử của câu lệnh thử-bắt-cuối cùng, bạn đang tuyên bố rằng việc thực thi có thể hủy bỏ tại bất kỳ thời điểm nào và sau đó tiếp tục khi bắt.

In a way, tryblocks are like transactions. Your catchhas to leave your program in a consistent state, no matter what happens in the try. For this reason it is good practice to start with a try-catch-finally statement when you are writing code that could throw exceptions. This helps you deﬁne what the user of that code should expect, no matter what goes wrong with the code that is executed in the try.  
Theo một cách nào đó, tryblocks giống như các giao dịch. Lưu ý của bạn là phải để chương trình của bạn ở trạng thái nhất quán, bất kể điều gì xảy ra trong quá trình thử. Vì lý do này, nên bắt đầu với câu lệnh try-catch-cuối cùng khi bạn đang viết mã có thể đưa ra ngoại lệ. Điều này giúp bạn xác định những gì người dùng mã đó sẽ mong đợi, bất kể có vấn đề gì xảy ra với mã được thực thi trong lần thử.

Let’s look at an example. We need to write some code that accesses a ﬁle and reads some serialized objects.  
Hãy xem một ví dụ. Chúng ta cần viết một số mã truy cập tệp và đọc một số đối tượng được tuần tự hóa.

We start with a unit test that shows that we’ll get an exception when the ﬁle doesn’t exist:  
Chúng tôi bắt đầu với một bài kiểm tra đơn vị cho thấy rằng chúng tôi sẽ nhận được một ngoại lệ khi tệp không tồn tại:

@Test(expected = StorageException.class)

public void retrieveSectionShouldThrowOnInvalidFileName() { sectionStore.retrieveSection("invalid - file");

}

The test drives us to create this stub:  
Bài kiểm tra thúc đẩy chúng tôi tạo sơ khai này:

public List<RecordedGrip> retrieveSection(String sectionName) { // dummy return until we have a real implementation

return new ArrayList<RecordedGrip>(); }

<https://www.facebook.com/groups/phattrienlaptrinh>

106 **Chapter 7: Error Handling**  
106 Chương 7: Xử lý lỗi

Our test fails because it doesn’t throw an exception. Next, we change our implementa-tion so that it attempts to access an invalid ﬁle. This operation throws an exception:  
Thử nghiệm của chúng tôi không thành công vì nó không đưa ra ngoại lệ. Tiếp theo, chúng tôi thay đổi triển khai của mình để nó cố gắng truy cập một tệp không hợp lệ. Thao tác này đưa ra một ngoại lệ:

public List<RecordedGrip> retrieveSection(String sectionName) { try {

FileInputStream stream = new FileInputStream(sectionName) } catch (Exception e) {

throw new StorageException("retrieval error", e); }

return new ArrayList<RecordedGrip>(); }

Our test passes now because we’ve caught the exception. At this point, we can refac-tor. We can narrow the type of the exception we catch to match the type that is actually thrown from the FileInputStream constructor: FileNotFoundException:  
Bài kiểm tra của chúng tôi đã vượt qua vì chúng tôi đã bắt được ngoại lệ. Tại thời điểm này, chúng ta có thể tái cấu trúc. Chúng ta có thể thu hẹp loại ngoại lệ mà chúng ta bắt được để khớp với loại thực sự được ném ra từ hàm tạo FileInputStream: FileNotFoundException:

public List<RecordedGrip> retrieveSection(String sectionName) { try {

FileInputStream stream = new FileInputStream(sectionName); stream.close();

} catch (FileNotFoundException e) {

throw new StorageException("retrieval error”, e); }

return new ArrayList<RecordedGrip>(); }

Now that we’ve deﬁned the scope with a try-catchstructure, we can use TDD to build up the rest of the logic that we need. That logic will be added between the creation of the FileInputStream and the close, and can pretend that nothing goes wrong.  
Bây giờ chúng ta đã xác định phạm vi bằng cấu trúc thử bắt, chúng ta có thể sử dụng TDD để xây dựng phần logic còn lại mà chúng ta cần. Logic đó sẽ được thêm vào giữa quá trình tạo FileInputStream và đóng và có thể giả vờ rằng không có gì sai.

Try to write tests that force exceptions, and then add behavior to your handler to sat-isfy your tests. This will cause you to build the transaction scope of the tryblock ﬁrst and will help you maintain the transaction nature of that scope.  
Cố gắng viết các bài kiểm tra bắt buộc có ngoại lệ, sau đó thêm hành vi vào trình xử lý của bạn để đáp ứng các bài kiểm tra của bạn. Điều này sẽ khiến bạn xây dựng phạm vi giao dịch của tryblock trước và sẽ giúp bạn duy trì bản chất giao dịch của phạm vi đó.

[**Use Unchecked Exceptions**](#_page_311_0)  
Sử dụng ngoại lệ không được kiểm tra

The debate is over. For years Java programmers have debated over the beneﬁts and liabili-ties of checked exceptions. When checked exceptions were introduced in the ﬁrst version of Java, they seemed like a great idea. The signature of every method would list all of the exceptions that it could pass to its caller. Moreover, these exceptions were part of the type of the method. Your code literally wouldn’t compile if the signature didn’t match what your code could do.  
Cuộc tranh luận đã kết thúc. Trong nhiều năm, các lập trình viên Java đã tranh luận về lợi ích và trách nhiệm pháp lý của các ngoại lệ được kiểm tra. Khi các ngoại lệ được kiểm tra được giới thiệu trong phiên bản đầu tiên của Java, chúng có vẻ là một ý tưởng tuyệt vời. Chữ ký của mọi phương thức sẽ liệt kê tất cả các ngoại lệ mà nó có thể chuyển đến trình gọi của nó. Hơn nữa, những ngoại lệ này là một phần của loại phương thức. Mã của bạn thực sự sẽ không được biên dịch nếu chữ ký không khớp với những gì mã của bạn có thể làm.

At the time, we thought that checked exceptions were a great idea; and yes, they can yield *some* beneﬁt. However, it is clear now that they aren’t necessary for the production of robust software. C# doesn’t have checked exceptions, and despite valiant attempts, C++ doesn’t either. Neither do Python or Ruby.Yet it is possible to write robust software in all of these languages. Because that is the case, we have to decide—really—whether checked exceptions are worth their price.  
Vào thời điểm đó, chúng tôi nghĩ rằng các ngoại lệ được kiểm tra là một ý tưởng tuyệt vời; và vâng, chúng có thể mang lại một số lợi ích. Tuy nhiên, rõ ràng là chúng không cần thiết cho việc sản xuất phần mềm mạnh. C# không có các ngoại lệ được kiểm tra và mặc dù đã có những nỗ lực dũng cảm, C++ cũng vậy. Python hay Ruby cũng vậy. Tuy nhiên, có thể viết phần mềm mạnh bằng tất cả các ngôn ngữ này. Bởi vì trong trường hợp đó, chúng tôi phải quyết định—thực sự—liệu các ngoại lệ được kiểm tra có xứng đáng với mức giá của chúng hay không.

<https://www.facebook.com/groups/phattrienlaptrinh>

**Deﬁne Exception Classes in Terms of a Caller’s Needs** 107  
Xác định các lớp ngoại lệ theo nhu cầu của người gọi 107

What price? The price of checked exceptions is an Open/Closed Principle1 violation. If you throw a checked exception from a method in your code and the catch is three levels above, *you must declare that exception in the signature of each method between you and the catch*. This means that a change at a low level of the software can force signature changes on many higher levels. The changed modules must be rebuilt and redeployed, even though nothing they care about changed.  
Giá bao nhiêu? Giá của các trường hợp ngoại lệ được kiểm tra là vi phạm Nguyên tắc Mở/Đóng1. Nếu bạn đưa ra một ngoại lệ đã kiểm tra từ một phương thức trong mã của mình và giá trị bắt cao hơn ba cấp độ ở trên, thì bạn phải khai báo ngoại lệ đó trong chữ ký của từng phương thức giữa bạn và giá trị bắt. Điều này có nghĩa là một thay đổi ở cấp độ thấp của phần mềm có thể buộc thay đổi chữ ký ở nhiều cấp độ cao hơn. Các mô-đun đã thay đổi phải được xây dựng lại và triển khai lại, mặc dù không có gì thay đổi mà chúng quan tâm.

Consider the calling hierarchy of a large system. Functions at the top call functions below them, which call more functions below them, ad inﬁnitum. Now let’s say one of the lowest level functions is modiﬁed in such a way that it must throw an exception. If that exception is checked, then the function signature must add a throws clause. But this means that every function that calls our modiﬁed function must also be modiﬁed either to catch the new exception or to append the appropriate throwsclause to its signature. Ad inﬁnitum. The net result is a cascade of changes that work their way from the lowest levels of the software to the highest! Encapsulation is broken because all functions in the path of a throw must know about details of that low-level exception. Given that the purpose of exceptions is to allow you to handle errors at a distance, it is a shame that checked excep-tions break encapsulation in this way.  
Xem xét hệ thống phân cấp cuộc gọi của một hệ thống lớn. Các chức năng ở trên cùng gọi các chức năng bên dưới chúng, các chức năng này gọi nhiều chức năng hơn bên dưới chúng, đến vô tận. Bây giờ, giả sử một trong những hàm cấp thấp nhất được sửa đổi theo cách mà nó phải đưa ra một ngoại lệ. Nếu ngoại lệ đó được chọn, thì chữ ký hàm phải thêm mệnh đề ném. Nhưng điều này có nghĩa là mọi hàm gọi hàm đã sửa đổi của chúng ta cũng phải được sửa đổi để bắt ngoại lệ mới hoặc để thêm mệnh đề ném thích hợp vào chữ ký của nó. Quảng cáo vô tận. Kết quả cuối cùng là một loạt các thay đổi hoạt động theo cách của chúng từ mức thấp nhất của phần mềm đến mức cao nhất! Đóng gói bị hỏng vì tất cả các chức năng trong đường dẫn ném phải biết chi tiết về ngoại lệ cấp thấp đó. Vì mục đích của các ngoại lệ là cho phép bạn xử lý các lỗi ở khoảng cách xa, thật đáng tiếc khi các ngoại lệ được kiểm tra phá vỡ sự đóng gói theo cách này.

Checked exceptions can sometimes be useful if you are writing a critical library: You must catch them. But in general application development the dependency costs outweigh the beneﬁts.  
Các ngoại lệ được kiểm tra đôi khi có thể hữu ích nếu bạn đang viết một thư viện quan trọng: Bạn phải nắm bắt chúng. Nhưng trong phát triển ứng dụng nói chung, chi phí phụ thuộc lớn hơn lợi ích.

[**Provide Context with Exceptions**](#_page_311_0)  
Cung cấp bối cảnh với các ngoại lệ

Each exception that you throw should provide enough context to determine the source and location of an error. In Java, you can get a stack trace from any exception; however, a stack trace can’t tell you the intent of the operation that failed.  
Mỗi ngoại lệ mà bạn đưa ra phải cung cấp đủ ngữ cảnh để xác định nguồn và vị trí của lỗi. Trong Java, bạn có thể lấy dấu vết ngăn xếp từ bất kỳ ngoại lệ nào; tuy nhiên, dấu vết ngăn xếp không thể cho bạn biết ý định của thao tác không thành công.

Create informative error messages and pass them along with your exceptions. Men-tion the operation that failed and the type of failure. If you are logging in your application, pass along enough information to be able to log the error in your catch.  
Tạo các thông báo lỗi đầy đủ thông tin và chuyển chúng cùng với các ngoại lệ của bạn. Đề cập đến hoạt động thất bại và loại thất bại. Nếu bạn đang đăng nhập vào ứng dụng của mình, hãy chuyển đủ thông tin để có thể ghi lại lỗi trong phần khai thác của bạn.

[**Deﬁne Exception Classes in Terms of a Caller’s Needs**](#_page_311_0)  
Xác định các lớp ngoại lệ theo nhu cầu của người gọi

There are many ways to classify errors. We can classify them by their source: Did they come from one component or another? Or their type: Are they device failures, network failures, or programming errors? However, when we deﬁne exception classes in an appli-cation, our most important concern should be *how they are caught*.  
Có nhiều cách phân loại lỗi. Chúng ta có thể phân loại chúng theo nguồn: Chúng đến từ thành phần này hay thành phần khác? Hoặc loại của chúng: Chúng có phải là lỗi thiết bị, lỗi mạng hay lỗi lập trình không? Tuy nhiên, khi chúng ta định nghĩa các lớp ngoại lệ trong một ứng dụng, mối quan tâm quan trọng nhất của chúng ta là cách chúng bị bắt.

1. [Martin].

<https://www.facebook.com/groups/phattrienlaptrinh>

108 **Chapter 7: Error Handling**  
108 Chương 7: Xử lý lỗi

Let’s look at an example of poor exception classiﬁcation. Here is a try-catch-finally statement for a third-party library call. It covers all of the exceptions that the calls can throw:  
Hãy xem xét một ví dụ về phân loại ngoại lệ kém. Đây là câu lệnh thử-bắt-cuối cùng cho lệnh gọi thư viện của bên thứ ba. Nó bao gồm tất cả các ngoại lệ mà các cuộc gọi có thể đưa ra:

ACMEPort port = new ACMEPort(12);

try { port.open();

} catch (DeviceResponseException e) { reportPortError(e);

logger.log("Device response exception", e); } catch (ATM1212UnlockedException e) {

reportPortError(e); logger.log("Unlock exception", e);

} catch (GMXError e) { reportPortError(e);

logger.log("Device response exception"); } finally {

… }

That statement contains a lot of duplication, and we shouldn’t be surprised. In most exception handling situations, the work that we do is relatively standard regardless of the actual cause. We have to record an error and make sure that we can proceed.  
Tuyên bố đó có rất nhiều điểm trùng lặp và chúng ta không nên ngạc nhiên. Trong hầu hết các tình huống xử lý ngoại lệ, công việc mà chúng tôi thực hiện là tương đối chuẩn bất kể nguyên nhân thực sự là gì. Chúng tôi phải ghi lại lỗi và đảm bảo rằng chúng tôi có thể tiếp tục.

In this case, because we know that the work that we are doing is roughly the same regardless of the exception, we can simplify our code considerably by wrapping the API that we are calling and making sure that it returns a common exception type:  
Trong trường hợp này, vì chúng tôi biết rằng công việc mà chúng tôi đang thực hiện gần như giống nhau bất kể ngoại lệ, nên chúng tôi có thể đơn giản hóa mã của mình một cách đáng kể bằng cách gói API mà chúng tôi đang gọi và đảm bảo rằng nó trả về một loại ngoại lệ phổ biến:

LocalPort port = new LocalPort(12); try {

port.open();

} catch (PortDeviceFailure e) { reportError(e); logger.log(e.getMessage(), e);

} finally { …

}

Our LocalPortclass is just a simple wrapper that catches and translates exceptions thrown by the ACMEPort class:  
LocalPortclass của chúng tôi chỉ là một trình bao bọc đơn giản để bắt và dịch các ngoại lệ do lớp ACMEPort đưa ra:

public class LocalPort { private ACMEPort innerPort;

public LocalPort(int portNumber) { innerPort = new ACMEPort(portNumber);

}

public void open() { try {

innerPort.open();

} catch (DeviceResponseException e) { throw new PortDeviceFailure(e);

} catch (ATM1212UnlockedException e) { throw new PortDeviceFailure(e);

} catch (GMXError e) {

<https://www.facebook.com/groups/phattrienlaptrinh>

**Deﬁne the Normal Flow** 109  
Xác định luồng thông thường 109

throw new PortDeviceFailure(e); }

} …

}

Wrappers like the one we deﬁned for ACMEPortcan be very useful. In fact, wrapping third-party APIs is a best practice. When you wrap a third-party API, you minimize your dependencies upon it: You can choose to move to a different library in the future without much penalty. Wrapping also makes it easier to mock out third-party calls when you are testing your own code.  
Các hàm bao giống như cái mà chúng tôi đã xác định cho ACMEPort có thể rất hữu ích. Trên thực tế, gói API của bên thứ ba là phương pháp hay nhất. Khi bạn gói một API của bên thứ ba, bạn giảm thiểu sự phụ thuộc của mình vào nó: Bạn có thể chọn chuyển sang một thư viện khác trong tương lai mà không bị phạt nhiều. Gói cũng giúp bạn dễ dàng loại bỏ các cuộc gọi của bên thứ ba khi bạn đang kiểm tra mã của riêng mình.

One ﬁnal advantage of wrapping is that you aren’t tied to a particular vendor’s API design choices. You can deﬁne an API that you feel comfortable with. In the preceding example, we deﬁned a single exception type for portdevice failure and found that we could write much cleaner code.  
Một lợi thế cuối cùng của gói là bạn không bị ràng buộc với các lựa chọn thiết kế API của nhà cung cấp cụ thể. Bạn có thể xác định một API mà bạn cảm thấy thoải mái. Trong ví dụ trước, chúng tôi đã xác định một loại ngoại lệ duy nhất cho lỗi thiết bị cổng và thấy rằng chúng tôi có thể viết mã gọn gàng hơn nhiều.

Often a single exception class is ﬁne for a particular area of code. The information sent with the exception can distinguish the errors. Use different classes only if there are times when you want to catch one exception and allow the other one to pass through.  
Thường thì một lớp ngoại lệ duy nhất là tốt cho một vùng mã cụ thể. Thông tin được gửi với ngoại lệ có thể phân biệt các lỗi. Chỉ sử dụng các lớp khác nhau nếu có những lúc bạn muốn bắt một ngoại lệ và cho phép ngoại lệ khác đi qua.

[**Deﬁne the Normal Flow**](#_page_311_0)  
Xác định dòng chảy bình thường

If you follow the advice in the preceding sections, you’ll end up with a good amount of separation between your business logic and your error handling. The bulk of your code will start to look like a clean unadorned algorithm. However, the pro-cess of doing this pushes error detection to the edges of your program. You wrap external APIs so that you can throw your  
Nếu bạn làm theo lời khuyên trong các phần trước, bạn sẽ có được sự tách biệt tốt giữa logic nghiệp vụ và việc xử lý lỗi của mình. Phần lớn mã của bạn sẽ bắt đầu trông giống như một thuật toán sạch sẽ không trang trí. Tuy nhiên, quá trình thực hiện điều này đẩy khả năng phát hiện lỗi đến các cạnh của chương trình của bạn. Bạn bọc các API bên ngoài để bạn có thể ném

own exceptions, and you deﬁne a handler above your code so that you can deal with any aborted computation. Most of the time this is a great approach, but there are some times when you may not want to abort.  
các ngoại lệ riêng và bạn xác định một trình xử lý phía trên mã của mình để bạn có thể xử lý mọi tính toán bị hủy bỏ. Hầu hết thời gian đây là một cách tiếp cận tuyệt vời, nhưng có một số lúc bạn có thể không muốn bỏ thai.

Let’s take a look at an example. Here is some awkward code that sums expenses in a billing application:  
Hãy xem một ví dụ. Đây là một số mã khó tính tổng chi phí trong ứng dụng thanh toán:

try {

MealExpenses expenses = expenseReportDAO.getMeals(employee.getID()); m\_total += expenses.getTotal();

} catch(MealExpensesNotFound e) { m\_total += getMealPerDiem();

}

In this business, if meals are expensed, they become part of the total. If they aren’t, the employee gets a meal *per diem* amount for that day. The exception clutters the logic. Wouldn’t it be better if we didn’t have to deal with the special case? If we didn’t, our code would look much simpler. It would look like this:  
Trong lĩnh vực kinh doanh này, nếu các bữa ăn được tính vào chi phí, chúng sẽ trở thành một phần của tổng số tiền. Nếu không, nhân viên sẽ nhận được một bữa ăn công tác phí cho ngày hôm đó. Ngoại lệ làm lộn xộn logic. Sẽ không tốt hơn nếu chúng ta không phải đối phó với trường hợp đặc biệt? Nếu chúng tôi không làm như vậy, mã của chúng tôi sẽ trông đơn giản hơn nhiều. Nó sẽ trông như thế này:

MealExpenses expenses = expenseReportDAO.getMeals(employee.getID()); m\_total += expenses.getTotal();

<https://www.facebook.com/groups/phattrienlaptrinh>

110 **Chapter 7: Error Handling**  
110 Chương 7: Xử lý lỗi

Can we make the code that simple? It turns out that we can. We can change the ExpenseReportDAO so that it always returns a MealExpense object. If there are no meal expenses, it returns a MealExpense object that returns the *per diem* as its total:  
Chúng ta có thể làm cho mã đơn giản không? Nó chỉ ra rằng chúng ta có thể. Chúng ta có thể thay đổi ExpenseReportDAO để nó luôn trả về một đối tượng MealExpense. Nếu không có chi phí bữa ăn nào, nó trả về một đối tượng MealExpense trả về công tác phí dưới dạng tổng của nó:

public class PerDiemMealExpenses implements MealExpenses { public int getTotal() {

// return the per diem default }

}

This is called the SPECIAL CASE PATTERN [Fowler]. You create a class or conﬁgure an object so that it handles a special case for you. When you do, the client code doesn’t have to deal with exceptional behavior. That behavior is encapsulated in the special case object.  
Đây được gọi là MẪU TRƯỜNG HỢP ĐẶC BIỆT [Fowler]. Bạn tạo một lớp hoặc cấu hình một đối tượng để nó xử lý một trường hợp đặc biệt cho bạn. Khi bạn làm như vậy, mã máy khách không phải đối phó với hành vi đặc biệt. Hành vi đó được gói gọn trong đối tượng trường hợp đặc biệt.

[**Don’t Return Null**](#_page_311_0)  
Không trả về Null

I think that any discussion about error handling should include mention of the things we do that invite errors. The ﬁrst on the list is returning null. I can’t begin to count the number of applications I’ve seen in which nearly every other line was a check for null. Here is some example code:  
Tôi nghĩ rằng bất kỳ cuộc thảo luận nào về xử lý lỗi nên bao gồm việc đề cập đến những điều chúng tôi làm gây ra lỗi. Cái đầu tiên trong danh sách trả về null. Tôi không thể bắt đầu đếm số lượng ứng dụng mà tôi đã thấy trong đó gần như mọi dòng khác đều kiểm tra giá trị rỗng. Đây là một số mã ví dụ:

public void registerItem(Item item) { if (item != null) {

ItemRegistry registry = peristentStore.getItemRegistry(); if (registry != null) {

Item existing = registry.getItem(item.getID());

if (existing.getBillingPeriod().hasRetailOwner()) { existing.register(item);

} }

} }

If you work in a code base with code like this, it might not look all that bad to you, but it is bad! When we return null, we are essentially creating work for ourselves and foisting problems upon our callers. All it takes is one missing nullcheck to send an application spinning out of control.  
Nếu bạn làm việc trong một cơ sở mã với mã như thế này, nó có thể không tệ lắm đối với bạn, nhưng nó rất tệ! Khi chúng tôi trả về null, về cơ bản, chúng tôi đang tạo ra công việc cho chính mình và giải quyết các vấn đề cho người gọi của chúng tôi. Tất cả những gì nó cần là một lần kiểm tra null bị thiếu để gửi một ứng dụng vượt khỏi tầm kiểm soát.

Did you notice the fact that there wasn’t a null check in the second line of that nested ifstatement? What would have happened at runtime if persistentStorewere null? We would have had a NullPointerException at runtime, and either someone is catching NullPointerExceptionat the top level or they are not. Either way it’s *bad*. What exactly should you do in response to a NullPointerExceptionthrown from the depths of your appli-cation?  
Bạn có nhận thấy thực tế là không có dấu kiểm null nào trong dòng thứ hai của câu lệnh if lồng nhau đó không? Điều gì sẽ xảy ra trong thời gian chạy nếu liên tụcStore là null? Chúng tôi đã có một NullPulumException trong thời gian chạy và ai đó đang bắt NullPulumException ở cấp cao nhất hoặc họ không. Dù bằng cách nào thì nó cũng tệ. Chính xác thì bạn nên làm gì để đáp lại một NullPulumException được ném ra từ sâu bên trong ứng dụng của bạn?

It’s easy to say that the problem with the code above is that it is missing a nullcheck, but in actuality, the problem is that it has *too many*. If you are tempted to return nullfrom a method, consider throwing an exception or returning a SPECIAL CASE object instead. If you are calling a null-returning method from a third-party API, consider wrapping that method with a method that either throws an exception or returns a special case object.  
Thật dễ dàng để nói rằng vấn đề với đoạn mã trên là nó thiếu kiểm tra null, nhưng trên thực tế, vấn đề là nó có quá nhiều. Nếu bạn muốn trả về giá trị rỗng từ một phương thức, thay vào đó, hãy xem xét đưa ra một ngoại lệ hoặc trả về một đối tượng TRƯỜNG HỢP ĐẶC BIỆT. Nếu bạn đang gọi một phương thức trả về giá trị rỗng từ API của bên thứ ba, hãy cân nhắc gói phương thức đó bằng một phương thức đưa ra một ngoại lệ hoặc trả về một đối tượng trường hợp đặc biệt.

<https://www.facebook.com/groups/phattrienlaptrinh>

**Don’t Pass Null** 111  
Đừng vượt qua Null 111

In many cases, special case objects are an easy remedy. Imagine that you have code like this:  
Trong nhiều trường hợp, các đối tượng trường hợp đặc biệt là một biện pháp khắc phục dễ dàng. Hãy tưởng tượng rằng bạn có mã như thế này:

List<Employee> employees = getEmployees(); if (employees != null) {

for(Employee e : employees) { totalPay += e.getPay();

} }

Right now, getEmployeescan return null, but does it have to? If we change getEmployeeso that it returns an empty list, we can clean up the code:  
Ngay bây giờ, getEmployeescan trả về null, nhưng nó có phải không? Nếu chúng ta thay đổi getEmployeeso để nó trả về một danh sách trống, thì chúng ta có thể xóa mã:

List<Employee> employees = getEmployees(); for(Employee e : employees) {

totalPay += e.getPay(); }

Fortunately, Java has Collections.emptyList(), and it returns a predeﬁned immutable list that we can use for this purpose:  
May mắn thay, Java có Collections.emptyList(), và nó trả về một danh sách bất biến được xác định trước mà chúng ta có thể sử dụng cho mục đích này:

public List<Employee> getEmployees() { if( .. there are no employees .. )

return Collections.emptyList(); }

If you code this way, you will minimize the chance of NullPointerExceptionsand your code will be cleaner.  
Nếu bạn viết mã theo cách này, bạn sẽ giảm thiểu khả năng xảy ra Ngoại lệ NullPulum và mã của bạn sẽ sạch hơn.

[**Don’t Pass Null**](#_page_311_0)  
Không vượt qua Null

Returning nullfrom methods is bad, but passing nullinto methods is worse. Unless you are working with an API which expects you to pass null, you should avoid passing null in your code whenever possible.  
Trả lại các phương thức nullfrom là không tốt, nhưng chuyển các phương thức nullinto còn tệ hơn. Trừ khi bạn đang làm việc với một API yêu cầu bạn chuyển null, bạn nên tránh chuyển null vào mã của mình bất cứ khi nào có thể.

Let’s look at an example to see why. Here is a simple method which calculates a met-ric for two points:  
Hãy xem một ví dụ để biết tại sao. Đây là một phương pháp đơn giản tính toán met-ric cho hai điểm:

public class MetricsCalculator {

public double xProjection(Point p1, Point p2) { return (p2.x – p1.x) \* 1.5;

} …

}

What happens when someone passes null as an argument?  
Điều gì xảy ra khi ai đó chuyển null làm đối số?

calculator.xProjection(null, new Point(12, 13)); We’ll get a NullPointerException, of course.

How can we ﬁx it? We could create a new exception type and throw it:  
Làm thế nào chúng ta có thể fix nó? Chúng ta có thể tạo một loại ngoại lệ mới và ném nó:

public class MetricsCalculator {

<https://www.facebook.com/groups/phattrienlaptrinh>

112 **Chapter 7: Error Handling**  
112 Chương 7: Xử lý lỗi

public double xProjection(Point p1, Point p2) { if (p1 == null || p2 == null) {

throw InvalidArgumentException(

"Invalid argument for MetricsCalculator.xProjection"); }

return (p2.x – p1.x) \* 1.5; }

}

Is this better? It might be a little better than a nullpointer exception, but remember, we have to deﬁne a handler for InvalidArgumentException. What should the handler do? Is there any good course of action?  
Tốt hơn chưa? Nó có thể tốt hơn một chút so với ngoại lệ nullpointer, nhưng hãy nhớ rằng, chúng ta phải xác định một trình xử lý cho UnlimitedArgumentException. Người xử lý nên làm gì? Có bất kỳ quá trình hành động tốt?

There is another alternative. We could use a set of assertions:  
Có một sự thay thế khác. Chúng ta có thể sử dụng một tập hợp các xác nhận:

public class MetricsCalculator {

public double xProjection(Point p1, Point p2) { assert p1 != null : "p1 should not be null"; assert p2 != null : "p2 should not be null"; return (p2.x – p1.x) \* 1.5;

} }

It’s good documentation, but it doesn’t solve the problem. If someone passes null, we’ll still have a runtime error.  
Đó là tài liệu tốt, nhưng nó không giải quyết được vấn đề. Nếu ai đó vượt qua null, chúng tôi sẽ vẫn gặp lỗi thời gian chạy.

In most programming languages there is no good way to deal with a nullthat is passed by a caller accidentally. Because this is the case, the rational approach is to forbid passing nullby default. When you do, you can code with the knowledge that a nullin an argument list is an indication of a problem, and end up with far fewer careless mistakes.  
Trong hầu hết các ngôn ngữ lập trình, không có cách nào tốt để xử lý một null mà người gọi vô tình chuyển qua. Bởi vì đây là trường hợp, cách tiếp cận hợp lý là cấm chuyển null theo mặc định. Khi bạn làm như vậy, bạn có thể viết mã với kiến thức rằng một danh sách đối số nullin là một dấu hiệu của một vấn đề và kết thúc với ít lỗi bất cẩn hơn rất nhiều.

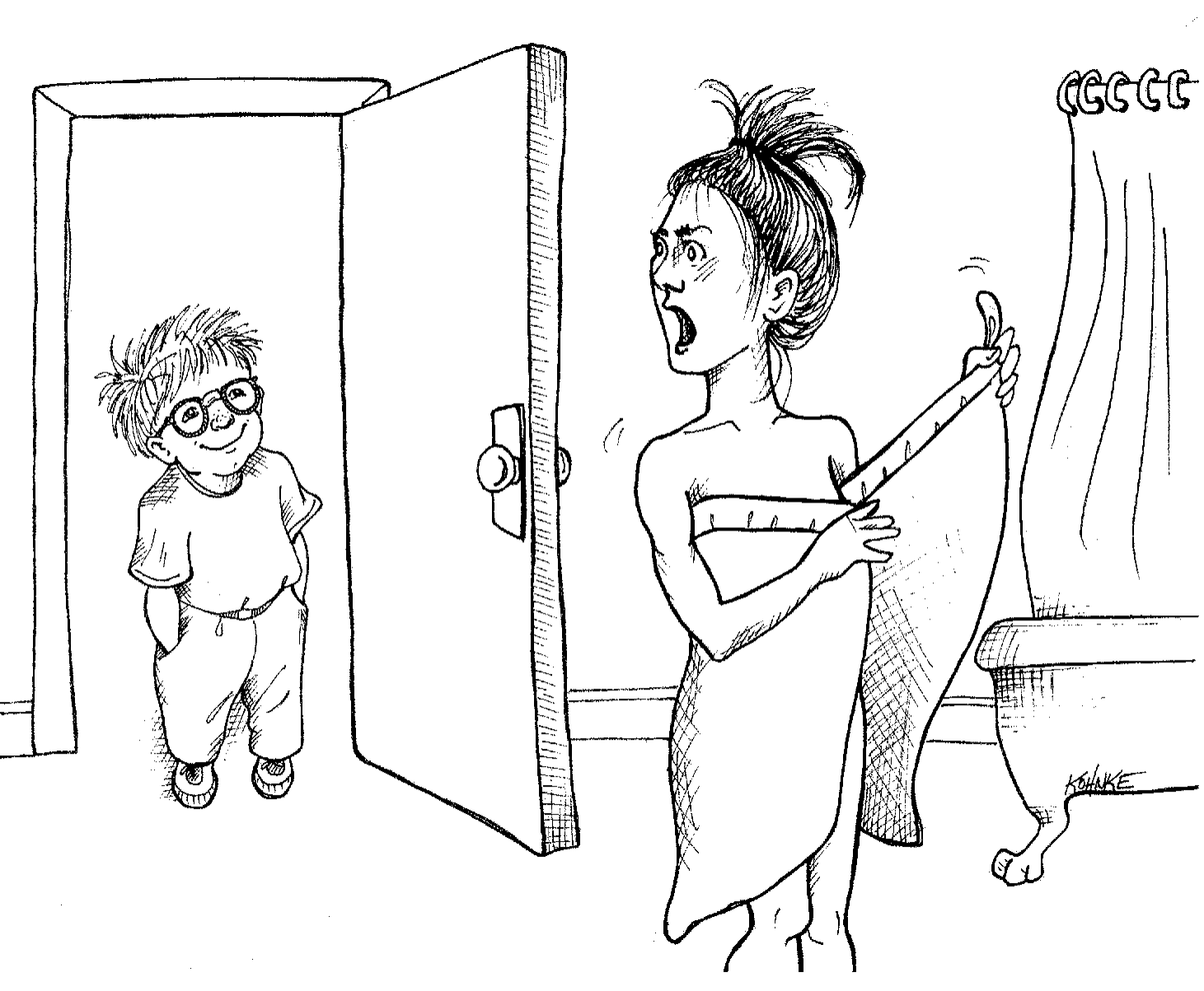
[**Conclusion**](#_page_311_0)  
Phần kết luận

Clean code is readable, but it must also be robust. These are not conﬂicting goals. We can write robust clean code if we see error handling as a separate concern, something that is viewable independently of our main logic. To the degree that we are able to do that, we can reason about it independently, and we can make great strides in the maintainability of our code.  
Mã sạch có thể đọc được, nhưng nó cũng phải mạnh mẽ. Đây không phải là những mục tiêu mâu thuẫn nhau. Chúng tôi có thể viết mã rõ ràng mạnh mẽ nếu chúng tôi coi việc xử lý lỗi là một mối quan tâm riêng biệt, thứ gì đó có thể xem được độc lập với logic chính của chúng tôi. Ở mức độ mà chúng tôi có thể làm điều đó, chúng tôi có thể suy luận về nó một cách độc lập và chúng tôi có thể đạt được những bước tiến lớn trong khả năng bảo trì mã của mình.

[**Bibliography**](#_page_311_0)  
Thư mục

**[Martin]:** *Agile Software Development: Principles, Patterns, and Practices,* Robert C. Martin, Prentice Hall, 2002.

<https://www.facebook.com/groups/phattrienlaptrinh>



[**8**](#_page_311_0)

[**Boundaries**](#_page_311_0)  
ranh giới

by James Grenning  
bởi James Grenning

We seldom control all the software in our systems. Sometimes we buy third-party pack-ages or use open source. Other times we depend on teams in our own company to produce components or subsystems for us. Somehow we must cleanly integrate this foreign code  
Chúng tôi hiếm khi kiểm soát tất cả phần mềm trong hệ thống của mình. Đôi khi chúng tôi mua gói của bên thứ ba hoặc sử dụng mã nguồn mở. Những lần khác, chúng tôi phụ thuộc vào các nhóm trong công ty của mình để sản xuất các thành phần hoặc hệ thống con cho chúng tôi. Bằng cách nào đó chúng ta phải tích hợp rõ ràng mã nước ngoài này

113

<https://www.facebook.com/groups/phattrienlaptrinh>

114 **Chapter 8: Boundaries**  
114 Chương 8: Ranh giới

with our own. In this chapter we look at practices and techniques to keep the boundaries of our software clean.  
với chính chúng ta. Trong chương này, chúng ta xem xét các phương pháp và kỹ thuật để giữ cho ranh giới của phần mềm luôn trong sạch.

[**Using Third-Party Code**](#_page_311_0)  
Sử dụng mã của bên thứ ba

There is a natural tension between the provider of an interface and the user of an interface. Providers of third-party packages and frameworks strive for broad applicability so they can work in many environments and appeal to a wide audience. Users, on the other hand, want an interface that is focused on their particular needs. This tension can cause problems at the boundaries of our systems.  
Có một sự căng thẳng tự nhiên giữa nhà cung cấp giao diện và người dùng giao diện. Các nhà cung cấp gói và khung của bên thứ ba cố gắng đạt được khả năng ứng dụng rộng rãi để chúng có thể hoạt động trong nhiều môi trường và thu hút nhiều đối tượng. Mặt khác, người dùng muốn có một giao diện tập trung vào các nhu cầu cụ thể của họ. Sự căng thẳng này có thể gây ra các vấn đề ở ranh giới hệ thống của chúng ta.

Let’s look at java.util.Mapas an example. As you can see by examining Figure 8-1, Maps have a very broad interface with plenty of capabilities. Certainly this power and ﬂexi-bility is useful, but it can also be a liability. For instance, our application might build up a Mapand pass it around. Our intention might be that none of the recipients of our Mapdelete anything in the map. But right there at the top of the list is the clear()method. Any user of the Maphas the power to clear it. Or maybe our design convention is that only particular types of objects can be stored in the Map, but Maps do not reliably constrain the types of objects placed within them. Any determined user can add items of any type to any Map.  
Hãy xem java.util.Mapas là một ví dụ. Như bạn có thể thấy bằng cách kiểm tra Hình 8-1, Bản đồ có giao diện rất rộng với nhiều khả năng. Chắc chắn sức mạnh và khả năng linh hoạt này là hữu ích, nhưng nó cũng có thể là một trách nhiệm pháp lý. Chẳng hạn, ứng dụng của chúng ta có thể xây dựng một Bản đồ và chuyển nó đi khắp nơi. Ý định của chúng tôi có thể là không ai trong số những người nhận Mapdelete của chúng tôi xóa bất kỳ thứ gì trên bản đồ. Nhưng ngay ở đầu danh sách là phương thức clear(). Bất kỳ người dùng nào của Maphas đều có quyền xóa nó. Hoặc có thể quy ước thiết kế của chúng tôi là chỉ các loại đối tượng cụ thể mới có thể được lưu trữ trong Bản đồ, nhưng Bản đồ không hạn chế một cách đáng tin cậy các loại đối tượng được đặt trong chúng. Bất kỳ người dùng xác định nào cũng có thể thêm các mục thuộc bất kỳ loại nào vào bất kỳ Bản đồ nào.

 clear() void – Map

 containsKey(Object key) boolean – Map

 containsValue(Object value) boolean – Map  entrySet() Set – Map

 equals(Object o) boolean – Map  get(Object key) Object – Map

 getClass() Class<? extends Object> – Object  hashCode() int – Map

 isEmpty() boolean – Map  keySet() Set – Map

 notify() void – Object

 notifyAll() void – Object

 put(Object key, Object value) Object – Map  putAll(Map t) void – Map

 remove(Object key) Object – Map  size() int – Map

 toString() String – Object  values() Collection – Map  wait() void – Object

 wait(long timeout) void – Object

 wait(long timeout, int nanos) void – Object

**Figure 8-1** The methods of Map  
Hình 8-1 Các phương pháp của Bản đồ

If our application needs a Map of Sensors, you might ﬁnd the sensors set up like this:  
Nếu ứng dụng của chúng tôi cần Bản đồ cảm biến, bạn có thể tìm thấy các cảm biến được thiết lập như sau:

Map sensors = new HashMap();

<https://www.facebook.com/groups/phattrienlaptrinh>

**Using Third-Party Code** 115  
Sử dụng mã của bên thứ ba 115

Then, when some other part of the code needs to access the sensor, you see this code:  
Sau đó, khi một số phần khác của mã cần truy cập vào cảm biến, bạn sẽ thấy mã này:

Sensor s = (Sensor)sensors.get(sensorId );

We don’t just see it once, but over and over again throughout the code. The client of this code carries the responsibility of getting an Objectfrom the Mapand casting it to the right type. This works, but it’s not clean code. Also, this code does not tell its story as well as it could. The readability of this code can be greatly improved by using generics, as shown below:  
Chúng tôi không chỉ nhìn thấy nó một lần mà còn lặp đi lặp lại trong toàn bộ mã. Máy khách của mã này chịu trách nhiệm lấy một Đối tượng từ Bản đồ và chuyển nó sang đúng loại. Điều này hoạt động, nhưng nó không phải là mã sạch. Ngoài ra, mã này không kể câu chuyện của nó tốt nhất có thể. Khả năng đọc của mã này có thể được cải thiện rất nhiều bằng cách sử dụng thuốc generic, như được hiển thị bên dưới:

Map<Sensor> sensors = new HashMap<Sensor>(); ...

Sensor s = sensors.get(sensorId );

However, this doesn’t solve the problem that Map<Sensor>provides more capability than we need or want.  
Tuy nhiên, điều này không giải quyết được vấn đề Map<Sensor>cung cấp nhiều khả năng hơn chúng ta cần hoặc muốn.

Passing an instance of Map<Sensor>liberally around the system means that there will be a lot of places to ﬁx if the interface to Mapever changes. You might think such a change to be unlikely, but remember that it changed when generics support was added in Java 5. Indeed, we’ve seen systems that are inhibited from using generics because of the sheer magnitude of changes needed to make up for the liberal use of Maps.  
Tự do truyền một phiên bản Map<Sensor> quanh hệ thống có nghĩa là sẽ có nhiều chỗ cần sửa nếu giao diện của Mapever thay đổi. Bạn có thể nghĩ rằng một sự thay đổi như vậy là khó xảy ra, nhưng hãy nhớ rằng nó đã thay đổi khi hỗ trợ thuốc generic được thêm vào trong Java 5. Thật vậy, chúng tôi đã thấy các hệ thống bị hạn chế sử dụng thuốc generic vì mức độ thay đổi lớn cần thiết để bù đắp cho sự thay đổi đó. sử dụng tự do của Bản đồ.

A cleaner way to use Mapmight look like the following. No user of Sensorswould care one bit if generics were used or not. That choice has become (and always should be) an implementation detail.  
Một cách sạch hơn để sử dụng Mapcó thể giống như sau. Không người dùng Cảm biến nào quan tâm dù chỉ một chút thuốc generic có được sử dụng hay không. Lựa chọn đó đã trở thành (và luôn phải là) một chi tiết triển khai.

public class Sensors {

private Map sensors = new HashMap();

public Sensor getById(String id) { return (Sensor) sensors.get(id);

}

//snip }

The interface at the boundary (Map) is hidden. It is able to evolve with very little impact on the rest of the application. The use of generics is no longer a big issue because the casting and type management is handled inside the Sensors class.  
Giao diện tại ranh giới (Bản đồ) bị ẩn. Nó có thể phát triển với rất ít tác động đến phần còn lại của ứng dụng. Việc sử dụng thuốc generic không còn là vấn đề lớn nữa vì việc truyền và quản lý kiểu được xử lý bên trong lớp Cảm biến.

This interface is also tailored and constrained to meet the needs of the application. It results in code that is easier to understand and harder to misuse. The Sensorsclass can enforce design and business rules.  
Giao diện này cũng được điều chỉnh và hạn chế để đáp ứng nhu cầu của ứng dụng. Nó dẫn đến mã dễ hiểu hơn và khó sử dụng sai hơn. Sensorsclass có thể thực thi các quy tắc thiết kế và kinh doanh.

We are not suggesting that every use of Mapbe encapsulated in this form. Rather, we are advising you not to pass Maps (or any other interface at a boundary) around your system. If you use a boundary interface like Map, keep it inside the class, or close family of classes, where it is used. Avoid returning it from, or accepting it as an argument to, public APIs.  
Chúng tôi không gợi ý rằng mọi hoạt động sử dụng Mapbe đều được gói gọn trong biểu mẫu này. Thay vào đó, chúng tôi khuyên bạn không nên đi qua Bản đồ (hoặc bất kỳ giao diện nào khác ở ranh giới) xung quanh hệ thống của mình. Nếu bạn sử dụng một giao diện ranh giới như Bản đồ, hãy giữ nó bên trong lớp hoặc họ gần các lớp, nơi nó được sử dụng. Tránh trả lại hoặc chấp nhận nó làm đối số cho các API công khai.

<https://www.facebook.com/groups/phattrienlaptrinh>

116 **Chapter 8: Boundaries**  
116 Chương 8: Ranh giới

[**Exploring and Learning Boundaries**](#_page_311_0)  
Khám phá và học hỏi ranh giới

Third-party code helps us get more functionality delivered in less time. Where do we start when we want to utilize some third-party package? It’s not our job to test the third-party code, but it may be in our best interest to write tests for the third-party code we use.  
Mã của bên thứ ba giúp chúng tôi nhận được nhiều chức năng hơn trong thời gian ngắn hơn. Chúng ta bắt đầu từ đâu khi muốn sử dụng một số gói của bên thứ ba? Công việc của chúng tôi không phải là kiểm tra mã của bên thứ ba, nhưng chúng tôi có thể có lợi nhất khi viết các bài kiểm tra cho mã của bên thứ ba mà chúng tôi sử dụng.

Suppose it is not clear how to use our third-party library. We might spend a day or two (or more) reading the documentation and deciding how we are going to use it. Then we might write our code to use the third-party code and see whether it does what we think. We would not be surprised to ﬁnd ourselves bogged down in long debugging sessions trying to ﬁgure out whether the bugs we are experiencing are in our code or theirs.  
Giả sử không rõ cách sử dụng thư viện bên thứ ba của chúng tôi. Chúng tôi có thể dành một hoặc hai ngày (hoặc hơn) để đọc tài liệu và quyết định cách chúng tôi sẽ sử dụng nó. Sau đó, chúng tôi có thể viết mã của mình để sử dụng mã của bên thứ ba và xem liệu mã đó có hoạt động như những gì chúng tôi nghĩ hay không. Chúng ta sẽ không ngạc nhiên khi thấy mình bị sa lầy trong các phiên sửa lỗi dài để cố gắng tìm hiểu xem lỗi mà chúng ta đang gặp phải nằm trong mã của chúng ta hay của họ.

Learning the third-party code is hard. Integrating the third-party code is hard too. Doing both at the same time is doubly hard. What if we took a different approach? Instead of experimenting and trying out the new stuff in our production code, we could write some tests to explore our understanding of the third-party code. Jim Newkirk calls such tests *learning tests.*1  
Học mã của bên thứ ba rất khó. Tích hợp mã của bên thứ ba cũng khó. Làm cả hai cùng một lúc khó gấp đôi. Điều gì sẽ xảy ra nếu chúng ta thực hiện một cách tiếp cận khác? Thay vì thử nghiệm và dùng thử nội dung mới trong mã sản xuất của mình, chúng tôi có thể viết một số bài kiểm tra để khám phá hiểu biết của mình về mã của bên thứ ba. Jim Newkirk gọi những bài kiểm tra như vậy là bài kiểm tra học tập.1

In learning tests we call the third-party API, as we expect to use it in our application. We’re essentially doing controlled experiments that check our understanding of that API. The tests focus on what we want out of the API.  
Trong các bài kiểm tra học tập, chúng tôi gọi API của bên thứ ba, vì chúng tôi muốn sử dụng nó trong ứng dụng của mình. Về cơ bản, chúng tôi đang thực hiện các thử nghiệm có kiểm soát để kiểm tra hiểu biết của chúng tôi về API đó. Các bài kiểm tra tập trung vào những gì chúng tôi muốn từ API.

[**Learning log4j**](#_page_311_0)  
Học log4j

Let’s say we want to use the apache log4jpackage rather than our own custom-built log-ger. We download it and open the introductory documentation page. Without too much reading we write our ﬁrst test case, expecting it to write “hello” to the console.  
Giả sử chúng ta muốn sử dụng gói log4j của apache thay vì trình ghi nhật ký được tạo tùy chỉnh của riêng chúng ta. Chúng tôi tải xuống và mở trang tài liệu giới thiệu. Không cần đọc quá nhiều, chúng tôi viết trường hợp thử nghiệm đầu tiên của mình, hy vọng nó sẽ ghi “xin chào” vào bảng điều khiển.

@Test

public void testLogCreate() {

Logger logger = Logger.getLogger("MyLogger"); logger.info("hello");

}

When we run it, the logger produces an error that tells us we need something called an Appender. After a little more reading we ﬁnd that there is a ConsoleAppender. So we create a ConsoleAppender and see whether we have unlocked the secrets of logging to the console.  
Khi chúng tôi chạy nó, bộ ghi sẽ tạo ra một lỗi cho chúng tôi biết rằng chúng tôi cần một thứ gọi là Appender. Sau khi đọc thêm một chút, chúng tôi thấy rằng có một ConsoleAppender. Vì vậy, chúng tôi tạo một ConsoleAppender và xem liệu chúng tôi đã mở khóa các bí mật đăng nhập vào bảng điều khiển hay chưa.

@Test

public void testLogAddAppender() {

Logger logger = Logger.getLogger("MyLogger"); ConsoleAppender appender = new ConsoleAppender(); logger.addAppender(appender); logger.info("hello");

}

1. [BeckTDD], pp. 136–137.

<https://www.facebook.com/groups/phattrienlaptrinh>

**Learning log4j** 117  
Học log4j 117

This time we ﬁnd that the Appenderhas no output stream. Odd—it seems logical that it’d have one. After a little help from Google, we try the following:  
Lần này, chúng tôi thấy rằng Appender không có luồng đầu ra. Kỳ lạ - có vẻ hợp lý khi nó có một cái. Sau một chút trợ giúp từ Google, chúng tôi thử các cách sau:

@Test

public void testLogAddAppender() {

Logger logger = Logger.getLogger("MyLogger"); logger.removeAllAppenders(); logger.addAppender(new ConsoleAppender(

new PatternLayout("%p %t %m%n"), ConsoleAppender.SYSTEM\_OUT));

logger.info("hello"); }

That worked; a log message that includes “hello” came out on the console! It seems odd that we have to tell the ConsoleAppender that it writes to the console.  
Điều đó đã hiệu quả; một thông báo nhật ký bao gồm “xin chào” xuất hiện trên bảng điều khiển! Có vẻ kỳ lạ là chúng ta phải nói với ConsoleAppender rằng nó ghi vào bảng điều khiển.

Interestingly enough, when we remove the ConsoleAppender.SystemOutargument, we see that “hello” is still printed. But when we take out the PatternLayout, it once again com-plains about the lack of an output stream. This is very strange behavior.  
Thật thú vị, khi chúng tôi xóa ConsoleAppender.SystemOutargument, chúng tôi thấy rằng “xin chào” vẫn được in. Nhưng khi chúng tôi gỡ bỏ PatternLayout, nó lại phàn nàn về việc thiếu luồng đầu ra. Đây là hành vi rất kỳ lạ.

Looking a little more carefully at the documentation, we see that the default ConsoleAppenderconstructor is “unconﬁgured,” which does not seem too obvious or useful. This feels like a bug, or at least an inconsistency, in log4j.  
Xem kỹ tài liệu hơn một chút, chúng ta thấy rằng ConsoleAppenderconstructor mặc định là “chưa được định cấu hình”, điều này có vẻ không quá rõ ràng hoặc hữu ích. Điều này giống như một lỗi hoặc ít nhất là sự không nhất quán trong log4j.

A bit more googling, reading, and testing, and we eventually wind up with Listing 8-1. We’ve discovered a great deal about the way that log4jworks, and we’ve encoded that knowledge into a set of simple unit tests.  
Google, đọc và kiểm tra thêm một chút nữa, và cuối cùng chúng ta có được Liệt kê 8-1. Chúng tôi đã khám phá ra rất nhiều điều về cách log4j hoạt động và chúng tôi đã mã hóa kiến thức đó thành một tập hợp các bài kiểm tra đơn vị đơn giản.

**Listing 8-1 LogTest.java  
Liệt kê 8-1 LogTest.java**

public class LogTest { private Logger logger;

@Before

public void initialize() {

logger = Logger.getLogger("logger"); logger.removeAllAppenders(); Logger.getRootLogger().removeAllAppenders();

} @Test

public void basicLogger() { BasicConfigurator.configure(); logger.info("basicLogger");

}

@Test

public void addAppenderWithStream() { logger.addAppender(new ConsoleAppender(

new PatternLayout("%p %t %m%n"), ConsoleAppender.SYSTEM\_OUT));

logger.info("addAppenderWithStream"); }

<https://www.facebook.com/groups/phattrienlaptrinh>

118 **Chapter 8: Boundaries**  
118 Chương 8: Ranh Giới

**Listing 8-1 (continued) LogTest.java**  
Liệt kê 8-1 (tiếp theo) LogTest.java

@Test

public void addAppenderWithoutStream() { logger.addAppender(new ConsoleAppender(

new PatternLayout("%p %t %m%n"))); logger.info("addAppenderWithoutStream");

} }

Now we know how to get a simple console logger initialized, and we can encapsulate that knowledge into our own logger class so that the rest of our application is isolated from the log4j boundary interface.  
Bây giờ chúng ta đã biết cách khởi tạo một trình ghi nhật ký giao diện điều khiển đơn giản và chúng ta có thể gói gọn kiến thức đó vào lớp trình ghi nhật ký của riêng mình để phần còn lại của ứng dụng được tách biệt khỏi giao diện ranh giới log4j.

[**Learning Tests Are Better Than Free**](#_page_311_0)  
Bài kiểm tra học tập tốt hơn miễn phí

The learning tests end up costing nothing. We had to learn the API anyway, and writing those tests was an easy and isolated way to get that knowledge. The learning tests were precise experiments that helped increase our understanding.  
Các bài kiểm tra học tập cuối cùng không tốn kém gì. Dù sao thì chúng tôi cũng phải học API và viết những bài kiểm tra đó là một cách dễ dàng và riêng biệt để có được kiến ​​thức đó. Các bài kiểm tra học tập là những thí nghiệm chính xác giúp nâng cao hiểu biết của chúng tôi.

Not only are learning tests free, they have a positive return on investment. When there are new releases of the third-party package, we run the learning tests to see whether there are behavioral differences.  
Các bài kiểm tra học tập không chỉ miễn phí mà còn có lợi tức đầu tư tích cực. Khi có các bản phát hành mới của gói bên thứ ba, chúng tôi sẽ chạy thử nghiệm học tập để xem liệu có sự khác biệt về hành vi hay không.

Learning tests verify that the third-party packages we are using work the way we expect them to. Once integrated, there are no guarantees that the third-party code will stay compatible with our needs. The original authors will have pressures to change their code to meet new needs of their own. They will ﬁx bugs and add new capabilities. With each release comes new risk. If the third-party package changes in some way incompatible with our tests, we will ﬁnd out right away.  
Các bài kiểm tra học tập xác minh rằng các gói của bên thứ ba mà chúng tôi đang sử dụng hoạt động theo cách chúng tôi mong đợi. Sau khi được tích hợp, không có gì đảm bảo rằng mã của bên thứ ba sẽ vẫn tương thích với nhu cầu của chúng tôi. Các tác giả ban đầu sẽ có áp lực phải thay đổi mã của họ để đáp ứng nhu cầu mới của riêng họ. Họ sẽ sửa lỗi và thêm các khả năng mới. Với mỗi lần phát hành đều có rủi ro mới. Nếu gói của bên thứ ba thay đổi theo một cách nào đó không tương thích với các thử nghiệm của chúng tôi, chúng tôi sẽ phát hiện ra ngay.

Whether you need the learning provided by the learning tests or not, a clean boundary should be supported by a set of outbound tests that exercise the interface the same way the production code does. Without these *boundary tests* to ease the migration, we might be tempted to stay with the old version longer than we should.  
Cho dù bạn có cần kiến thức được cung cấp bởi các bài kiểm tra học tập hay không, thì một ranh giới rõ ràng phải được hỗ trợ bởi một tập hợp các bài kiểm tra bên ngoài thực hiện giao diện giống như cách mà mã sản xuất thực hiện. Nếu không có các thử nghiệm ranh giới này để giảm bớt sự di chuyển, chúng ta có thể muốn ở lại với phiên bản cũ lâu hơn mức cần thiết.

[**Using Code That Does Not Yet Exist**](#_page_311_0)  
Sử dụng mã chưa tồn tại

There is another kind of boundary, one that separates the known from the unknown. There are often places in the code where our knowledge seems to drop off the edge. Sometimes what is on the other side of the boundary is unknowable (at least right now). Sometimes we choose to look no farther than the boundary.  
Có một loại ranh giới khác, ranh giới ngăn cách cái đã biết với cái chưa biết. Thường có những chỗ trong đoạn mã mà kiến thức của chúng ta dường như bị bỏ quên. Đôi khi những gì ở phía bên kia của ranh giới là không thể biết được (ít nhất là ngay bây giờ). Đôi khi chúng ta chọn không nhìn xa hơn ranh giới.

A number of years back I was part of a team developing software for a radio com-munications system. There was a subsystem, the “Transmitter,” that we knew little about, and the people responsible for the subsystem had not gotten to the point of deﬁning their interface. We did not want to be blocked, so we started our work far away from the unknown part of the code.  
Cách đây vài năm, tôi là thành viên của nhóm phát triển phần mềm cho hệ thống liên lạc vô tuyến. Có một hệ thống con, “Máy phát,” mà chúng tôi biết rất ít, và những người chịu trách nhiệm về hệ thống con đó đã không đi đến chỗ xác định giao diện của chúng. Chúng tôi không muốn bị chặn, vì vậy chúng tôi bắt đầu công việc của mình từ xa phần mã chưa biết.

<https://www.facebook.com/groups/phattrienlaptrinh>

**Using Code That Does Not Yet Exist** 119  
Sử dụng mã chưa tồn tại 119

We had a pretty good idea of where our world ended and the new world began. As we worked, we sometimes bumped up against this boundary. Though mists and clouds of ignorance obscured our view beyond the boundary, our work made us aware of what we *wanted* the boundary interface to be. We wanted to tell the transmitter something like this:  
Chúng tôi đã có một ý tưởng khá hay về nơi thế giới của chúng ta kết thúc và thế giới mới bắt đầu. Khi chúng tôi làm việc, đôi khi chúng tôi đụng phải ranh giới này. Mặc dù sương mù và những đám mây vô minh che khuất tầm nhìn của chúng tôi bên ngoài ranh giới, nhưng công việc của chúng tôi giúp chúng tôi nhận thức được những gì chúng tôi muốn giao diện ranh giới trở thành. Chúng tôi muốn nói với người phát điều gì đó như thế này:

*Key the transmitter on the provided frequency and emit an analog representation of the data coming from this stream.  
Nhập bộ phát trên tần số được cung cấp và phát ra biểu diễn tương tự của dữ liệu đến từ luồng này.*

We had no idea how that would be done because the API had not been designed yet. So we decided to work out the details later.  
Chúng tôi không biết điều đó sẽ được thực hiện như thế nào vì API vẫn chưa được thiết kế. Vì vậy, chúng tôi quyết định tìm hiểu chi tiết sau.

To keep from being blocked, we deﬁned our own interface. We called it something catchy, like Transmitter. We gave it a method called transmitthat took a frequency and a data stream. This was the interface we *wished* we had.  
Để không bị chặn, chúng tôi đã xác định giao diện của riêng mình. Chúng tôi gọi nó là một cái gì đó hấp dẫn, như Máy phát. Chúng tôi cung cấp cho nó một phương thức gọi là truyền nhận một tần số và một luồng dữ liệu. Đây là giao diện mà chúng tôi ước mình có.

One good thing about writing the interface we wish we had is that it’s under our control. This helps keep client code more readable and focused on what it is trying to accomplish.  
Một điều tốt khi viết giao diện mà chúng tôi mong muốn là nó nằm trong tầm kiểm soát của chúng tôi. Điều này giúp giữ cho mã máy khách dễ đọc hơn và tập trung vào những gì nó đang cố gắng hoàn thành.

In Figure 8-2, you can see that we insulated the CommunicationsControllerclasses from the transmitter API (which was out of our control and undeﬁned). By using our own application speciﬁc interface, we kept our CommunicationsController code clean and expressive. Once the transmitter API was deﬁned, we wrote the TransmitterAdapterto bridge the gap. The ADAPTER2 encapsulated the interaction with the API and provides a single place to change when the API evolves.  
Trong Hình 8-2, bạn có thể thấy rằng chúng tôi đã cách ly các lớp CommunicationsController khỏi API máy phát (ngoài tầm kiểm soát của chúng tôi và chưa được xác định). Bằng cách sử dụng giao diện dành riêng cho ứng dụng của chúng tôi, chúng tôi đã giữ cho mã CommunicationsController của mình rõ ràng và rõ ràng. Khi API máy phát đã được xác định, chúng tôi đã viết TransmitterAdapter để thu hẹp khoảng cách. ADAPTER2 gói gọn tương tác với API và cung cấp một nơi duy nhất để thay đổi khi API phát triển.

**Figure 8-2** Predicting the transmitter  
Hình 8-2 Dự đoán máy phát

This design also gives us a very convenient seam3 in the code for testing. Using a suitable FakeTransmitter, we can test the CommunicationsControllerclasses. We can also create boundary tests once we have the TransmitterAPIthat make sure we are using the API correctly.  
Thiết kế này cũng cung cấp cho chúng ta một seam3 trong mã rất tiện lợi để thử nghiệm. Sử dụng FakeTransmitter phù hợp, chúng ta có thể kiểm tra các lớp CommunicationsController. Chúng tôi cũng có thể tạo các thử nghiệm ranh giới sau khi có API Bộ phát để đảm bảo rằng chúng tôi đang sử dụng API đúng cách.

2. See the Adapter pattern in [GOF]. 3. See more about seams in [WELC].  
2. Xem mẫu Bộ điều hợp trong [GOF]. 3. Xem thêm về đường may trong [WELC].

<https://www.facebook.com/groups/phattrienlaptrinh>

120 **Chapter 8: Boundaries**  
120 Chương 8: Ranh giới

[**Clean Boundaries**](#_page_311_0)  
ranh giới sạch

Interesting things happen at boundaries. Change is one of those things. Good software designs accommodate change without huge investments and rework. When we use code that is out of our control, special care must be taken to protect our investment and make sure future change is not too costly.  
Những điều thú vị xảy ra ở ranh giới. Thay đổi là một trong những điều đó. Các thiết kế phần mềm tốt phù hợp với sự thay đổi mà không cần đầu tư lớn và làm lại. Khi chúng tôi sử dụng mã nằm ngoài tầm kiểm soát của mình, chúng tôi phải đặc biệt lưu ý để bảo vệ khoản đầu tư của mình và đảm bảo thay đổi trong tương lai không quá tốn kém.

Code at the boundaries needs clear separation and tests that deﬁne expectations. We should avoid letting too much of our code know about the third-party particulars. It’s better to depend on something *you* control than on something you don’t control, lest it end up controlling you.  
Mã tại các ranh giới cần sự tách biệt rõ ràng và các bài kiểm tra xác định các kỳ vọng. Chúng ta nên tránh để quá nhiều mã của mình biết về thông tin chi tiết của bên thứ ba. Thà phụ thuộc vào thứ bạn kiểm soát còn hơn phụ thuộc vào thứ bạn không kiểm soát, kẻo cuối cùng nó sẽ kiểm soát bạn.

We manage third-party boundaries by having very few places in the code that refer to them. We may wrap them as we did with Map, or we may use an ADAPTER to convert from our perfect interface to the provided interface. Either way our code speaks to us better, promotes internally consistent usage across the boundary, and has fewer maintenance points when the third-party code changes.  
Chúng tôi quản lý ranh giới của bên thứ ba bằng cách có rất ít vị trí trong mã đề cập đến họ. Chúng tôi có thể gói chúng như chúng tôi đã làm với Bản đồ hoặc chúng tôi có thể sử dụng ADAPTER để chuyển đổi từ giao diện hoàn hảo của chúng tôi sang giao diện được cung cấp. Dù bằng cách nào thì mã của chúng tôi nói với chúng tôi tốt hơn, thúc đẩy việc sử dụng nhất quán nội bộ trên toàn ranh giới và có ít điểm bảo trì hơn khi mã của bên thứ ba thay đổi.

[**Bibliography**](#_page_311_0)  
Thư mục

**[BeckTDD]:** *Test Driven Development,* Kent Beck, Addison-Wesley, 2003.

**[GOF]:** *Design Patterns: Elements of Reusable Object Oriented Software,* Gamma et al., Addison-Wesley, 1996.

**[WELC]:** *Working Effectively with Legacy Code,*Addison-Wesley, 2004.

<https://www.facebook.com/groups/phattrienlaptrinh>



[**9**](#_page_311_0)

[**Unit Tests**](#_page_311_0)  
bài kiểm tra đơn vị

Our profession has come a long way in the last ten years. In 1997 no one had heard of Test Driven Development. For the vast majority of us, unit tests were short bits of throw-away code that we wrote to make sure our programs “worked.” We would painstakingly write our classes and methods, and then we would concoct some ad hoc code to test them. Typically this would involve some kind of simple driver program that would allow us to manually interact with the program we had written.  
Nghề nghiệp của chúng tôi đã đi một chặng đường dài trong mười năm qua. Vào năm 1997, không ai nghe nói về Test Driven Development. Đối với đại đa số chúng ta, các bài kiểm tra đơn vị là những đoạn mã ngắn dùng một lần mà chúng ta đã viết để đảm bảo chương trình của mình “hoạt động”. Chúng tôi sẽ cẩn thận viết các lớp và phương thức của mình, sau đó chúng tôi sẽ tạo ra một số mã đặc biệt để kiểm tra chúng. Thông thường, điều này sẽ liên quan đến một số loại chương trình trình điều khiển đơn giản cho phép chúng tôi tương tác thủ công với chương trình mà chúng tôi đã viết.

I remember writing a C++ program for an embedded real-time system back in the mid-90s. The program was a simple timer with the following signature:  
Tôi nhớ đã viết một chương trình C++ cho một hệ thống nhúng thời gian thực vào giữa những năm 90. Chương trình là một bộ đếm thời gian đơn giản với chữ ký sau:

void Timer::ScheduleCommand(Command\* theCommand, int milliseconds)

The idea was simple; the executemethod of the Commandwould be executed in a new thread after the speciﬁed number of milliseconds. The problem was, how to test it.  
Ý tưởng rất đơn giản; phương thức thực thi của Lệnh sẽ được thực thi trong một luồng mới sau số mili giây đã chỉ định. Vấn đề là, làm thế nào để kiểm tra nó.

121

<https://www.facebook.com/groups/phattrienlaptrinh>

122 **Chapter 9: Unit Tests**  
122 Chương 9: Bài kiểm tra đơn vị

I cobbled together a simple driver program that listened to the keyboard. Every time a character was typed, it would schedule a command that would type the same character ﬁve seconds later. Then I tapped out a rhythmic melody on the keyboard and waited for that melody to replay on the screen ﬁve seconds later.  
Tôi đã tập hợp một chương trình trình điều khiển đơn giản để nghe bàn phím. Mỗi khi một ký tự được gõ, nó sẽ lên lịch cho một lệnh sẽ gõ cùng một ký tự trong 5 giây sau đó. Sau đó, tôi gõ một giai điệu nhịp nhàng trên bàn phím và đợi giai điệu đó phát lại trên màn hình năm giây sau đó.

“I . . . want-a-girl . . . just . . . like-the-girl-who-marr . . . ied . . . dear . . . old . . . dad.”  
"TÔI . . . muốn có một cô gái. . . chỉ . . . như-cô-gái-đã-kết-hôn . . . Thân mến . . . cũ . . . bố."

I actually sang that melody while typing the “.” key, and then I sang it again as the dots appeared on the screen.  
Tôi thực sự đã hát giai điệu đó trong khi gõ “.” và sau đó tôi hát lại khi các dấu chấm xuất hiện trên màn hình.

That was my test! Once I saw it work and demonstrated it to my colleagues, I threw the test code away.  
Đó là bài kiểm tra của tôi! Khi tôi thấy nó hoạt động và trình diễn nó cho các đồng nghiệp của mình, tôi đã ném mã thử nghiệm đi.

As I said, our profession has come a long way. Nowadays I would write a test that made sure that every nook and cranny of that code worked as I expected it to. I would isolate my code from the operating system rather than just calling the standard timing functions. I would mock out those timing functions so that I had absolute control over the time. I would schedule commands that set boolean ﬂags, and then I would step the time forward, watching those ﬂags and ensuring that they went from false to true just as I changed the time to the right value.  
Như tôi đã nói, nghề nghiệp của chúng tôi đã đi một chặng đường dài. Ngày nay, tôi sẽ viết một bài kiểm tra để đảm bảo rằng mọi ngóc ngách của mã đó đều hoạt động như tôi mong đợi. Tôi sẽ tách mã của mình khỏi hệ điều hành thay vì chỉ gọi các hàm định thời gian tiêu chuẩn. Tôi sẽ giả lập các chức năng thời gian đó để tôi có quyền kiểm soát tuyệt đối về thời gian. Tôi sẽ lên lịch cho các lệnh thiết lập các cờ boolean, sau đó tôi sẽ tăng dần thời gian, quan sát các cờ đó và đảm bảo rằng chúng chuyển từ sai thành đúng ngay khi tôi thay đổi thời gian thành giá trị đúng.

Once I got a suite of tests to pass, I would make sure that those tests were convenient to run for anyone else who needed to work with the code. I would ensure that the tests and the code were checked in together into the same source package.  
Sau khi vượt qua một loạt bài kiểm tra, tôi sẽ đảm bảo rằng những bài kiểm tra đó thuận tiện để chạy cho bất kỳ ai khác cần làm việc với mã. Tôi sẽ đảm bảo rằng các bài kiểm tra và mã được kiểm tra cùng nhau trong cùng một gói nguồn.

Yes, we’ve come a long way; but we have farther to go. The Agile and TDD move-ments have encouraged many programmers to write automated unit tests, and more are joining their ranks every day. But in the mad rush to add testing to our discipline, many programmers have missed some of the more subtle, and important, points of writing good tests.  
Vâng, chúng tôi đã đi một chặng đường dài; nhưng chúng ta còn phải đi xa hơn nữa. Các phong trào Agile và TDD đã khuyến khích nhiều lập trình viên viết các bài kiểm tra đơn vị tự động và nhiều người khác đang gia nhập hàng ngũ của họ mỗi ngày. Nhưng trong cơn vội vã điên cuồng để thêm kiểm tra vào nguyên tắc của chúng tôi, nhiều lập trình viên đã bỏ lỡ một số điểm tinh tế và quan trọng hơn để viết các bài kiểm tra tốt.

[**The Three Laws of TDD**](#_page_311_0)  
Ba định luật của TDD

By now everyone knows that TDD asks us to write unit tests ﬁrst, before we write produc-tion code. But that rule is just the tip of the iceberg. Consider the following three laws:1  
Bây giờ mọi người đều biết rằng TDD yêu cầu chúng ta viết các bài kiểm tra đơn vị trước, trước khi chúng ta viết mã sản xuất. Nhưng quy định đó chỉ là phần nổi của tảng băng chìm. Xét ba định luật sau:1

**First Law** You may not write production code until you have written a failing unit test.  
Luật đầu tiên Bạn không được viết mã sản xuất cho đến khi bạn viết một bài kiểm tra đơn vị không thành công.

**Second Law** You may not write more of a unit test than is sufﬁcient to fail, and not com-piling is failing.  
Luật thứ hai Bạn không thể viết nhiều bài kiểm tra đơn vị hơn mức đủ để thất bại và không biên dịch là thất bại.

**Third Law** You may not write more production code than is sufﬁcient to pass the cur-rently failing test.  
Định luật thứ ba Bạn không thể viết nhiều mã sản xuất hơn mức đủ để vượt qua bài kiểm tra hiện đang thất bại.

1. *Professionalism and Test-Driven Development*, Robert C. Martin, Object Mentor, IEEE Software, May/June 2007 (Vol. 24, No. 3) pp. 32–36  
1. Professionalism and Test-Driven Development, Robert C. Martin, Object Mentor, IEEE Software, tháng 5/tháng 6 năm 2007 (Tập 24, Số 3) trang 32–36

<http://doi.ieeecomputersociety.org/10.1109/MS.2007.85>

<https://www.facebook.com/groups/phattrienlaptrinh>

**Keeping Tests Clean** 123  
Giữ Sạch Sẽ Xét Nghiệm 123

These three laws lock you into a cycle that is perhaps thirty seconds long. The tests and the production code are written *together,* with the tests just a few seconds ahead of the production code.  
Ba định luật này khóa bạn vào một chu kỳ có lẽ dài ba mươi giây. Các bài kiểm tra và mã sản xuất được viết cùng nhau, với các bài kiểm tra chỉ vài giây trước mã sản xuất.

If we work this way, we will write dozens of tests every day, hundreds of tests every month, and thousands of tests every year. If we work this way, those tests will cover virtu-ally all of our production code. The sheer bulk of those tests, which can rival the size of the production code itself, can present a daunting management problem.  
Nếu chúng ta làm việc theo cách này, chúng ta sẽ viết hàng chục bài kiểm tra mỗi ngày, hàng trăm bài kiểm tra mỗi tháng và hàng nghìn bài kiểm tra mỗi năm. Nếu chúng tôi làm việc theo cách này, những thử nghiệm đó sẽ gần như bao gồm tất cả mã sản xuất của chúng tôi. Phần lớn các thử nghiệm đó, có thể cạnh tranh với kích thước của chính mã sản xuất, có thể gây ra một vấn đề quản lý khó khăn.

[**Keeping Tests Clean**](#_page_311_0)  
Giữ các bài kiểm tra sạch sẽ

Some years back I was asked to coach a team who had explicitly decided that their test code *should not* be maintained to the same standards of quality as their production code. They gave each other license to break the rules in their unit tests. “Quick and dirty” was the watchword. Their variables did not have to be well named, their test functions did not need to be short and descriptive. Their test code did not need to be well designed and thoughtfully partitioned. So long as the test code worked, and so long as it covered the pro-duction code, it was good enough.  
Vài năm trước, tôi được yêu cầu huấn luyện một nhóm đã quyết định rõ ràng rằng mã thử nghiệm của họ không nên được duy trì theo cùng tiêu chuẩn chất lượng như mã sản xuất của họ. Họ đã trao cho nhau giấy phép để phá vỡ các quy tắc trong bài kiểm tra đơn vị của họ. “Nhanh và bẩn” là khẩu hiệu. Các biến của chúng không cần phải được đặt tên hay, các chức năng kiểm tra của chúng không cần phải ngắn gọn và mang tính mô tả. Mã thử nghiệm của họ không cần phải được thiết kế tốt và phân vùng cẩn thận. Chừng nào mã kiểm tra còn hoạt động, và chừng nào nó bao phủ mã sản xuất, chừng đó là đủ tốt.

Some of you reading this might sympathize with that decision. Perhaps, long in the past, you wrote tests of the kind that I wrote for that Timerclass. It’s a huge step from writing that kind of throw-away test, to writing a suite of automated unit tests. So, like the team I was coaching, you might decide that having dirty tests is better than having no tests.  
Một số bạn đọc điều này có thể thông cảm với quyết định đó. Có lẽ, từ rất lâu trước đây, bạn đã viết các bài kiểm tra thuộc loại mà tôi đã viết cho Timerclass đó. Đó là một bước tiến lớn từ việc viết loại bài kiểm tra bỏ đi đó sang việc viết một bộ bài kiểm tra đơn vị tự động. Vì vậy, giống như đội mà tôi đang huấn luyện, bạn có thể quyết định rằng có các bài kiểm tra bẩn vẫn tốt hơn là không có bài kiểm tra nào.

What this team did not realize was that having dirty tests is equivalent to, if not worse than, having no tests. The problem is that tests must change as the production code evolves. The dirtier the tests, the harder they are to change. The more tangled the test code, the more likely it is that you will spend more time cramming new tests into the suite than it takes to write the new production code. As you modify the production code, old tests start to fail, and the mess in the test code makes it hard to get those tests to pass again. So the tests become viewed as an ever-increasing liability.  
Điều mà nhóm này không nhận ra là việc có các bài kiểm tra bẩn tương đương với việc không có bài kiểm tra nào, nếu không muốn nói là tệ hơn. Vấn đề là các bài kiểm tra phải thay đổi khi mã sản xuất phát triển. Các bài kiểm tra càng bẩn, chúng càng khó thay đổi. Mã kiểm tra càng rối rắm thì càng có nhiều khả năng bạn sẽ dành nhiều thời gian hơn để nhồi nhét các kiểm tra mới vào bộ phần mềm hơn là viết mã sản xuất mới. Khi bạn sửa đổi mã sản xuất, các bài kiểm tra cũ bắt đầu thất bại và sự lộn xộn trong mã kiểm tra khiến cho các bài kiểm tra đó khó vượt qua lần nữa. Vì vậy, các bài kiểm tra được xem như một trách nhiệm pháp lý ngày càng tăng.

From release to release the cost of maintaining my team’s test suite rose. Eventually it became the single biggest complaint among the developers. When managers asked why their estimates were getting so large, the developers blamed the tests. In the end they were forced to discard the test suite entirely.  
Từ bản phát hành này đến bản phát hành khác, chi phí duy trì bộ thử nghiệm của nhóm tôi đều tăng. Cuối cùng, nó trở thành khiếu nại lớn nhất của các nhà phát triển. Khi các nhà quản lý hỏi tại sao ước tính của họ lại quá lớn, các nhà phát triển đã đổ lỗi cho các bài kiểm tra. Cuối cùng, họ buộc phải loại bỏ hoàn toàn bộ thử nghiệm.

But, without a test suite they lost the ability to make sure that changes to their code base worked as expected. Without a test suite they could not ensure that changes to one part of their system did not break other parts of their system. So their defect rate began to rise. As the number of unintended defects rose, they started to fear making changes. They stopped cleaning their production code because they feared the changes would do more harm than good. Their production code began to rot. In the end they were left with no tests, tangled and bug-riddled production code, frustrated customers, and the feeling that their testing effort had failed them.  
Tuy nhiên, nếu không có bộ kiểm tra, họ sẽ mất khả năng đảm bảo rằng những thay đổi đối với cơ sở mã của họ hoạt động như mong đợi. Nếu không có bộ kiểm thử, họ không thể đảm bảo rằng những thay đổi đối với một phần trong hệ thống của họ không làm hỏng các phần khác trong hệ thống của họ. Vì vậy, tỷ lệ lỗi của họ bắt đầu tăng lên. Khi số lượng lỗi ngoài ý muốn tăng lên, họ bắt đầu sợ thay đổi. Họ đã ngừng làm sạch mã sản xuất của mình vì họ sợ những thay đổi sẽ gây hại nhiều hơn là có lợi. Mã sản xuất của họ bắt đầu thối rữa. Cuối cùng, họ không có bài kiểm tra nào, mã sản xuất rối rắm và đầy lỗi, khách hàng thất vọng và cảm giác rằng nỗ lực thử nghiệm của họ đã thất bại.

<https://www.facebook.com/groups/phattrienlaptrinh>

124 **Chapter 9: Unit Tests**  
124 Chương 9: Bài kiểm tra đơn vị

In a way they were right. Their testing effort *had* failed them. But it was their decision to allow the tests to be messy that was the seed of that failure. Had they kept their tests clean, their testing effort would not have failed. I can say this with some certainty because I have participated in, and coached, many teams who have been successful with *clean* unit tests.  
Theo một cách nào đó họ đã đúng. Nỗ lực thử nghiệm của họ đã làm họ thất bại. Nhưng chính quyết định của họ để cho phép các bài kiểm tra trở nên lộn xộn là mầm mống của sự thất bại đó. Nếu họ giữ cho các bài kiểm tra của mình trong sạch, thì nỗ lực kiểm tra của họ đã không thất bại. Tôi có thể nói điều này một cách chắc chắn vì tôi đã tham gia và huấn luyện nhiều đội đã thành công với các bài kiểm tra đơn vị sạch.

The moral of the story is simple: *Test code is just as important as production code.* It is not a second-class citizen. It requires thought, design, and care. It must be kept as clean as production code.  
Đạo đức của câu chuyện rất đơn giản: Mã kiểm tra cũng quan trọng như mã sản xuất. Nó không phải là công dân hạng hai. Nó đòi hỏi sự suy nghĩ, thiết kế và chăm sóc. Nó phải được giữ sạch sẽ như mã sản xuất.

[**Tests Enable the -ilities**](#_page_311_0)  
Kiểm tra Kích hoạt -ilities

If you don’t keep your tests clean, you will lose them. And without them, you lose the very thing that keeps your production code ﬂexible. Yes, you read that correctly. It is *unit tests* that keep our code ﬂexible, maintainable, and reusable. The reason is simple. If you have tests, you do not fear making changes to the code! Without tests every change is a possible bug. No matter how ﬂexible your architecture is, no matter how nicely partitioned your design, without tests you will be reluctant to make changes because of the fear that you will introduce undetected bugs.  
Nếu bạn không giữ sạch các bài kiểm tra của mình, bạn sẽ mất chúng. Và không có chúng, bạn sẽ mất đi thứ giữ cho mã sản xuất của bạn linh hoạt. Vâng, bạn đã đọc dúng điều đó. Các bài kiểm tra đơn vị giúp mã của chúng tôi linh hoạt, có thể bảo trì và có thể tái sử dụng. Lý do rất đơn giản. Nếu bạn có các bài kiểm tra, bạn không sợ phải thay đổi mã! Nếu không có kiểm tra, mọi thay đổi đều có thể xảy ra lỗi. Cho dù kiến trúc của bạn linh hoạt đến đâu, cho dù thiết kế của bạn được phân vùng độc đáo đến đâu, nếu không có các bài kiểm tra, bạn sẽ miễn cưỡng thực hiện các thay đổi vì sợ rằng bạn sẽ đưa ra các lỗi chưa được phát hiện.

But *with* tests that fear virtually disappears. The higher your test coverage, the less your fear. You can make changes with near impunity to code that has a less than stellar architecture and a tangled and opaque design. Indeed, you can *improve* that architecture and design without fear!  
Nhưng với các bài kiểm tra, nỗi sợ hãi hầu như biến mất. Phạm vi kiểm tra của bạn càng cao, nỗi sợ hãi của bạn càng ít. Bạn có thể thực hiện các thay đổi mà gần như không bị trừng phạt đối với mã có kiến trúc kém xuất sắc và thiết kế rối rắm và không rõ ràng. Thật vậy, bạn có thể cải thiện kiến trúc và thiết kế đó mà không sợ hãi!

So having an automated suite of unit tests that cover the production code is the key to keeping your design and architecture as clean as possible. Tests enable all the -ilities, because tests enable *change*.  
Vì vậy, có một bộ thử nghiệm đơn vị tự động bao gồm mã sản xuất là chìa khóa để giữ cho thiết kế và kiến trúc của bạn sạch sẽ nhất có thể. Các thử nghiệm cho phép tất cả các khả năng, bởi vì các thử nghiệm cho phép thay đổi.

So if your tests are dirty, then your ability to change your code is hampered, and you begin to lose the ability to improve the structure of that code. The dirtier your tests, the dirtier your code becomes. Eventually you lose the tests, and your code rots.  
Vì vậy, nếu các bài kiểm tra của bạn bị bẩn, thì khả năng thay đổi mã của bạn sẽ bị cản trở và bạn bắt đầu mất khả năng cải thiện cấu trúc của mã đó. Các bài kiểm tra của bạn càng bẩn, mã của bạn càng trở nên bẩn hơn. Cuối cùng, bạn làm mất các bài kiểm tra và mã của bạn bị hỏng.

[**Clean Tests**](#_page_311_0)  
Kiểm tra sạch

What makes a clean test? Three things. Readability, readability, and readability. Read-ability is perhaps even more important in unit tests than it is in production code. What makes tests readable? The same thing that makes all code readable: clarity, simplicity, and density of expression. In a test you want to say a lot with as few expressions as possible.  
Điều gì làm cho một bài kiểm tra sạch? Ba thứ. Khả năng đọc, khả năng đọc và khả năng đọc. Khả năng đọc có lẽ còn quan trọng hơn trong các bài kiểm tra đơn vị so với mã sản xuất. Điều gì làm cho các bài kiểm tra có thể đọc được? Điều tương tự làm cho tất cả mã có thể đọc được: rõ ràng, đơn giản và mật độ biểu đạt. Trong một bài kiểm tra, bạn muốn nói nhiều với càng ít cách diễn đạt càng tốt.

Consider the code from FitNesse in Listing 9-1. These three tests are difﬁcult to understand and can certainly be improved. First, there is a terrible amount of duplicate code [G5] in the repeated calls to addPageand assertSubString. More importantly, this code is just loaded with details that interfere with the expressiveness of the test.  
Hãy xem xét đoạn mã từ FitNesse trong Liệt kê 9-1. Ba bài kiểm tra này rất khó hiểu và chắc chắn có thể được cải thiện. Đầu tiên, có một số lượng lớn mã trùng lặp [G5] trong các lần gọi lặp lại addPage và assertSubString. Quan trọng hơn, mã này chỉ được tải với các chi tiết cản trở tính biểu cảm của bài kiểm tra.

<https://www.facebook.com/groups/phattrienlaptrinh>

**Clean Tests** 125  
Kiểm tra sạch 125

**Listing 9-1 SerializedPageResponderTest.java  
Liệt kê 9-1 SerializedPageResponderTest.java**

public void testGetPageHieratchyAsXml() throws Exception {

crawler.addPage(root, PathParser.parse("PageOne")); crawler.addPage(root, PathParser.parse("PageOne.ChildOne")); crawler.addPage(root, PathParser.parse("PageTwo"));

request.setResource("root"); request.addInput("type", "pages");

Responder responder = new SerializedPageResponder(); SimpleResponse response =

(SimpleResponse) responder.makeResponse( new FitNesseContext(root), request);

String xml = response.getContent();

assertEquals("text/xml", response.getContentType()); assertSubString("<name>PageOne</name>", xml); assertSubString("<name>PageTwo</name>", xml); assertSubString("<name>ChildOne</name>", xml);

}

public void testGetPageHieratchyAsXmlDoesntContainSymbolicLinks() throws Exception

{

WikiPage pageOne = crawler.addPage(root, PathParser.parse("PageOne")); crawler.addPage(root, PathParser.parse("PageOne.ChildOne")); crawler.addPage(root, PathParser.parse("PageTwo"));

PageData data = pageOne.getData(); WikiPageProperties properties = data.getProperties();

WikiPageProperty symLinks = properties.set(SymbolicPage.PROPERTY\_NAME); symLinks.set("SymPage", "PageTwo");

pageOne.commit(data);

request.setResource("root"); request.addInput("type", "pages");

Responder responder = new SerializedPageResponder(); SimpleResponse response =

(SimpleResponse) responder.makeResponse( new FitNesseContext(root), request);

String xml = response.getContent();

assertEquals("text/xml", response.getContentType()); assertSubString("<name>PageOne</name>", xml); assertSubString("<name>PageTwo</name>", xml); assertSubString("<name>ChildOne</name>", xml); assertNotSubString("SymPage", xml);

}

public void testGetDataAsHtml() throws Exception {

crawler.addPage(root, PathParser.parse("TestPageOne"), "test page");

request.setResource("TestPageOne"); request.addInput("type", "data");

<https://www.facebook.com/groups/phattrienlaptrinh>

126 **Chapter 9: Unit Tests**  
126 Chương 9: Bài kiểm tra đơn vị

**Listing 9-1 (continued) SerializedPageResponderTest.java**  
Liệt kê 9-1 (tiếp theo) SerializedPageResponderTest.java

Responder responder = new SerializedPageResponder(); SimpleResponse response =

(SimpleResponse) responder.makeResponse( new FitNesseContext(root), request);

String xml = response.getContent();

assertEquals("text/xml", response.getContentType()); assertSubString("test page", xml); assertSubString("<Test", xml);

}

For example, look at the PathParser calls. They transform strings into PagePath instances used by the crawlers. This transformation is completely irrelevant to the test at hand and serves only to obfuscate the intent. The details surrounding the creation of the responderand the gathering and casting of the responseare also just noise. Then there’s the ham-handed way that the request URL is built from a resource and an argument. (I helped write this code, so I feel free to roundly criticize it.)  
Ví dụ: hãy xem các cuộc gọi PathParser. Chúng biến đổi các chuỗi thành các phiên bản PagePath được trình thu thập thông tin sử dụng. Sự chuyển đổi này hoàn toàn không liên quan đến bài kiểm tra hiện tại và chỉ nhằm mục đích che giấu ý định. Các chi tiết xung quanh việc tạo phản hồi và thu thập và truyền phản hồi cũng chỉ là tiếng ồn. Sau đó, có một cách thủ công mà URL yêu cầu được tạo từ một tài nguyên và một đối số. (Tôi đã giúp viết mã này, vì vậy tôi có thể thoải mái chỉ trích nó.)

In the end, this code was not designed to be read. The poor reader is inundated with a swarm of details that must be understood before the tests make any real sense.  
Cuối cùng, mã này không được thiết kế để đọc. Người đọc kém bị ngập trong một loạt các chi tiết phải được hiểu trước khi các bài kiểm tra thực sự có ý nghĩa.

Now consider the improved tests in Listing 9-2. These tests do the exact same thing, but they have been refactored into a much cleaner and more explanatory form.  
Bây giờ hãy xem xét các kiểm tra cải tiến trong Liệt kê 9-2. Các bài kiểm tra này thực hiện chính xác điều tương tự, nhưng chúng đã được cấu trúc lại thành một dạng rõ ràng hơn và dễ giải thích hơn.

**Listing 9-2  
Liệt kê 9-2**

**SerializedPageResponderTest.java (refactored)  
SerializedPageResponderTest.java (đã tái cấu trúc)**

public void testGetPageHierarchyAsXml() throws Exception { makePages("PageOne", "PageOne.ChildOne", "PageTwo");

submitRequest("root", "type:pages");

assertResponseIsXML(); assertResponseContains(

"<name>PageOne</name>", "<name>PageTwo</name>", "<name>ChildOne</name>" );

}

public void testSymbolicLinksAreNotInXmlPageHierarchy() throws Exception { WikiPage page = makePage("PageOne");

makePages("PageOne.ChildOne", "PageTwo");

addLinkTo(page, "PageTwo", "SymPage");

submitRequest("root", "type:pages");

assertResponseIsXML(); assertResponseContains(

"<name>PageOne</name>", "<name>PageTwo</name>", "<name>ChildOne</name>" );

assertResponseDoesNotContain("SymPage"); }

<https://www.facebook.com/groups/phattrienlaptrinh>

**Clean Tests** 127  
Kiểm tra sạch 127

**Listing 9-2 (continued) SerializedPageResponderTest.java (refactored)**  
Liệt kê 9-2 (tiếp theo) SerializedPageResponderTest.java (đã tái cấu trúc)

public void testGetDataAsXml() throws Exception { makePageWithContent("TestPageOne", "test page");

submitRequest("TestPageOne", "type:data");

assertResponseIsXML(); assertResponseContains("test page", "<Test");

}

The BUILD-OPERATE-CHECK2 pattern is made obvious by the structure of these tests. Each of the tests is clearly split into three parts. The ﬁrst part builds up the test data, the second part operates on that test data, and the third part checks that the operation yielded the expected results.  
Mô hình XÂY DỰNG-VẬN HÀNH-KIỂM TRA2 được thể hiện rõ ràng nhờ cấu trúc của các thử nghiệm này. Mỗi bài kiểm tra được chia thành ba phần rõ ràng. Phần đầu tiên xây dựng dữ liệu thử nghiệm, phần thứ hai hoạt động trên dữ liệu thử nghiệm đó và phần thứ ba kiểm tra xem hoạt động có mang lại kết quả như mong đợi hay không.

Notice that the vast majority of annoying detail has been eliminated. The tests get right to the point and use only the data types and functions that they truly need. Anyone who reads these tests should be able to work out what they do very quickly, without being misled or overwhelmed by details.  
Lưu ý rằng phần lớn các chi tiết khó chịu đã được loại bỏ. Các bài kiểm tra đi thẳng vào vấn đề và chỉ sử dụng các loại dữ liệu và chức năng mà chúng thực sự cần. Bất kỳ ai đọc các bài kiểm tra này đều có thể nhanh chóng tìm ra những gì họ làm mà không bị nhầm lẫn hoặc choáng ngợp bởi các chi tiết.

[**Domain-Speciﬁc Testing Language**](#_page_311_0)  
Ngôn ngữ kiểm tra dành riêng cho miền

The tests in Listing 9-2 demonstrate the technique of building a domain-speciﬁc language for your tests. Rather than using the APIs that programmers use to manipulate the sys-tem, we build up a set of functions and utilities that make use of those APIs and that make the tests more convenient to write and easier to read. These functions and utilities become a specialized API used by the tests. They are a testing *language* that program-mers use to help themselves to write their tests and to help those who must read those tests later on.  
Các bài kiểm tra trong Liệt kê 9-2 thể hiện kỹ thuật xây dựng một ngôn ngữ dành riêng cho miền cho các bài kiểm tra của bạn. Thay vì sử dụng các API mà các lập trình viên sử dụng để thao tác với hệ thống, chúng tôi xây dựng một tập hợp các chức năng và tiện ích sử dụng các API đó và làm cho các bài kiểm tra viết thuận tiện hơn và dễ đọc hơn. Các chức năng và tiện ích này trở thành một API chuyên biệt được sử dụng bởi các bài kiểm tra. Chúng là một ngôn ngữ thử nghiệm mà những người lập trình sử dụng để giúp họ viết các bài kiểm tra của họ và để giúp những người phải đọc các bài kiểm tra đó sau này.

This testing API is not designed up front; rather it evolves from the continued refac-toring of test code that has gotten too tainted by obfuscating detail. Just as you saw me refactor Listing 9-1 into Listing 9-2, so too will disciplined developers refactor their test code into more succinct and expressive forms.  
API thử nghiệm này không được thiết kế trước; thay vào đó, nó phát triển từ việc tiếp tục cấu trúc lại mã kiểm tra đã trở nên quá bẩn bởi các chi tiết khó hiểu. Giống như bạn đã thấy tôi cấu trúc lại Liệt kê 9-1 thành Liệt kê 9-2, các nhà phát triển có kỷ luật cũng sẽ cấu trúc lại mã thử nghiệm của họ thành các dạng ngắn gọn và biểu cảm hơn.

[**A Dual Standard**](#_page_311_0)  
Tiêu chuẩn kép

In one sense the team I mentioned at the beginning of this chapter had things right. The code within the testing API *does* have a different set of engineering standards than produc-tion code. It must still be simple, succinct, and expressive, but it need not be as efﬁcient as production code. After all, it runs in a test environment, not a production environment, and those two environment have very different needs.  
Ở một khía cạnh nào đó, nhóm nghiên cứu mà tôi đề cập ở đầu chương này đã có những điều đúng đắn. Mã trong API thử nghiệm có một bộ tiêu chuẩn kỹ thuật khác với mã sản xuất. Nó vẫn phải đơn giản, ngắn gọn và biểu cảm, nhưng nó không nhất thiết phải hiệu quả như mã sản xuất. Rốt cuộc, nó chạy trong môi trường thử nghiệm, không phải môi trường sản xuất và hai môi trường đó có nhu cầu rất khác nhau.

2. [http://ﬁtnesse.org/FitNesse.AcceptanceTestPatterns](http://fitnesse.org/FitNesse.AcceptanceTestPatterns)

<https://www.facebook.com/groups/phattrienlaptrinh>

128 **Chapter 9: Unit Tests**  
128 Chương 9: Bài kiểm tra đơn vị

Consider the test in Listing 9-3. I wrote this test as part of an environment control sys-tem I was prototyping. Without going into the details you can tell that this test checks that the low temperature alarm, the heater, and the blower are all turned on when the tempera-ture is “way too cold.”  
Hãy xem xét bài kiểm tra trong Liệt kê 9-3. Tôi đã viết thử nghiệm này như một phần của hệ thống kiểm soát môi trường mà tôi đang tạo mẫu. Không đi sâu vào chi tiết, bạn có thể nói rằng thử nghiệm này kiểm tra xem báo động nhiệt độ thấp, lò sưởi và quạt gió có được bật khi nhiệt độ “quá lạnh”.

**Listing 9-3 EnvironmentControllerTest.java  
Liệt kê 9-3 EnvironmentControllerTest.java**

@Test

public void turnOnLoTempAlarmAtThreashold() throws Exception { hw.setTemp(WAY\_TOO\_COLD);

controller.tic(); assertTrue(hw.heaterState()); assertTrue(hw.blowerState()); assertFalse(hw.coolerState()); assertFalse(hw.hiTempAlarm()); assertTrue(hw.loTempAlarm());

}

There are, of course, lots of details here. For example, what is that ticfunction all about? In fact, I’d rather you not worry about that while reading this test. I’d rather you just worry about whether you agree that the end state of the system is consistent with the tem-perature being “way too cold.”  
Tất nhiên, có rất nhiều chi tiết ở đây. Ví dụ, chức năng tic đó là gì? Trên thực tế, tôi muốn bạn đừng lo lắng về điều đó khi đọc bài kiểm tra này. Tôi muốn bạn chỉ lo lắng về việc liệu bạn có đồng ý rằng trạng thái cuối cùng của hệ thống phù hợp với nhiệt độ “quá lạnh” hay không.

Notice, as you read the test, that your eye needs to bounce back and forth between the name of the state being checked, and the *sense* of the state being checked. You see heaterState, and then your eyes glissade left to assertTrue. You see coolerStateand your eyes must track left to assertFalse. This is tedious and unreliable. It makes the test hard to read.  
Lưu ý rằng khi bạn đọc bài kiểm tra, mắt của bạn cần phải đảo qua đảo lại giữa tên của trạng thái đang được kiểm tra và cảm giác về trạng thái đang được kiểm tra. Bạn nhìn thấy heatState, và sau đó mắt bạn lướt sang trái để khẳng địnhTrue. Bạn thấy trạng thái mát hơn và mắt của bạn phải theo dõi bên trái để khẳng định Sai. Điều này là tẻ nhạt và không đáng tin cậy. Nó làm cho bài kiểm tra khó đọc.

I improved the reading of this test greatly by transforming it into Listing 9-4.  
Tôi đã cải thiện đáng kể việc đọc bài kiểm tra này bằng cách biến nó thành Liệt kê 9-4.

**Listing 9-4**

**EnvironmentControllerTest.java (refactored)  
Môi trườngControllerTest.java (tái cấu trúc)**

@Test

public void turnOnLoTempAlarmAtThreshold() throws Exception {

wayTooCold();

assertEquals("HBchL", hw.getState()); }

Of course I hid the detail of the ticfunction by creating a wayTooColdfunction. But the thing to note is the strange string in the assertEquals. Upper case means “on,” lower case means “off,” and the letters are always in the following order: {heater, blower, cooler, hi-temp-alarm, lo-temp-alarm}.  
Tất nhiên tôi đã giấu chi tiết của hàm tic bằng cách tạo hàm wayTooCold. Nhưng điều cần lưu ý là chuỗi lạ trong assertEquals. Chữ hoa có nghĩa là “bật”, chữ thường có nghĩa là “tắt” và các chữ cái luôn theo thứ tự sau: {máy sưởi, quạt gió, máy làm mát, báo nhiệt độ cao, báo động nhiệt độ thấp}.

Even though this is close to a violation of the rule about mental mapping,3 it seems appropriate in this case. Notice, once you know the meaning, your eyes glide across  
Mặc dù điều này gần như vi phạm quy tắc về lập bản đồ tinh thần,3 nó có vẻ phù hợp trong trường hợp này. Chú ý, khi đã biết nghĩa, mắt lướt ngang

3. “Avoid Mental Mapping” on page 25.  
3. “Tránh lập bản đồ tư duy” ở trang 25.

<https://www.facebook.com/groups/phattrienlaptrinh>

**Clean Tests** 129  
Kiểm tra sạch 129

that string and you can quickly interpret the results. Reading the test becomes almost a pleasure. Just take a look at Listing 9-5 and see how easy it is to understand these tests.  
chuỗi đó và bạn có thể nhanh chóng giải thích kết quả. Đọc bài kiểm tra gần như trở thành một niềm vui. Chỉ cần nhìn vào Liệt kê 9-5 và xem việc hiểu các bài kiểm tra này dễ dàng như thế nào.

**Listing 9-5  
Liệt kê 9-5**

**EnvironmentControllerTest.java (bigger selection)  
Môi trườngControllerTest.java (lựa chọn lớn hơn)**

@Test

public void turnOnCoolerAndBlowerIfTooHot() throws Exception { tooHot();

assertEquals("hBChl", hw.getState()); }

@Test

public void turnOnHeaterAndBlowerIfTooCold() throws Exception { tooCold();

assertEquals("HBchl", hw.getState()); }

@Test

public void turnOnHiTempAlarmAtThreshold() throws Exception { wayTooHot();

assertEquals("hBCHl", hw.getState()); }

@Test

public void turnOnLoTempAlarmAtThreshold() throws Exception { wayTooCold();

assertEquals("HBchL", hw.getState()); }

The getStatefunction is shown in Listing 9-6. Notice that this is not very efﬁcient code. To make it efﬁcient, I probably should have used a StringBuffer.  
Hàm getState được hiển thị trong Liệt kê 9-6. Lưu ý rằng đây không phải là mã rất hiệu quả. Để làm cho nó hiệu quả, có lẽ tôi nên sử dụng StringBuffer.

**Listing 9-6 MockControlHardware.java**  
Liệt kê 9-6 MockControlHardware.java

public String getState() { String state = "";

state += heater ? "H" : "h"; state += blower ? "B" : "b"; state += cooler ? "C" : "c"; state += hiTempAlarm ? "H" : "h"; state += loTempAlarm ? "L" : "l"; return state;

}

StringBuffers are a bit ugly. Even in production code I will avoid them if the cost is small; and you could argue that the cost of the code in Listing 9-6 is very small. However, this application is clearly an embedded real-time system, and it is likely that computer and memory resources are very constrained. The *test* environment, however, is not likely to be constrained at all.  
StringBuffers hơi xấu. Ngay cả trong mã sản xuất, tôi sẽ tránh chúng nếu chi phí nhỏ; và bạn có thể lập luận rằng chi phí của mã trong Liệt kê 9-6 là rất nhỏ. Tuy nhiên, ứng dụng này rõ ràng là một hệ thống thời gian thực được nhúng và có khả năng tài nguyên máy tính và bộ nhớ rất hạn chế. Tuy nhiên, môi trường thử nghiệm dường như không bị hạn chế chút nào.

<https://www.facebook.com/groups/phattrienlaptrinh>

130 **Chapter 9: Unit Tests**  
130 Chương 9: Bài kiểm tra đơn vị

That is the nature of the dual standard. There are things that you might never do in a production environment that are perfectly ﬁne in a test environment. Usually they involve issues of memory or CPU efﬁciency. But they *never* involve issues of cleanliness.  
Đó là bản chất của tiêu chuẩn kép. Có những thứ bạn có thể không bao giờ làm được trong môi trường sản xuất lại hoàn toàn ổn trong môi trường thử nghiệm. Thông thường chúng liên quan đến các vấn đề về bộ nhớ hoặc hiệu suất của CPU. Nhưng họ không bao giờ liên quan đến vấn đề sạch sẽ.

[**One Assert per Test**](#_page_311_0)  
Một xác nhận cho mỗi bài kiểm tra

There is a school of thought4 that says that every test function in a JUnit test should have one and only one assert statement. This rule may seem draconian, but the advantage can be seen in Listing 9-5. Those tests come to a single conclusion that is quick and easy to understand.  
Có một trường phái tư tưởng4 nói rằng mọi chức năng kiểm tra trong bài kiểm tra JUnit nên có một và chỉ một câu lệnh khẳng định. Quy tắc này có vẻ hà khắc, nhưng lợi thế có thể được nhìn thấy trong Liệt kê 9-5. Những bài kiểm tra đó đưa ra một kết luận duy nhất nhanh chóng và dễ hiểu.

But what about Listing 9-2? It seems unreasonable that we could somehow easily merge the assertion that the output is XML and that it contains certain substrings. How-ever, we can break the test into two separate tests, each with its own particular assertion, as shown in Listing 9-7.  
Nhưng còn Liệt kê 9-2 thì sao? Có vẻ như không hợp lý khi bằng cách nào đó chúng ta có thể dễ dàng hợp nhất khẳng định rằng đầu ra là XML và nó chứa các chuỗi con nhất định. Tuy nhiên, chúng ta có thể chia bài kiểm tra thành hai bài kiểm tra riêng biệt, mỗi bài kiểm tra có xác nhận cụ thể của riêng nó, như trong Liệt kê 9-7.

**Listing 9-7  
Liệt kê 9-7**

**SerializedPageResponderTest.java (Single Assert)  
SerializedPageResponderTest.java (Xác nhận đơn)**

public void testGetPageHierarchyAsXml() throws Exception { givenPages("PageOne", "PageOne.ChildOne", "PageTwo");

whenRequestIsIssued("root", "type:pages");

thenResponseShouldBeXML(); }

public void testGetPageHierarchyHasRightTags() throws Exception { givenPages("PageOne", "PageOne.ChildOne", "PageTwo");

whenRequestIsIssued("root", "type:pages");

thenResponseShouldContain(

"<name>PageOne</name>", "<name>PageTwo</name>", "<name>ChildOne</name>" );

}

Notice that I have changed the names of the functions to use the common given-when-then5 convention. This makes the tests even easier to read. Unfortunately, splitting the tests as shown results in a lot of duplicate code.  
Lưu ý rằng tôi đã thay đổi tên của các chức năng để sử dụng quy ước đã cho-khi-thì5 chung. Điều này làm cho các bài kiểm tra dễ đọc hơn. Thật không may, việc chia nhỏ các bài kiểm tra như được hiển thị dẫn đến rất nhiều mã trùng lặp.

We can eliminate the duplication by using the TEMPLATE METHOD6 pattern and putting the *given/when* parts in the base class, and the *then* parts in different derivatives. Or we could create a completely separate test class and put the *given* and *when* parts in the @Beforefunc-tion, and the *when* parts in each @Test function. But this seems like too much mechanism for such a minor issue. In the end, I prefer the multiple asserts in Listing 9-2.  
Chúng ta có thể loại bỏ sự trùng lặp bằng cách sử dụng mẫu PHƯƠNG PHÁP TEMPLATE6 và đặt các phần đã cho/khi vào lớp cơ sở và các phần sau đó vào các dẫn xuất khác nhau. Hoặc chúng ta có thể tạo một lớp kiểm tra hoàn toàn riêng biệt và đặt các phần đã cho và khi trong @ Beforefunc-tion và các phần khi trong mỗi chức năng @Test. Nhưng điều này có vẻ như quá nhiều cơ chế cho một vấn đề nhỏ như vậy. Cuối cùng, tôi thích nhiều khẳng định hơn trong Liệt kê 9-2.

4. See DaveAstel’s blog entry: [http://www.artima.com/weblogs/viewpost.jsp?thread=35578](http://www.artima.com/weblogs/viewpost.jsp?thread=35578%20///http://www.artima.com/weblogs/viewpost.jsp?thread=35578\\\) 5. [RSpec].

6. [GOF].

<https://www.facebook.com/groups/phattrienlaptrinh>

**One Assert per Test** 131  
Một xác nhận cho mỗi lần kiểm tra 131

I think the single assert rule is a good guideline.7 I usually try to create a domain-speciﬁc testing language that supports it, as in Listing 9-5. But I am not afraid to put more than one assert in a test. I think the best thing we can say is that the number of asserts in a test ought to be minimized.  
Tôi nghĩ quy tắc khẳng định duy nhất là một hướng dẫn tốt.7 Tôi thường cố gắng tạo một ngôn ngữ thử nghiệm dành riêng cho miền hỗ trợ nó, như trong Liệt kê 9-5. Nhưng tôi không ngại đưa ra nhiều hơn một khẳng định trong một bài kiểm tra. Tôi nghĩ điều tốt nhất chúng ta có thể nói là số lượng xác nhận trong một bài kiểm tra phải được giảm thiểu.

[**Single Concept per Test**](#_page_311_0)  
Khái niệm duy nhất cho mỗi bài kiểm tra

Perhaps a better rule is that we want to test a single concept in each test function. We don’t want long test functions that go testing one miscellaneous thing after another. Listing 9-8 is an example of such a test. This test should be split up into three independent tests because it tests three independent things. Merging them all together into the same function forces the reader to ﬁgure out why each section is there and what is being tested by that section.  
Có lẽ một quy tắc tốt hơn là chúng tôi muốn kiểm tra một khái niệm duy nhất trong mỗi chức năng kiểm tra. Chúng tôi không muốn các chức năng kiểm tra dài chạy thử hết thứ linh tinh này đến thứ khác. Liệt kê 9-8 là một ví dụ về phép thử như vậy. Bài kiểm tra này nên được chia thành ba bài kiểm tra độc lập vì nó kiểm tra ba điều độc lập. Việc hợp nhất tất cả chúng lại với nhau thành cùng một chức năng buộc người đọc phải tìm ra lý do tại sao mỗi phần lại ở đó và phần đó đang kiểm tra điều gì.

**Listing 9-8**

/\*\*

\* Miscellaneous tests for the addMonths() method. \*/

public void testAddMonths() {

SerialDate d1 = SerialDate.createInstance(31, 5, 2004);

SerialDate d2 = SerialDate.addMonths(1, d1); assertEquals(30, d2.getDayOfMonth()); assertEquals(6, d2.getMonth()); assertEquals(2004, d2.getYYYY());

SerialDate d3 = SerialDate.addMonths(2, d1); assertEquals(31, d3.getDayOfMonth()); assertEquals(7, d3.getMonth()); assertEquals(2004, d3.getYYYY());

SerialDate d4 = SerialDate.addMonths(1, SerialDate.addMonths(1, d1)); assertEquals(30, d4.getDayOfMonth());

assertEquals(7, d4.getMonth()); assertEquals(2004, d4.getYYYY());

}

The three test functions probably ought to be like this:  
Ba chức năng kiểm tra có lẽ nên như thế này:

• *Given* the last day of a month with 31 days (like May):  
• Cho ngày cuối tháng có 31 ngày (như tháng 5):

**1.** *When* you add one month, such that the last day of that month is the 30th (like June), *then* the date should be the 30th of that month, not the 31st.  
1. Khi bạn thêm một tháng, sao cho ngày cuối cùng của tháng đó là ngày 30 (như tháng 6), thì ngày đó phải là ngày 30 của tháng đó, không phải ngày 31.

**2.** *When* you add two months to that date, such that the ﬁnal month has 31 days, *then* the date should be the 31st.  
2. Khi bạn thêm hai tháng vào ngày đó, sao cho tháng cuối cùng có 31 ngày, thì ngày đó sẽ là ngày 31.

7. “Keep to the code!”  
7. “Giữ nguyên tắc!”

<https://www.facebook.com/groups/phattrienlaptrinh>

132 **Chapter 9: Unit Tests**  
132 Chương 9: Bài kiểm tra đơn vị

• *Given* the last day of a month with 30 days in it (like June):  
• Cho ngày cuối cùng của tháng có 30 ngày (như tháng 6):

**1.** *When* you add one month such that the last day of that month has 31 days, *then* the date should be the 30th, not the 31st.  
1. Khi bạn thêm một tháng sao cho ngày cuối cùng của tháng đó có 31 ngày, thì ngày đó phải là ngày 30 chứ không phải ngày 31.

Stated like this, you can see that there is a general rule hiding amidst the miscella-neous tests. When you increment the month, the date can be no greater than the last day of the month. This implies that incrementing the month on February 28th should yield March 28th.*That* test is missing and would be a useful test to write.  
Nói như thế này, bạn có thể thấy rằng có một quy tắc chung ẩn trong các bài kiểm tra không rõ ràng. Khi bạn tăng tháng, ngày không được lớn hơn ngày cuối cùng của tháng. Điều này ngụ ý rằng việc tăng tháng vào ngày 28 tháng 2 sẽ mang lại kết quả là ngày 28 tháng 3. Bài kiểm tra đó bị thiếu và sẽ là một bài kiểm tra hữu ích để viết.

So it’s not the multiple asserts in each section of Listing 9-8 that causes the problem. Rather it is the fact that there is more than one concept being tested. So probably the best rule is that you should minimize the number of asserts per concept and test just one con-cept per test function.  
Vì vậy, không phải nhiều xác nhận trong mỗi phần của Liệt kê 9-8 gây ra vấn đề. Thay vào đó, thực tế là có nhiều hơn một khái niệm đang được thử nghiệm. Vì vậy, có lẽ quy tắc tốt nhất là bạn nên giảm thiểu số lượng xác nhận cho mỗi khái niệm và chỉ kiểm tra một khái niệm cho mỗi chức năng kiểm tra.

[**F.I.R.S.T.8**](#_page_311_0)  
F.I.R.S.T.8

Clean tests follow ﬁve other rules that form the above acronym:  
Clean tests tuân theo năm quy tắc khác tạo thành từ viết tắt ở trên:

**Fast** Tests should be fast. They should run quickly. When tests run slow, you won’t want to run them frequently. If you don’t run them frequently, you won’t ﬁnd problems early enough to ﬁx them easily.You won’t feel as free to clean up the code. Eventually the code will begin to rot.  
Kiểm tra nhanh nên nhanh. Họ nên chạy nhanh. Khi các bài kiểm tra chạy chậm, bạn sẽ không muốn chạy chúng thường xuyên. Nếu bạn không chạy chúng thường xuyên, bạn sẽ không phát hiện ra các vấn đề đủ sớm để khắc phục chúng một cách dễ dàng. Bạn sẽ không cảm thấy thoải mái khi dọn dẹp mã. Cuối cùng, mã sẽ bắt đầu thối rữa.

**Independent** Tests should not depend on each other. One test should not set up the condi-tions for the next test. You should be able to run each test independently and run the tests in any order you like. When tests depend on each other, then the ﬁrst one to fail causes a cas-cade of downstream failures, making diagnosis difﬁcult and hiding downstream defects.  
Kiểm tra độc lập không nên phụ thuộc vào nhau. Một thử nghiệm không nên thiết lập các điều kiện cho thử nghiệm tiếp theo. Bạn sẽ có thể chạy từng thử nghiệm một cách độc lập và chạy thử nghiệm theo bất kỳ thứ tự nào bạn muốn. Khi các thử nghiệm phụ thuộc lẫn nhau, thì thử nghiệm đầu tiên bị lỗi sẽ gây ra một loạt các lỗi hạ nguồn, khiến cho việc chẩn đoán trở nên khó khăn và che giấu các lỗi phía sau.

**Repeatable** Tests should be repeatable in any environment. You should be able to run the tests in the production environment, in the QA environment, and on your laptop while riding home on the train without a network. If your tests aren’t repeatable in any environ-ment, then you’ll always have an excuse for why they fail. You’ll also ﬁnd yourself unable to run the tests when the environment isn’t available.  
Các bài kiểm tra có thể lặp lại nên được lặp lại trong bất kỳ môi trường nào. Bạn sẽ có thể chạy thử nghiệm trong môi trường sản xuất, trong môi trường QA và trên máy tính xách tay của mình khi đi tàu về nhà mà không có mạng. Nếu các bài kiểm tra của bạn không thể lặp lại trong bất kỳ môi trường nào, thì bạn sẽ luôn có lý do tại sao chúng thất bại. Bạn cũng sẽ thấy mình không thể chạy thử nghiệm khi không có môi trường.

**Self-Validating** The tests should have a boolean output. Either they pass or fail. You should not have to read through a log ﬁle to tell whether the tests pass. You should not have to manually compare two different text ﬁles to see whether the tests pass. If the tests aren’t self-validating, then failure can become subjective and running the tests can require a long manual evaluation.  
Tự xác thực Các bài kiểm tra phải có đầu ra boolean. Hoặc là họ vượt qua hoặc thất bại. Bạn không cần phải đọc qua tệp nhật ký để biết liệu các bài kiểm tra có vượt qua hay không. Bạn không cần phải so sánh hai tệp văn bản khác nhau theo cách thủ công để xem liệu các bài kiểm tra có đạt hay không. Nếu các thử nghiệm không tự xác thực, thì lỗi có thể trở nên chủ quan và việc chạy thử nghiệm có thể yêu cầu đánh giá thủ công trong thời gian dài.

8. Object Mentor Training Materials.  
8. Tài liệu đào tạo Cố vấn đối tượng.

<https://www.facebook.com/groups/phattrienlaptrinh>

**Bibliography** 133  
Thư mục 133

**Timely** The tests need to be written in a timely fashion. Unit tests should be written *just before* the production code that makes them pass. If you write tests after the production code, then you may ﬁnd the production code to be hard to test. You may decide that some production code is too hard to test. You may not design the production code to be testable.  
Kịp thời Các bài kiểm tra cần phải được viết một cách kịp thời. Các bài kiểm tra đơn vị nên được viết ngay trước mã sản xuất giúp chúng vượt qua. Nếu bạn viết các bài kiểm tra sau mã sản xuất, thì bạn có thể thấy mã sản xuất khó kiểm tra. Bạn có thể quyết định rằng một số mã sản xuất quá khó để kiểm tra. Bạn không thể thiết kế mã sản xuất để có thể kiểm tra được.

[**Conclusion**](#_page_311_0)  
Phần kết luận

We have barely scratched the surface of this topic. Indeed, I think an entire book could be written about *clean tests*. Tests are as important to the health of a project as the production code is. Perhaps they are even more important, because tests preserve and enhance the ﬂexibility, maintainability, and reusability of the production code. So keep your tests con-stantly clean. Work to make them expressive and succinct. Invent testing APIs that act as domain-speciﬁc language that helps you write the tests.  
Chúng tôi hầu như không làm trầy xước bề mặt của chủ đề này. Thật vậy, tôi nghĩ có thể viết cả một cuốn sách về clean tests. Các bài kiểm tra cũng quan trọng đối với sức khỏe của một dự án giống như mã sản xuất. Có lẽ chúng thậm chí còn quan trọng hơn, bởi vì các bài kiểm tra bảo tồn và nâng cao tính linh hoạt, khả năng bảo trì và khả năng sử dụng lại của mã sản xuất. Vì vậy, hãy giữ cho các bài kiểm tra của bạn liên tục sạch sẽ. Làm việc để làm cho chúng biểu cảm và cô đọng. Phát minh các API thử nghiệm hoạt động như ngôn ngữ dành riêng cho miền giúp bạn viết các bài kiểm tra.

If you let the tests rot, then your code will rot too. Keep your tests clean.  
Nếu bạn để các bài kiểm tra bị hỏng, thì mã của bạn cũng sẽ bị hỏng. Giữ cho bài kiểm tra của bạn sạch sẽ.

[**Bibliography**](#_page_311_0)  
Thư mục

**[RSpec]:** *RSpec: Behavior Driven Development for Ruby Programmers*, Aslak Hellesøy, David Chelimsky, Pragmatic Bookshelf, 2008.

**[GOF]:** *Design Patterns: Elements of Reusable Object Oriented Software*, Gamma et al., Addison-Wesley, 1996.

<https://www.facebook.com/groups/phattrienlaptrinh>

*This page intentionally left blank*

<https://www.facebook.com/groups/phattrienlaptrinh>

[**10**](#_page_311_0)

[**Classes**](#_page_311_0)  
Các lớp học

with Jeff Langr  
với Jeff Langr

So far in this book we have focused on how to write lines and blocks of code well. We have delved into proper composition of functions and how they interrelate. But for all the atten-tion to the expressiveness of code statements and the functions they comprise, we still don’t have clean code until we’ve paid attention to higher levels of code organization. Let’s talk about clean classes.  
Cho đến giờ trong cuốn sách này, chúng ta đã tập trung vào cách viết tốt các dòng và khối mã. Chúng tôi đã đi sâu vào thành phần thích hợp của các chức năng và cách chúng liên quan đến nhau. Nhưng đối với tất cả sự chú ý đến tính biểu cảm của các câu lệnh mã và các chức năng mà chúng bao gồm, chúng ta vẫn không có mã sạch cho đến khi chúng ta chú ý đến các mức tổ chức mã cao hơn. Hãy nói về các lớp học sạch sẽ.

135

<https://www.facebook.com/groups/phattrienlaptrinh>

136 **Chapter 10: Classes**  
136 Chương 10: Lớp học

[**Class Organization**](#_page_311_0)  
Tổ chức lớp học

Following the standard Java convention, a class should begin with a list of variables. Pub-lic static constants, if any, should come ﬁrst. Then private static variables, followed by pri-vate instance variables. There is seldom a good reason to have a public variable.  
Theo quy ước Java tiêu chuẩn, một lớp nên bắt đầu bằng một danh sách các biến. Các hằng số tĩnh công khai, nếu có, sẽ xuất hiện trước. Sau đó, các biến tĩnh riêng tư, tiếp theo là các biến đối tượng riêng tư. Hiếm khi có một lý do chính đáng để có một biến công khai.

Public functions should follow the list of variables. We like to put the private utilities called by a public function right after the public function itself. This follows the stepdown rule and helps the program read like a newspaper article.  
Các chức năng công cộng nên tuân theo danh sách các biến. Chúng tôi muốn đặt các tiện ích riêng được gọi bởi chức năng công khai ngay sau chính chức năng công cộng. Điều này tuân theo quy tắc giảm dần và giúp chương trình đọc giống như một bài báo.

[**Encapsulation**](#_page_311_0)  
đóng gói

We like to keep our variables and utility functions private, but we’re not fanatic about it. Sometimes we need to make a variable or utility function protected so that it can be accessed by a test. For us, tests rule. If a test in the same package needs to call a function or access a variable, we’ll make it protected or package scope. However, we’ll ﬁrst look for a way to maintain privacy. Loosening encapsulation is always a last resort.  
Chúng tôi muốn giữ các biến và chức năng tiện ích của mình ở chế độ riêng tư, nhưng chúng tôi không cuồng tín về điều đó. Đôi khi chúng ta cần tạo một biến hoặc chức năng tiện ích được bảo vệ để nó có thể được truy cập bằng một bài kiểm tra. Đối với chúng tôi, quy tắc kiểm tra. Nếu một thử nghiệm trong cùng một gói cần gọi một hàm hoặc truy cập một biến, chúng tôi sẽ đặt nó ở phạm vi gói hoặc được bảo vệ. Tuy nhiên, trước tiên chúng ta sẽ tìm cách duy trì quyền riêng tư. Nới lỏng bao bì luôn là phương án cuối cùng.

[**Classes Should Be Small!**](#_page_383_0)  
Các lớp học nên nhỏ!

The ﬁrst rule of classes is that they should be small. The second rule of classes is that they should be smaller than that. No, we’re not going to repeat the exact same text from the *Functions* chapter. But as with functions, smaller is the primary rule when it comes to designing classes. As with functions, our immediate question is always “How small?”  
Nguyên tắc đầu tiên của các lớp học là chúng phải nhỏ. Quy tắc thứ hai của các lớp học là chúng phải nhỏ hơn thế. Không, chúng ta sẽ không lặp lại chính xác nội dung từ chương Hàm. Nhưng đối với các hàm, nhỏ hơn là nguyên tắc cơ bản khi thiết kế các lớp. Đối với các chức năng, câu hỏi ngay lập tức của chúng tôi luôn là "Nhỏ như thế nào?"

With functions we measured size by counting physical lines. With classes we use a different measure. We count *responsibilities.*1  
Với các chức năng, chúng tôi đo kích thước bằng cách đếm các dòng vật lý. Với các lớp học, chúng tôi sử dụng một biện pháp khác. Chúng tôi tính trách nhiệm.1

Listing 10-1 outlines a class, SuperDashboard, that exposes about 70 public methods. Most developers would agree that it’s a bit too super in size. Some developers might refer to SuperDashboard as a “God class.”  
Liệt kê 10-1 phác thảo một lớp, SuperDashboard, hiển thị khoảng 70 phương thức công khai. Hầu hết các nhà phát triển sẽ đồng ý rằng nó có kích thước hơi quá lớn. Một số nhà phát triển có thể gọi SuperDashboard là “Lớp Chúa”.

**Listing 10-1**

**Too Many Responsibilities  
Quá nhiều trách nhiệm**

public class SuperDashboard extends JFrame implements MetaDataUser public String getCustomizerLanguagePath()

public void setSystemConfigPath(String systemConfigPath) public String getSystemConfigDocument()

public void setSystemConfigDocument(String systemConfigDocument) public boolean getGuruState()

public boolean getNoviceState() public boolean getOpenSourceState()

public void showObject(MetaObject object) public void showProgress(String s)

1. [RDD].

<https://www.facebook.com/groups/phattrienlaptrinh>

**Classes Should Be Small!** 137  
Các lớp học nên nhỏ! 137

**Listing 10-1 (continued)**

**Too Many Responsibilities  
Quá nhiều trách nhiệm**

public boolean isMetadataDirty()

public void setIsMetadataDirty(boolean isMetadataDirty) public Component getLastFocusedComponent()

public void setLastFocused(Component lastFocused) public void setMouseSelectState(boolean isMouseSelected) public boolean isMouseSelected()

public LanguageManager getLanguageManager() public Project getProject()

public Project getFirstProject() public Project getLastProject() public String getNewProjectName()

public void setComponentSizes(Dimension dim) public String getCurrentDir()

public void setCurrentDir(String newDir)

public void updateStatus(int dotPos, int markPos) public Class[] getDataBaseClasses()

public MetadataFeeder getMetadataFeeder() public void addProject(Project project)

public boolean setCurrentProject(Project project) public boolean removeProject(Project project) public MetaProjectHeader getProgramMetadata() public void resetDashboard()

public Project loadProject(String fileName, String projectName) public void setCanSaveMetadata(boolean canSave)

public MetaObject getSelectedObject() public void deselectObjects()

public void setProject(Project project)

public void editorAction(String actionName, ActionEvent event) public void setMode(int mode)

public FileManager getFileManager()

public void setFileManager(FileManager fileManager) public ConfigManager getConfigManager()

public void setConfigManager(ConfigManager configManager) public ClassLoader getClassLoader()

public void setClassLoader(ClassLoader classLoader) public Properties getProps()

public String getUserHome() public String getBaseDir()

public int getMajorVersionNumber() public int getMinorVersionNumber() public int getBuildNumber() public MetaObject pasting(

MetaObject target, MetaObject pasted, MetaProject project) public void processMenuItems(MetaObject metaObject)

public void processMenuSeparators(MetaObject metaObject) public void processTabPages(MetaObject metaObject) public void processPlacement(MetaObject object)

public void processCreateLayout(MetaObject object)

public void updateDisplayLayer(MetaObject object, int layerIndex) public void propertyEditedRepaint(MetaObject object)

public void processDeleteObject(MetaObject object) public boolean getAttachedToDesigner()

public void processProjectChangedState(boolean hasProjectChanged) public void processObjectNameChanged(MetaObject object)

public void runProject()

<https://www.facebook.com/groups/phattrienlaptrinh>

138 **Chapter 10: Classes**  
138 Chương 10: Lớp học

**Listing 10-1 (continued)**  
Liệt kê 10-1 (tiếp theo)

**Too Many Responsibilities  
Quá nhiều trách nhiệm**

public void setAçowDragging(boolean allowDragging) public boolean allowDragging()

public boolean isCustomizing() public void setTitle(String title) public IdeMenuBar getIdeMenuBar()

public void showHelper(MetaObject metaObject, String propertyName) // ... many non-public methods follow ...

}

But what if SuperDashboard contained only the methods shown in Listing 10-2?  
Nhưng nếu SuperDashboard chỉ chứa các phương thức được hiển thị trong Liệt kê 10-2 thì sao?

**Listing 10-2 Small Enough?  
Liệt kê 10-2 Đủ nhỏ?**

public class SuperDashboard extends JFrame implements MetaDataUser public Component getLastFocusedComponent()

public void setLastFocused(Component lastFocused) public int getMajorVersionNumber()

public int getMinorVersionNumber() public int getBuildNumber()

}

Five methods isn’t too much, is it? In this case it is because despite its small number of methods, SuperDashboard has too many *responsibilities*.  
Năm phương pháp không phải là quá nhiều, phải không? Trong trường hợp này là do mặc dù có ít phương thức nhưng SuperDashboard có quá nhiều trách nhiệm.

The name of a class should describe what responsibilities it fulﬁlls. In fact, naming is probably the ﬁrst way of helping determine class size. If we cannot derive a concise name for a class, then it’s likely too large. The more ambiguous the class name, the more likely it has too many responsibilities. For example, class names including weasel words like Processor or Manager or Super often hint at unfortunate aggregation of responsibilities.  
Tên của một lớp nên mô tả những trách nhiệm mà nó thực hiện. Trên thực tế, đặt tên có lẽ là cách đầu tiên giúp xác định quy mô lớp học. Nếu chúng ta không thể lấy được một cái tên ngắn gọn cho một lớp, thì nó có thể quá lớn. Tên lớp càng mơ hồ thì càng có nhiều khả năng nó có quá nhiều trách nhiệm. Ví dụ: các tên lớp bao gồm các từ khó hiểu như Bộ xử lý hoặc Người quản lý hoặc Siêu cấp thường gợi ý về sự tập hợp trách nhiệm đáng tiếc.

We should also be able to write a brief description of the class in about 25 words, without using the words “if,” “and,” “or,” or “but.” How would we describe the SuperDashboard? “The SuperDashboardprovides access to the component that last held the focus, and it also allows us to track the version and build numbers.” The ﬁrst “and” is a hint that SuperDashboard has too many responsibilities.  
Chúng ta cũng có thể viết một đoạn mô tả ngắn gọn về lớp học trong khoảng 25 từ, không sử dụng các từ “nếu”, “và”, “hoặc” hoặc “nhưng”. Chúng tôi sẽ mô tả SuperDashboard như thế nào? “SuperDashboard cung cấp quyền truy cập vào thành phần giữ tiêu điểm lần cuối và nó cũng cho phép chúng tôi theo dõi phiên bản và số bản dựng.” Chữ "và" đầu tiên gợi ý rằng SuperDashboard có quá nhiều trách nhiệm.

[**The Single Responsibility Principle**](#_page_383_0)  
Nguyên tắc chịu trách nhiệm duy nhất

The Single Responsibility Principle (SRP)2 states that a class or module should have one, and only one, *reason to change*. This principle gives us both a deﬁnition of responsibility, and a guidelines for class size. Classes should have one responsibility—one reason to change.  
Nguyên tắc Trách nhiệm Đơn lẻ (SRP)2 nêu rõ rằng một lớp hoặc mô-đun nên có một và chỉ một lý do để thay đổi. Nguyên tắc này cho chúng ta cả định nghĩa về trách nhiệm lẫn hướng dẫn về quy mô lớp học. Các lớp nên có một trách nhiệm—một lý do để thay đổi.

2. You can read much more about this principle in [PPP].  
2. Bạn có thể đọc thêm về nguyên tắc này trong [PPP].

<https://www.facebook.com/groups/phattrienlaptrinh>

**Classes Should Be Small!** 139  
Các lớp học nên nhỏ! 139

The seemingly small SuperDashboardclass in Listing 10-2 has two reasons to change. First, it tracks version information that would seemingly need to be updated every time the software gets shipped. Second, it manages Java Swing components (it is a derivative of JFrame, the Swing representation of a top-level GUI window). No doubt we’ll want to update the version number if we change any of the Swing code, but the converse isn’t nec-essarily true: We might change the version information based on changes to other code in the system.  
SuperDashboardclass có vẻ nhỏ trong Liệt kê 10-2 có hai lý do để thay đổi. Đầu tiên, nó theo dõi thông tin phiên bản dường như cần được cập nhật mỗi khi phần mềm được xuất xưởng. Thứ hai, nó quản lý các thành phần Java Swing (nó là một dẫn xuất của JFrame, biểu diễn Swing của cửa sổ GUI cấp cao nhất). Chắc chắn chúng ta sẽ muốn cập nhật số phiên bản nếu chúng ta thay đổi bất kỳ mã Swing nào, nhưng điều ngược lại không nhất thiết đúng: Chúng ta có thể thay đổi thông tin phiên bản dựa trên những thay đổi đối với mã khác trong hệ thống.

Trying to identify responsibilities (reasons to change) often helps us recognize and create better abstractions in our code. We can easily extract all three SuperDashboard methods that deal with version information into a separate class named Version.(See Listing 10-3.) The Versionclass is a construct that has a high potential for reuse in other applications!  
Cố gắng xác định trách nhiệm (lý do thay đổi) thường giúp chúng tôi nhận ra và tạo ra sự trừu tượng hóa tốt hơn trong mã của chúng tôi. Chúng ta có thể dễ dàng trích xuất cả ba phương thức SuperDashboard xử lý thông tin phiên bản vào một lớp riêng biệt có tên là Phiên bản. (Xem Liệt kê 10-3.) Lớp Phiên bản là một cấu trúc có tiềm năng sử dụng lại cao trong các ứng dụng khác!

**Listing 10-3**

**A single-responsibility class  
Một lớp trách nhiệm duy nhất**

public class Version {

public int getMajorVersionNumber() public int getMinorVersionNumber() public int getBuildNumber()

}

SRP is one of the more important concept in OO design. It’s also one of the simpler concepts to understand and adhere to. Yet oddly, SRP is often the most abused class design principle. We regularly encounter classes that do far too many things. Why?  
SRP là một trong những khái niệm quan trọng hơn trong thiết kế OO. Nó cũng là một trong những khái niệm đơn giản hơn để hiểu và tuân theo. Tuy nhiên, thật kỳ lạ, SRP thường là nguyên tắc thiết kế lớp bị lạm dụng nhiều nhất. Chúng tôi thường xuyên bắt gặp các lớp làm quá nhiều việc. Tại sao?

Getting software to work and making software clean are two very different activities. Most of us have limited room in our heads, so we focus on getting our code to work more than organization and cleanliness. This is wholly appropriate. Maintaining a separation of concerns is just as important in our programming *activities* as it is in our programs.  
Bắt phần mềm hoạt động và làm sạch phần mềm là hai hoạt động rất khác nhau. Hầu hết chúng ta đều có không gian hạn chế trong đầu, vì vậy chúng ta tập trung vào việc làm cho mã của mình hoạt động nhiều hơn là tính tổ chức và sự sạch sẽ. Điều này là hoàn toàn phù hợp. Duy trì sự tách biệt giữa các mối quan tâm cũng quan trọng trong các hoạt động lập trình của chúng tôi cũng như trong các chương trình của chúng tôi.

The problem is that too many of us think that we are done once the program works. We fail to switch to the *other* concern of organization and cleanliness. We move on to the next problem rather than going back and breaking the overstuffed classes into decoupled units with single responsibilities.  
Vấn đề là quá nhiều người trong chúng ta nghĩ rằng chúng ta đã hoàn thành một khi chương trình hoạt động. Chúng tôi không chuyển sang mối quan tâm khác về tổ chức và sự sạch sẽ. Chúng tôi chuyển sang vấn đề tiếp theo thay vì quay lại và chia các lớp bị nhồi nhét quá nhiều thành các đơn vị tách rời với các trách nhiệm duy nhất.

At the same time, many developers fear that a large number of small, single-purpose classes makes it more difﬁcult to understand the bigger picture. They are concerned that they must navigate from class to class in order to ﬁgure out how a larger piece of work gets accomplished.  
Đồng thời, nhiều nhà phát triển lo ngại rằng một số lượng lớn các lớp nhỏ, đơn mục đích sẽ khiến việc hiểu bức tranh lớn hơn trở nên khó khăn hơn. Họ lo ngại rằng họ phải điều hướng từ lớp này sang lớp khác để tìm ra cách hoàn thành một phần công việc lớn hơn.

However, a system with many small classes has no more moving parts than a system with a few large classes. There is just as much to learn in the system with a few large classes. So the question is: Do you want your tools organized into toolboxes with many small drawers each containing well-deﬁned and well-labeled components? Or do you want a few drawers that you just toss everything into?  
Tuy nhiên, một hệ thống có nhiều lớp nhỏ không có nhiều bộ phận chuyển động hơn một hệ thống có một vài lớp lớn. Có rất nhiều thứ để học trong hệ thống với một vài lớp học lớn. Vì vậy, câu hỏi đặt ra là: Bạn có muốn các công cụ của mình được sắp xếp thành các hộp công cụ với nhiều ngăn kéo nhỏ, mỗi ngăn chứa các bộ phận được xác định rõ ràng và được dán nhãn rõ ràng không? Hay bạn muốn có một vài ngăn kéo mà bạn có thể ném mọi thứ vào?

Every sizable system will contain a large amount of logic and complexity. The pri-mary goal in managing such complexity is to *organize* it so that a developer knows where  
Mỗi hệ thống lớn sẽ chứa một lượng lớn logic và độ phức tạp. Mục tiêu chính trong việc quản lý sự phức tạp như vậy là tổ chức nó để nhà phát triển biết nơi

<https://www.facebook.com/groups/phattrienlaptrinh>

140 **Chapter 10: Classes**  
140 Chương 10: Lớp học

to look to ﬁnd things and need only understand the directly affected complexity at any given time. In contrast, a system with larger, multipurpose classes always hampers us by insisting we wade through lots of things we don’t need to know right now.  
để tìm kiếm mọi thứ và chỉ cần hiểu sự phức tạp bị ảnh hưởng trực tiếp tại bất kỳ thời điểm nào. Ngược lại, một hệ thống với các lớp học đa năng, lớn hơn luôn cản trở chúng tôi bằng cách khăng khăng đòi chúng tôi phải lướt qua nhiều thứ mà chúng tôi không cần biết ngay bây giờ.

To restate the former points for emphasis: We want our systems to be composed of many small classes, not a few large ones. Each small class encapsulates a single responsi-bility, has a single reason to change, and collaborates with a few others to achieve the desired system behaviors.  
Để nhắc lại các điểm cần nhấn mạnh trước đây: Chúng tôi muốn các hệ thống của chúng tôi bao gồm nhiều lớp nhỏ, chứ không phải một vài lớp lớn. Mỗi lớp nhỏ gói gọn một khả năng chịu trách nhiệm duy nhất, có một lý do duy nhất để thay đổi và cộng tác với một số lớp khác để đạt được các hành vi hệ thống mong muốn.

[**Cohesion**](#_page_383_0)  
Sự gắn kết

Classes should have a small number of instance variables. Each of the methods of a class should manipulate one or more of those variables. In general the more variables a method manipulates the more cohesive that method is to its class. A class in which each variable is used by each method is maximally cohesive.  
Các lớp nên có một số lượng nhỏ các biến thể hiện. Mỗi phương thức của một lớp sẽ thao tác một hoặc nhiều biến đó. Nói chung, một phương thức thao tác càng nhiều biến thì phương thức đó càng gắn kết với lớp của nó. Một lớp trong đó mỗi biến được sử dụng bởi mỗi phương pháp được gắn kết tối đa.

In general it is neither advisable nor possible to create such maximally cohesive classes; on the other hand, we would like cohesion to be high. When cohesion is high, it means that the methods and variables of the class are co-dependent and hang together as a logical whole.  
Nói chung, không nên và cũng không thể tạo ra các lớp gắn kết tối đa như vậy; mặt khác, chúng tôi muốn sự gắn kết cao. Khi sự gắn kết cao, điều đó có nghĩa là các phương thức và biến của lớp đồng phụ thuộc và gắn kết với nhau như một tổng thể hợp lý.

Consider the implementation of a Stackin Listing 10-4. This is a very cohesive class. Of the three methods only size() fails to use both the variables.  
Xem xét việc triển khai Danh sách Stackin 10-4. Đây là một lớp học rất gắn kết. Trong ba phương thức, chỉ có size() không sử dụng được cả hai biến.

**Listing 10-4  
Liệt kê 10-4**

**Stack.java A cohesive class.  
Stack.java Một lớp gắn kết.**

public class Stack {

private int topOfStack = 0;

List<Integer> elements = new LinkedList<Integer>();

public int size() { return topOfStack;

}

public void push(int element) { topOfStack++; elements.add(element);

}

public int pop() throws PoppedWhenEmpty { if (topOfStack == 0)

throw new PoppedWhenEmpty();

int element = elements.get(--topOfStack); elements.remove(topOfStack);

return element; }

}

The strategy of keeping functions small and keeping parameter lists short can some-times lead to a proliferation of instance variables that are used by a subset of methods. When this happens, it almost always means that there is at least one other class trying to  
Chiến lược giữ cho các hàm nhỏ và giữ cho danh sách tham số ngắn đôi khi có thể dẫn đến sự gia tăng các biến thể hiện được sử dụng bởi một tập hợp con các phương thức. Khi điều này xảy ra, điều đó hầu như luôn có nghĩa là có ít nhất một lớp khác đang cố gắng

<https://www.facebook.com/groups/phattrienlaptrinh>

**Classes Should Be Small!** 141  
Các lớp học nên nhỏ! 141

get out of the larger class. You should try to separate the variables and methods into two or more classes such that the new classes are more cohesive.  
ra khỏi lớp lớn hơn. Bạn nên cố gắng tách các biến và phương thức thành hai hoặc nhiều lớp sao cho các lớp mới gắn kết hơn.

[**Maintaining Cohesion Results in Many Small Classes**](#_page_383_0)  
Duy trì kết quả gắn kết trong nhiều lớp nhỏ

Just the act of breaking large functions into smaller functions causes a proliferation of classes. Consider a large function with many variables declared within it. Let’s say you want to extract one small part of that function into a separate function. However, the code you want to extract uses four of the variables declared in the function. Must you pass all four of those variables into the new function as arguments?  
Chỉ hành động chia các chức năng lớn thành các chức năng nhỏ hơn cũng gây ra sự gia tăng các lớp. Hãy xem xét một hàm lớn với nhiều biến được khai báo bên trong nó. Giả sử bạn muốn trích xuất một phần nhỏ của chức năng đó thành một chức năng riêng biệt. Tuy nhiên, mã bạn muốn trích xuất sử dụng bốn biến được khai báo trong hàm. Bạn có phải chuyển tất cả bốn biến đó vào hàm mới dưới dạng đối số không?

Not at all! If we promoted those four variables to instance variables of the class, then we could extract the code without passing *any* variables at all. It would be *easy* to break the function up into small pieces.  
Không có gì! Nếu chúng ta thăng cấp bốn biến đó thành các biến thể hiện của lớp, thì chúng ta có thể trích xuất mã mà không cần chuyển bất kỳ biến nào. Sẽ dễ dàng chia chức năng thành các phần nhỏ.

Unfortunately, this also means that our classes lose cohesion because they accumulate more and more instance variables that exist solely to allow a few functions to share them. But wait! If there are a few functions that want to share certain variables, doesn’t that make them a class in their own right? Of course it does. When classes lose cohesion, split them!  
Thật không may, điều này cũng có nghĩa là các lớp của chúng ta mất đi sự gắn kết bởi vì chúng tích lũy ngày càng nhiều biến đối tượng tồn tại chỉ để cho phép một vài hàm chia sẻ chúng. Nhưng đợi đã! Nếu có một vài hàm muốn chia sẻ một số biến nhất định, thì điều đó không biến chúng thành một lớp theo đúng nghĩa của chúng sao? Tất nhiên là thế. Khi các lớp học mất đi sự gắn kết, hãy tách chúng ra!

So breaking a large function into many smaller functions often gives us the opportu-nity to split several smaller classes out as well. This gives our program a much better orga-nization and a more transparent structure.  
Vì vậy, việc chia nhỏ một hàm lớn thành nhiều hàm nhỏ hơn thường mang lại cho chúng ta cơ hội để chia nhỏ một số lớp nhỏ hơn. Điều này mang lại cho chương trình của chúng tôi một tổ chức tốt hơn nhiều và một cấu trúc minh bạch hơn.

As a demonstration of what I mean, let’s use a time-honored example taken from Knuth’s wonderful book *Literate Programming.*3 Listing 10-5 shows a translation into Java of Knuth’s PrintPrimes program. To be fair to Knuth, this is not the program as he wrote it but rather as it was output by his WEB tool. I’m using it because it makes a great starting place for breaking up a big function into many smaller functions and classes.  
Để minh họa cho điều tôi muốn nói, chúng ta hãy sử dụng một ví dụ lâu đời lấy từ cuốn sách tuyệt vời của Knuth Literate Programming.3 Liệt kê 10-5 cho thấy một bản dịch sang Java của chương trình PrintPrimes của Knuth. Công bằng mà nói với Knuth, đây không phải là chương trình như anh ấy đã viết mà đúng hơn là nó được xuất ra bởi công cụ WEB của anh ấy. Tôi đang sử dụng nó vì nó là nơi khởi đầu tuyệt vời để chia nhỏ một hàm lớn thành nhiều hàm và lớp nhỏ hơn.

**Listing 10-5 PrintPrimes.java** package literatePrimes;  
Liệt kê 10-5 Gói PrintPrimes.java LiteratePrimes;

public class PrintPrimes {

public static void main(String[] args) { final int M = 1000;

final int RR = 50; final int CC = 4; final int WW = 10; final int ORDMAX = 30;

int P[] = new int[M + 1]; int PAGENUMBER;

int PAGEOFFSET; int ROWOFFSET; int C;

3. [Knuth92].

<https://www.facebook.com/groups/phattrienlaptrinh>

142 **Chapter 10: Classes**  
142 Chương 10: Lớp học

**Listing 10-5 (continued) PrintPrimes.java  
Liệt kê 10-5 (tiếp theo) PrintPrimes.java**

int J; int K;

boolean JPRIME; int ORD;

int SQUARE; int N;

int MULT[] = new int[ORDMAX + 1];

J = 1; K = 1;

P[1] = 2; ORD = 2; SQUARE = 9;

while (K < M) { do {

J = J + 2;

if (J == SQUARE) { ORD = ORD + 1;

SQUARE = P[ORD] \* P[ORD]; MULT[ORD - 1] = J;

}

N = 2;

JPRIME = true;

while (N < ORD && JPRIME) { while (MULT[N] < J)

MULT[N] = MULT[N] + P[N] + P[N]; if (MULT[N] == J)

JPRIME = false; N = N + 1;

}

} while (!JPRIME); K = K + 1;

P[K] = J; }

{

PAGENUMBER = 1; PAGEOFFSET = 1;

while (PAGEOFFSET <= M) { System.out.println("The First " + M +

" Prime Numbers --- Page " + PAGENUMBER); System.out.println("");

for (ROWOFFSET = PAGEOFFSET; ROWOFFSET < PAGEOFFSET + RR; ROWOFFSET++){ for (C = 0; C < CC;C++)

if (ROWOFFSET + C \* RR <= M) System.out.format("%10d", P[ROWOFFSET + C \* RR]);

System.out.println(""); } System.out.println("\f");

PAGENUMBER = PAGENUMBER + 1; PAGEOFFSET = PAGEOFFSET + RR \* CC;

} }

} }

<https://www.facebook.com/groups/phattrienlaptrinh>

**Classes Should Be Small!** 143  
Các lớp học nên nhỏ! 143

This program, written as a single function, is a mess. It has a deeply indented struc-ture, a plethora of odd variables, and a tightly coupled structure. At the very least, the one big function should be split up into a few smaller functions.  
Chương trình này, được viết dưới dạng một hàm duy nhất, là một mớ hỗn độn. Nó có một cấu trúc thụt vào sâu, rất nhiều biến lẻ và một cấu trúc liên kết chặt chẽ. Ít nhất, một chức năng lớn nên được chia thành một vài chức năng nhỏ hơn.

Listing 10-6 through Listing 10-8 show the result of splitting the code in Listing 10-5 into smaller classes and functions, and choosing meaningful names for those classes, func-tions, and variables.  
Liệt kê 10-6 đến Liệt kê 10-8 cho thấy kết quả của việc chia mã trong Liệt kê 10-5 thành các lớp và hàm nhỏ hơn, đồng thời chọn các tên có ý nghĩa cho các lớp, hàm và biến đó.

**Listing 10-6**

**PrimePrinter.java (refactored)** package literatePrimes;  
PrimePrinter.java (tái cấu trúc) gói LiteratePrimes;

public class PrimePrinter {

public static void main(String[] args) { final int NUMBER\_OF\_PRIMES = 1000;

int[] primes = PrimeGenerator.generate(NUMBER\_OF\_PRIMES);

final int ROWS\_PER\_PAGE = 50; final int COLUMNS\_PER\_PAGE = 4; RowColumnPagePrinter tablePrinter =

new RowColumnPagePrinter(ROWS\_PER\_PAGE, COLUMNS\_PER\_PAGE,

"The First " + NUMBER\_OF\_PRIMES + " Prime Numbers");

tablePrinter.print(primes); }

}

**Listing 10-7 RowColumnPagePrinter.java** package literatePrimes;  
Liệt kê 10-7 RowColumnPagePrinter.java gói LiteratePrimes;

import java.io.PrintStream;

public class RowColumnPagePrinter { private int rowsPerPage;

private int columnsPerPage; private int numbersPerPage; private String pageHeader; private PrintStream printStream;

public RowColumnPagePrinter(int rowsPerPage, int columnsPerPage, String pageHeader) {

this.rowsPerPage = rowsPerPage; this.columnsPerPage = columnsPerPage; this.pageHeader = pageHeader;

numbersPerPage = rowsPerPage \* columnsPerPage; printStream = System.out;

}

<https://www.facebook.com/groups/phattrienlaptrinh>

144 **Chapter 10: Classes**  
144 Chương 10: Lớp học

**Listing 10-7 (continued) RowColumnPagePrinter.java**  
Liệt kê 10-7 (tiếp theo) RowColumnPagePrinter.java

public void print(int data[]) { int pageNumber = 1;

for (int firstIndexOnPage = 0; firstIndexOnPage < data.length; firstIndexOnPage += numbersPerPage) {

int lastIndexOnPage =

Math.min(firstIndexOnPage + numbersPerPage - 1, data.length - 1);

printPageHeader(pageHeader, pageNumber); printPage(firstIndexOnPage, lastIndexOnPage, data); printStream.println("\f");

pageNumber++; }

}

private void printPage(int firstIndexOnPage, int lastIndexOnPage, int[] data) {

int firstIndexOfLastRowOnPage = firstIndexOnPage + rowsPerPage - 1;

for (int firstIndexInRow = firstIndexOnPage; firstIndexInRow <= firstIndexOfLastRowOnPage; firstIndexInRow++) {

printRow(firstIndexInRow, lastIndexOnPage, data); printStream.println("");

} }

private void printRow(int firstIndexInRow, int lastIndexOnPage, int[] data) {

for (int column = 0; column < columnsPerPage; column++) { int index = firstIndexInRow + column \* rowsPerPage;

if (index <= lastIndexOnPage) printStream.format("%10d", data[index]);

} }

private void printPageHeader(String pageHeader, int pageNumber) {

printStream.println(pageHeader + " --- Page " + pageNumber); printStream.println("");

}

public void setOutput(PrintStream printStream) { this.printStream = printStream;

} }

<https://www.facebook.com/groups/phattrienlaptrinh>

**Classes Should Be Small!** 145  
Các lớp học nên nhỏ! 145

**Listing 10-8 PrimeGenerator.java** package literatePrimes;  
Liệt kê 10-8 PrimeGenerator.java gói LiteratePrimes;

import java.util.ArrayList;

public class PrimeGenerator { private static int[] primes;

private static ArrayList<Integer> multiplesOfPrimeFactors;

protected static int[] generate(int n) { primes = new int[n];

multiplesOfPrimeFactors = new ArrayList<Integer>(); set2AsFirstPrime(); checkOddNumbersForSubsequentPrimes();

return primes; }

private static void set2AsFirstPrime() { primes[0] = 2; multiplesOfPrimeFactors.add(2);

}

private static void checkOddNumbersForSubsequentPrimes() { int primeIndex = 1;

for (int candidate = 3; primeIndex < primes.length; candidate += 2) {

if (isPrime(candidate)) primes[primeIndex++] = candidate;

} }

private static boolean isPrime(int candidate) {

if (isLeastRelevantMultipleOfNextLargerPrimeFactor(candidate)) { multiplesOfPrimeFactors.add(candidate);

return false; }

return isNotMultipleOfAnyPreviousPrimeFactor(candidate); }

private static boolean isLeastRelevantMultipleOfNextLargerPrimeFactor(int candidate) {

int nextLargerPrimeFactor = primes[multiplesOfPrimeFactors.size()];

int leastRelevantMultiple = nextLargerPrimeFactor \* nextLargerPrimeFactor; return candidate == leastRelevantMultiple;

}

private static boolean isNotMultipleOfAnyPreviousPrimeFactor(int candidate) {

for (int n = 1; n < multiplesOfPrimeFactors.size(); n++) {

if (isMultipleOfNthPrimeFactor(candidate, n)) return false;

}

<https://www.facebook.com/groups/phattrienlaptrinh>

146 **Chapter 10: Classes**  
146 Chương 10: Lớp học

**Listing 10-8 (continued) PrimeGenerator.java**  
Liệt kê 10-8 (tiếp theo) PrimeGenerator.java

return true; }

private static boolean isMultipleOfNthPrimeFactor(int candidate, int n) {

return

candidate == smallestOddNthMultipleNotLessThanCandidate(candidate, n); }

private static int

smallestOddNthMultipleNotLessThanCandidate(int candidate, int n) { int multiple = multiplesOfPrimeFactors.get(n);

while (multiple < candidate) multiple += 2 \* primes[n];

multiplesOfPrimeFactors.set(n, multiple); return multiple;

} }

The ﬁrst thing you might notice is that the program got a lot longer. It went from a little over one page to nearly three pages in length. There are several reasons for this growth. First, the refactored program uses longer, more descriptive variable names. Second, the refactored program uses function and class declarations as a way to add commentary to the code. Third, we used whitespace and formatting techniques to keep the program readable.  
Điều đầu tiên bạn có thể nhận thấy là chương trình dài hơn rất nhiều. Nó dài từ hơn một trang một chút đến gần ba trang. Có một số lý do cho sự tăng trưởng này. Đầu tiên, chương trình được tái cấu trúc sử dụng các tên biến dài hơn, mang tính mô tả hơn. Thứ hai, chương trình tái cấu trúc sử dụng khai báo hàm và lớp như một cách để thêm chú thích vào mã. Thứ ba, chúng tôi đã sử dụng các kỹ thuật định dạng và khoảng trắng để giữ cho chương trình có thể đọc được.

Notice how the program has been split into three main responsibilities. The main program is contained in the PrimePrinterclass all by itself. Its responsibility is to handle the execution environment. It will change if the method of invocation changes. For example, if this program were converted to a SOAP service, this is the class that would be affected.  
Lưu ý cách chương trình đã được chia thành ba trách nhiệm chính. Chương trình chính được chứa trong lớp PrimePrinter hoàn toàn. Trách nhiệm của nó là xử lý môi trường thực thi. Nó sẽ thay đổi nếu phương thức gọi thay đổi. Ví dụ: nếu chương trình này được chuyển đổi sang dịch vụ SOAP, thì đây là lớp sẽ bị ảnh hưởng.

The RowColumnPagePrinterknows all about how to format a list of numbers into pages with a certain number of rows and columns. If the formatting of the output needed changing, then this is the class that would be affected.  
RowColumnPagePrinter biết tất cả về cách định dạng danh sách số thành các trang có số hàng và cột nhất định. Nếu định dạng của đầu ra cần thay đổi, thì đây là lớp sẽ bị ảnh hưởng.

The PrimeGeneratorclass knows how to generate a list prime numbers. Notice that it is not meant to be instantiated as an object. The class is just a useful scope in which its variables can be declared and kept hidden. This class will change if the algorithm for computing prime numbers changes.  
Lớp PrimeGenerator biết cách tạo danh sách các số nguyên tố. Lưu ý rằng nó không có nghĩa là được khởi tạo như một đối tượng. Lớp chỉ là một phạm vi hữu ích trong đó các biến của nó có thể được khai báo và ẩn đi. Lớp này sẽ thay đổi nếu thuật toán tính số nguyên tố thay đổi.

This was not a rewrite! We did not start over from scratch and write the program over again. Indeed, if you look closely at the two different programs, you’ll see that they use the same algorithm and mechanics to get their work done.  
Đây không phải là viết lại! Chúng tôi đã không bắt đầu lại từ đầu và viết lại chương trình. Thật vậy, nếu bạn xem xét kỹ hai chương trình khác nhau, bạn sẽ thấy rằng chúng sử dụng cùng một thuật toán và cơ chế để hoàn thành công việc của mình.

The change was made by writing a test suite that veriﬁed the *precise* behavior of the ﬁrst program. Then a myriad of tiny little changes were made, one at a time. After each change the program was executed to ensure that the behavior had not changed. One tiny step after another, the ﬁrst program was cleaned up and transformed into the second.  
Sự thay đổi được thực hiện bằng cách viết một bộ thử nghiệm xác minh hành vi chính xác của chương trình đầu tiên. Sau đó, vô số thay đổi nhỏ được thực hiện, từng cái một. Sau mỗi thay đổi, chương trình được thực thi để đảm bảo rằng hành vi không thay đổi. Từng bước nhỏ một, chương trình đầu tiên được làm sạch và chuyển sang chương trình thứ hai.

<https://www.facebook.com/groups/phattrienlaptrinh>

**Organizing for Change** 147  
Tổ chức để Thay đổi 147

[**Organizing for Change**](#_page_383_0)  
Tổ chức để thay đổi

For most systems, change is continual. Every change subjects us to the risk that the remainder of the system no longer works as intended. In a clean system we organize our classes so as to reduce the risk of change.  
Đối với hầu hết các hệ thống, thay đổi là liên tục. Mọi thay đổi đều khiến chúng tôi gặp rủi ro là phần còn lại của hệ thống không còn hoạt động như dự kiến. Trong một hệ thống sạch sẽ, chúng tôi tổ chức các lớp học của mình để giảm rủi ro thay đổi.

The Sqlclass in Listing 10-9 is used to generate properly formed SQL strings given appropriate metadata. It’s a work in progress and, as such, doesn’t yet support SQL func-tionality like updatestatements. When the time comes for the Sqlclass to support an updatestatement, we’ll have to “open up” this class to make modiﬁcations. The problem with opening a class is that it introduces risk. Any modiﬁcations to the class have the potential of breaking other code in the class. It must be fully retested.  
Sqlclass trong Liệt kê 10-9 được sử dụng để tạo các chuỗi SQL được định dạng đúng cho siêu dữ liệu thích hợp. Đó là một công việc đang được tiến hành và do đó, chưa hỗ trợ chức năng SQL như các câu lệnh cập nhật. Khi đến lúc Sqlclass hỗ trợ một câu lệnh cập nhật, chúng ta sẽ phải “mở” lớp này để thực hiện các sửa đổi. Vấn đề với việc mở một lớp học là nó gây ra rủi ro. Bất kỳ sửa đổi nào đối với lớp đều có khả năng phá vỡ mã khác trong lớp. Nó phải được kiểm tra lại đầy đủ.

**Listing 10-9  
Liệt kê 10-9**

**A class that must be opened for change  
Một lớp học phải được mở ra để thay đổi**

public class Sql {

public Sql(String table, Column[] columns) public String create()

public String insert(Object[] fields) public String selectAll()

public String findByKey(String keyColumn, String keyValue) public String select(Column column, String pattern) public String select(Criteria criteria)

public String preparedInsert()

private String columnList(Column[] columns)

private String valuesList(Object[] fields, final Column[] columns) private String selectWithCriteria(String criteria)

private String placeholderList(Column[] columns) }

The Sqlclass must change when we add a new type of statement. It also must change when we alter the details of a single statement type—for example, if we need to modify the selectfunctionality to support subselects. These two reasons to change mean that the Sql class violates the SRP.  
Sqlclass phải thay đổi khi chúng ta thêm một loại câu lệnh mới. Nó cũng phải thay đổi khi chúng ta thay đổi các chi tiết của một loại câu lệnh—ví dụ: nếu chúng ta cần sửa đổi chức năng chọn để hỗ trợ các lựa chọn phụ. Hai lý do thay đổi này có nghĩa là lớp Sql vi phạm SRP.

We can spot this SRP violation from a simple organizational standpoint. The method outline of Sql shows that there are private methods, such as selectWithCriteria, that appear to relate only to select statements.  
Chúng tôi có thể phát hiện vi phạm SRP này từ quan điểm tổ chức đơn giản. Phác thảo phương thức của Sql cho thấy rằng có các phương thức riêng tư, chẳng hạn như selectWithCriteria, dường như chỉ liên quan đến các câu lệnh được chọn.

Private method behavior that applies only to a small subset of a class can be a useful heuristic for spotting potential areas for improvement. However, the primary spur for tak-ing action should be system change itself. If the Sqlclass is deemed logically complete, then we need not worry about separating the responsibilities. If we won’t need update functionality for the foreseeable future, then we should leave Sqlalone. But as soon as we ﬁnd ourselves opening up a class, we should consider ﬁxing our design.  
Hành vi phương thức riêng chỉ áp dụng cho một tập hợp con nhỏ của một lớp có thể là một kinh nghiệm hữu ích để phát hiện các khu vực tiềm năng cần cải thiện. Tuy nhiên, động lực chính cho hành động thực hiện nên là chính sự thay đổi hệ thống. Nếu Sqlclass được coi là hoàn chỉnh về mặt logic, thì chúng ta không cần lo lắng về việc phân chia trách nhiệm. Nếu chúng tôi không cần chức năng cập nhật trong tương lai gần, thì chúng tôi nên rời khỏi Sqlalone. Nhưng ngay khi thấy mình mở một lớp học, chúng ta nên xem xét sửa đổi thiết kế của mình.

What if we considered a solution like that in Listing 10-10? Each public interface method deﬁned in the previous Sqlfrom Listing 10-9 is refactored out to its own derivative of the Sqlclass. Note that the private methods, such as valuesList, move directly where  
Điều gì sẽ xảy ra nếu chúng ta xem xét một giải pháp như thế trong Liệt kê 10-10? Mỗi phương thức giao diện công khai được định nghĩa trong Sql trước từ Liệt kê 10-9 được cấu trúc lại thành dẫn xuất riêng của nó từ Sqlclass. Lưu ý rằng các phương thức riêng tư, chẳng hạn như valueList, di chuyển trực tiếp đến nơi

<https://www.facebook.com/groups/phattrienlaptrinh>

148 **Chapter 10: Classes**  
148 Chương 10: Lớp học

they are needed. The common private behavior is isolated to a pair of utility classes, Where and ColumnList.  
chúng là cần thiết. Hành vi riêng tư phổ biến được tách biệt với một cặp lớp tiện ích, Where và ColumnList.

**Listing 10-10**

**A set of closed classes  
Tập hợp các lớp đóng**

abstract public class Sql {

public Sql(String table, Column[] columns) abstract public String generate();

}

public class CreateSql extends Sql {

public CreateSql(String table, Column[] columns) @Override public String generate()

}

public class SelectSql extends Sql {

public SelectSql(String table, Column[] columns) @Override public String generate()

}

public class InsertSql extends Sql {

public InsertSql(String table, Column[] columns, Object[] fields) @Override public String generate()

private String valuesList(Object[] fields, final Column[] columns) }

public class SelectWithCriteriaSql extends Sql { public SelectWithCriteriaSql(

String table, Column[] columns, Criteria criteria) @Override public String generate()

}

public class SelectWithMatchSql extends Sql { public SelectWithMatchSql(

String table, Column[] columns, Column column, String pattern) @Override public String generate()

}

public class FindByKeySql extends Sql public FindByKeySql(

String table, Column[] columns, String keyColumn, String keyValue) @Override public String generate()

}

public class PreparedInsertSql extends Sql {

public PreparedInsertSql(String table, Column[] columns) @Override public String generate() {

private String placeholderList(Column[] columns) }

public class Where {

public Where(String criteria) public String generate()

}

<https://www.facebook.com/groups/phattrienlaptrinh>

**Organizing for Change** 149  
Tổ chức để Thay đổi 149

**Listing 10-10 (continued)**

**A set of closed classes  
Tập hợp các lớp đóng**

public class ColumnList {

public ColumnList(Column[] columns) public String generate()

}

The code in each class becomes excruciatingly simple. Our required comprehension time to understand any class decreases to almost nothing. The risk that one function could break another becomes vanishingly small. From a test standpoint, it becomes an easier task to prove all bits of logic in this solution, as the classes are all isolated from one another.  
Mã trong mỗi lớp trở nên cực kỳ đơn giản. Thời gian hiểu cần thiết của chúng tôi để hiểu bất kỳ lớp nào giảm xuống gần như bằng không. Nguy cơ một chức năng có thể phá vỡ một chức năng khác trở nên cực kỳ nhỏ. Từ quan điểm kiểm tra, việc chứng minh tất cả các bit logic trong giải pháp này sẽ trở thành một nhiệm vụ dễ dàng hơn, vì tất cả các lớp đều được cách ly với nhau.

Equally important, when it’s time to add the updatestatements, none of the existing classes need change! We code the logic to build updatestatements in a new subclass of Sql named UpdateSql. No other code in the system will break because of this change.  
Quan trọng không kém, khi đến lúc thêm các câu lệnh cập nhật, không có lớp nào hiện có cần thay đổi! Chúng tôi viết mã logic để xây dựng các câu lệnh cập nhật trong một lớp con mới của Sql có tên là UpdateSql. Không có mã nào khác trong hệ thống sẽ bị hỏng do thay đổi này.

Our restructured Sqllogic represents the best of all worlds. It supports the SRP. It also supports another key OO class design principle known as the Open-Closed Principle, or OCP:4 Classes should be open for extension but closed for modiﬁcation. Our restructured Sqlclass is open to allow new functionality via subclassing, but we can make this change while keeping every other class closed. We simply drop our UpdateSql class in place.  
Sqllogic được tái cấu trúc của chúng tôi đại diện cho thế giới tốt nhất. Nó hỗ trợ SRP. Nó cũng hỗ trợ một nguyên tắc thiết kế lớp OO quan trọng khác được gọi là Nguyên tắc Mở-Đóng, hoặc OCP:4 Các lớp nên mở để mở rộng nhưng đóng để sửa đổi. Sqlclass được cấu trúc lại của chúng tôi mở để cho phép chức năng mới thông qua phân lớp, nhưng chúng tôi có thể thực hiện thay đổi này trong khi vẫn đóng mọi lớp khác. Chúng tôi chỉ cần bỏ lớp UpdateSql tại chỗ.

We want to structure our systems so that we muck with as little as possible when we update them with new or changed features. In an ideal system, we incorporate new fea-tures by extending the system, not by making modiﬁcations to existing code.  
Chúng tôi muốn cấu trúc hệ thống của mình sao cho chúng tôi làm hỏng ít nhất có thể khi cập nhật chúng với các tính năng mới hoặc đã thay đổi. Trong một hệ thống lý tưởng, chúng tôi kết hợp các tính năng mới bằng cách mở rộng hệ thống, chứ không phải bằng cách sửa đổi mã hiện có.

[**Isolating from Change**](#_page_383_0)  
Cô lập từ Thay đổi

Needs will change, therefore code will change. We learned in OO 101 that there are con-crete classes, which contain implementation details (code), and abstract classes, which represent concepts only. A client class depending upon concrete details is at risk when those details change. We can introduce interfaces and abstract classes to help isolate the impact of those details.  
Nhu cầu sẽ thay đổi, do đó mã sẽ thay đổi. Chúng ta đã học trong OO 101 rằng có các lớp cụ thể, chứa các chi tiết triển khai (mã) và các lớp trừu tượng, chỉ biểu thị các khái niệm. Một lớp khách hàng phụ thuộc vào các chi tiết cụ thể sẽ gặp rủi ro khi các chi tiết đó thay đổi. Chúng ta có thể giới thiệu các giao diện và lớp trừu tượng để giúp cô lập tác động của những chi tiết đó.

Dependencies upon concrete details create challenges for testing our system. If we’re building a Portfolioclass and it depends upon an external TokyoStockExchangeAPI to derive the portfolio’s value, our test cases are impacted by the volatility of such a lookup. It’s hard to write a test when we get a different answer every ﬁve minutes!  
Sự phụ thuộc vào các chi tiết cụ thể tạo ra những thách thức cho việc thử nghiệm hệ thống của chúng tôi. Nếu chúng tôi đang xây dựng một Danh mục đầu tư và nó phụ thuộc vào một TokyoStockExchangeAPI bên ngoài để lấy giá trị của danh mục đầu tư, thì các trường hợp thử nghiệm của chúng tôi sẽ bị ảnh hưởng bởi tính không ổn định của việc tra cứu như vậy. Thật khó để viết một bài kiểm tra khi chúng ta nhận được một câu trả lời khác nhau cứ sau năm phút!

Instead of designing Portfolioso that it directly depends upon TokyoStockExchange, we create an interface, StockExchange, that declares a single method:  
Thay vì thiết kế Danh mục đầu tư phụ thuộc trực tiếp vào TokyoStockExchange, chúng tôi tạo một giao diện, StockExchange, khai báo một phương thức duy nhất:

public interface StockExchange { Money currentPrice(String symbol);

}

4. [PPP].

<https://www.facebook.com/groups/phattrienlaptrinh>

150 **Chapter 10: Classes**  
150 Chương 10: Lớp học

We design TokyoStockExchangeto implement this interface. We also make sure that the constructor of Portfolio takes a StockExchange reference as an argument:  
Chúng tôi thiết kế TokyoStockExchange để triển khai giao diện này. Chúng tôi cũng đảm bảo rằng hàm tạo của Danh mục đầu tư lấy tham chiếu StockExchange làm đối số:

public Portfolio {

private StockExchange exchange;

public Portfolio(StockExchange exchange) { this.exchange = exchange;

}

// ... }

Now our test can create a testable implementation of the StockExchangeinterface that emulates the TokyoStockExchange. This test implementation will ﬁx the current value for any symbol we use in testing. If our test demonstrates purchasing ﬁve shares of Microsoft for our portfolio, we code the test implementation to always return $100 per share of Microsoft. Our test implementation of the StockExchangeinterface reduces to a simple table lookup. We can then write a test that expects $500 for our overall portfolio value.

public class PortfolioTest {

private FixedStockExchangeStub exchange; private Portfolio portfolio;

@Before

protected void setUp() throws Exception { exchange = new FixedStockExchangeStub(); exchange.fix("MSFT", 100);

portfolio = new Portfolio(exchange); }

@Test

public void GivenFiveMSFTTotalShouldBe500() throws Exception { portfolio.add(5, "MSFT");

Assert.assertEquals(500, portfolio.value()); }

}

If a system is decoupled enough to be tested in this way, it will also be more ﬂexible and promote more reuse. The lack of coupling means that the elements of our system are better isolated from each other and from change. This isolation makes it easier to under-stand each element of the system.  
Nếu một hệ thống đủ tách biệt để có thể thử nghiệm theo cách này, thì nó cũng sẽ linh hoạt hơn và thúc đẩy việc tái sử dụng nhiều hơn. Việc thiếu khớp nối có nghĩa là các yếu tố trong hệ thống của chúng tôi được cách ly tốt hơn với nhau và khỏi sự thay đổi. Sự cô lập này giúp hiểu từng thành phần của hệ thống dễ dàng hơn.

By minimizing coupling in this way, our classes adhere to another class design princi-ple known as the Dependency Inversion Principle (DIP).5 In essence, the DIP says that our classes should depend upon abstractions, not on concrete details.  
Bằng cách giảm thiểu khớp nối theo cách này, các lớp của chúng tôi tuân thủ một nguyên tắc thiết kế lớp khác được gọi là Nguyên tắc đảo ngược phụ thuộc (DIP).5 Về bản chất, DIP nói rằng các lớp của chúng tôi nên phụ thuộc vào sự trừu tượng, không phải trên các chi tiết cụ thể.

Instead of being dependent upon the implementation details of the TokyoStock-Exchangeclass, our Portfolioclass is now dependent upon the StockExchangeinterface. The StockExchangeinterface represents the abstract concept of asking for the current price of a symbol. This abstraction isolates all of the speciﬁc details of obtaining such a price, including from where that price is obtained.  
Thay vì phụ thuộc vào các chi tiết triển khai của lớp TokyoStock-Exchange, lớp Danh mục đầu tư của chúng tôi hiện phụ thuộc vào giao diện StockExchange. Giao diện StockExchange đại diện cho khái niệm trừu tượng về yêu cầu giá hiện tại của một biểu tượng. Sự trừu tượng hóa này cô lập tất cả các chi tiết cụ thể để có được một mức giá như vậy, bao gồm cả việc mức giá đó được lấy từ đâu.

5. [PPP].

<https://www.facebook.com/groups/phattrienlaptrinh>

**Bibliography** 151  
Thư mục 151

[**Bibliography**](#_page_383_0)  
Thư mục

**[RDD]:** *Object Design: Roles, Responsibilities, and Collaborations*, Rebecca Wirfs-Brock et al., Addison-Wesley, 2002.

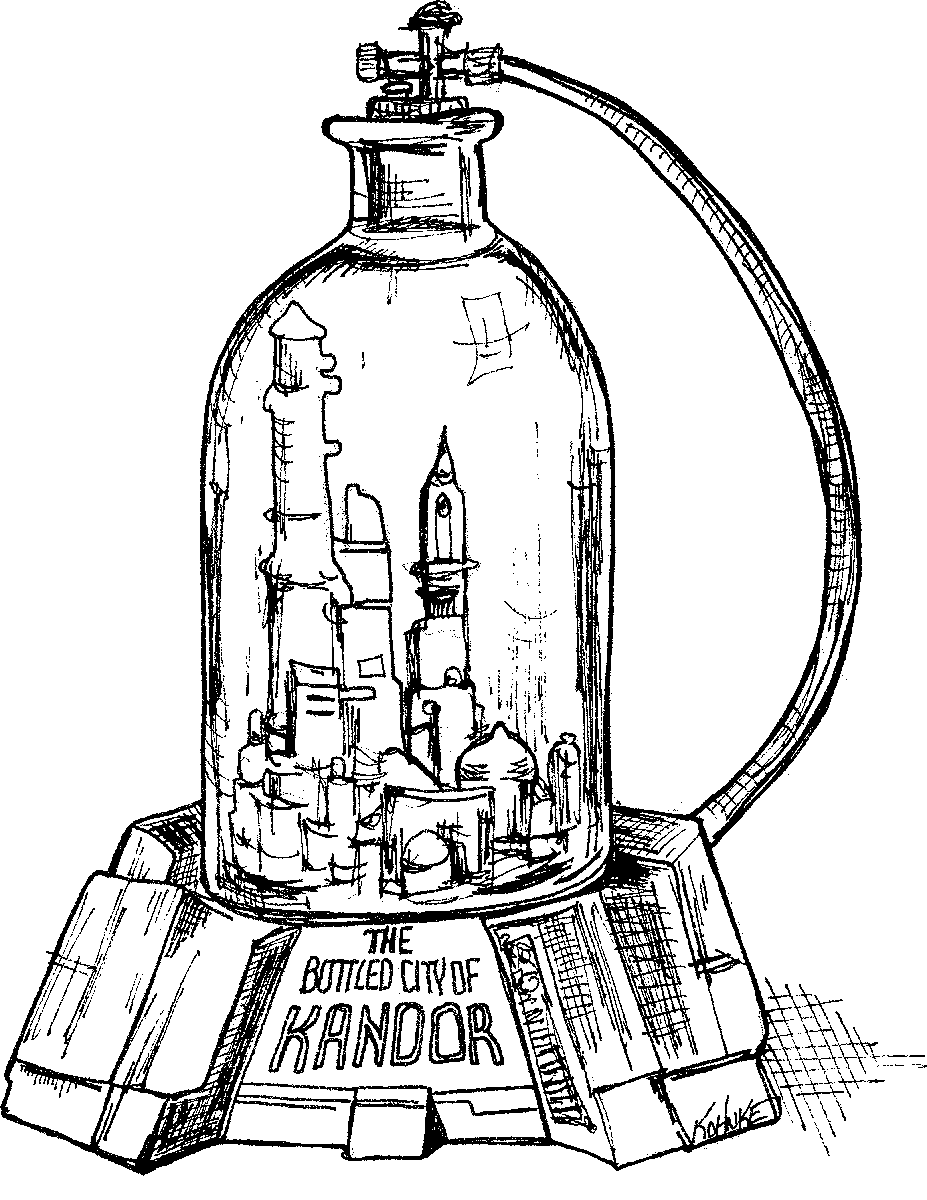
**[PPP]:** *Agile Software Development: Principles, Patterns, and Practices*, Robert C. Martin, Prentice Hall, 2002.

**[Knuth92]:** *Literate Programming,* Donald E. Knuth, Center for the Study of language and Information, Leland Stanford Junior University, 1992.

<https://www.facebook.com/groups/phattrienlaptrinh>

*This page intentionally left blank*

<https://www.facebook.com/groups/phattrienlaptrinh>



[**11**](#_page_383_0)

[**Systems**](#_page_383_0)  
hệ thống

by Dr. Kevin Dean Wampler

*“Complexity kills. It sucks the life out of developers, it makes products difﬁcult to plan, build, and test.”  
“Sự phức tạp giết chết. Nó hút hết sức sống của các nhà phát triển, nó làm cho các sản phẩm trở nên khó lập kế hoạch, xây dựng và thử nghiệm.”*

—Ray Ozzie, CTO, Microsoft Corporation

153

<https://www.facebook.com/groups/phattrienlaptrinh>

154 **Chapter 11: Systems**  
154 Chương 11: Hệ thống

[**How Would You Build a City?**](#_page_383_0)  
Làm thế nào bạn sẽ xây dựng một thành phố?

Could you manage all the details yourself? Probably not. Even managing an existing city is too much for one person. Yet, cities work (most of the time). They work because cities have teams of people who manage particular parts of the city, the water systems, power systems, trafﬁc, law enforcement, building codes, and so forth. Some of those people are responsible for the *big picture*, while others focus on the details.  
Bạn có thể tự quản lý tất cả các chi tiết không? Chắc là không. Ngay cả việc quản lý một thành phố hiện tại cũng là quá sức đối với một người. Tuy nhiên, các thành phố hoạt động (hầu hết thời gian). Chúng hoạt động vì các thành phố có các nhóm người quản lý các bộ phận cụ thể của thành phố, hệ thống nước, hệ thống điện, giao thông, thực thi pháp luật, quy tắc xây dựng, v.v. Một số người chịu trách nhiệm về bức tranh lớn, trong khi những người khác tập trung vào các chi tiết.

Cities also work because they have evolved appropriate levels of abstraction and mod-ularity that make it possible for individuals and the “components” they manage to work effectively, even without understanding the big picture.  
Các thành phố cũng hoạt động vì chúng đã phát triển các mức độ trừu tượng và mô đun thích hợp giúp cho các cá nhân và “các thành phần” mà họ quản lý có thể hoạt động hiệu quả, ngay cả khi không hiểu bức tranh toàn cảnh.

Although software teams are often organized like that too, the systems they work on often don’t have the same separation of concerns and levels of abstraction. Clean code helps us achieve this at the lower levels of abstraction. In this chapter let us consider how to stay clean at higher levels of abstraction, the *system* level.  
Mặc dù các nhóm phần mềm cũng thường được tổ chức như vậy, nhưng các hệ thống mà họ làm việc thường không có sự tách biệt về mối quan tâm và mức độ trừu tượng giống nhau. Mã sạch giúp chúng tôi đạt được điều này ở mức độ trừu tượng thấp hơn. Trong chương này, chúng ta hãy xem xét làm thế nào để giữ sạch ở cấp độ trừu tượng cao hơn, cấp độ hệ thống.

[**Separate Constructing a System from Using It**](#_page_383_0)  
Tách biệt việc xây dựng một hệ thống với việc sử dụng nó

First, consider that *construction* is a very different process from *use*. As I write this, there is a new hotel under construction that I see out my window in Chicago. Today it is a bare concrete box with a construction crane and elevator bolted to the outside. The busy people there all wear hard hats and work clothes. In a year or so the hotel will be ﬁnished. The crane and elevator will be gone. The building will be clean, encased in glass window walls and attractive paint. The people working and staying there will look a lot different too.  
Đầu tiên, hãy xem xét rằng xây dựng là một quá trình rất khác với sử dụng. Khi tôi viết bài này, có một khách sạn mới đang được xây dựng mà tôi nhìn thấy qua cửa sổ ở Chicago. Ngày nay nó là một hộp bê tông trần với cần cẩu xây dựng và thang máy được bắt vít ra bên ngoài. Những người bận rộn ở đó đều đội mũ cứng và mặc quần áo lao động. Trong khoảng một năm nữa khách sạn sẽ hoàn thành. Cần cẩu và thang máy sẽ biến mất. Tòa nhà sẽ sạch sẽ, được bao bọc bởi những bức tường cửa sổ bằng kính và màu sơn bắt mắt. Những người làm việc và ở lại đó cũng sẽ khác đi rất nhiều.

*Software systems should separate the startup process, when the application objects are constructed and the dependencies are “wired” together, from the runtime logic that takes over after startup.*

The startup process is a *concern* that any application must address. It is the ﬁrst *con-cern* that we will examine in this chapter. The *separation of concerns* is one of the oldest and most important design techniques in our craft.  
Quá trình khởi động là mối quan tâm mà bất kỳ ứng dụng nào cũng phải giải quyết. Đó là mối quan tâm đầu tiên mà chúng ta sẽ xem xét trong chương này. Việc tách biệt các mối quan tâm là một trong những kỹ thuật thiết kế lâu đời nhất và quan trọng nhất trong nghề thủ công của chúng tôi.

Unfortunately, most applications don’t separate this concern. The code for the startup process is ad hoc and it is mixed in with the runtime logic. Here is a typical example:  
Thật không may, hầu hết các ứng dụng không tách rời mối quan tâm này. Mã cho quá trình khởi động là đặc biệt và nó được trộn lẫn với logic thời gian chạy. Đây là một ví dụ điển hình:

public Service getService() { if (service == null)

service = new MyServiceImpl(...); // Good enough default for most cases? return service;

}

This is the LAZY INITIALIZATION/EVALUATION idiom, and it has several merits. We don’t incur the overhead of construction unless we actually use the object, and our startup times can be faster as a result. We also ensure that null is never returned.  
Đây là thành ngữ KHỞI TẠO/ĐÁNH GIÁ LƯỜI, và nó có một số giá trị. Chúng tôi không phải chịu chi phí xây dựng trừ khi chúng tôi thực sự sử dụng đối tượng và kết quả là thời gian khởi động của chúng tôi có thể nhanh hơn. Chúng tôi cũng đảm bảo rằng null không bao giờ được trả lại.

<https://www.facebook.com/groups/phattrienlaptrinh>

**Separate Constructing a System from Using It** 155  
Tách việc xây dựng một hệ thống khỏi việc sử dụng nó 155

However, we now have a hard-coded dependency on MyServiceImpland everything its constructor requires (which I have elided). We can’t compile without resolving these dependencies, even if we never actually use an object of this type at runtime!  
Tuy nhiên, giờ đây chúng tôi có một sự phụ thuộc được mã hóa cứng vào MyServiceImpland mọi thứ mà hàm tạo của nó yêu cầu (mà tôi đã bỏ qua). Chúng tôi không thể biên dịch mà không giải quyết các phụ thuộc này, ngay cả khi chúng tôi không bao giờ thực sự sử dụng một đối tượng thuộc loại này trong thời gian chạy!

Testing can be a problem. If MyServiceImplis a heavyweight object, we will need to make sure that an appropriate TEST DOUBLE1 or MOCK OBJECT gets assigned to the ser-vice ﬁeld before this method is called during unit testing. Because we have construction logic mixed in with normal runtime processing, we should test all execution paths (for example, the nulltest and its block). Having both of these responsibilities means that the method is doing more than one thing, so we are breaking the *Single Responsibility Principle* in a small way.  
Thử nghiệm có thể là một vấn đề. Nếu MyServiceImplis là một đối tượng hạng nặng, chúng tôi sẽ cần đảm bảo rằng một ĐỐI TƯỢNG TEST DOUBLE1 hoặc MOCK thích hợp được gán cho trường dịch vụ trước khi phương thức này được gọi trong quá trình kiểm tra đơn vị. Bởi vì chúng tôi có logic xây dựng trộn lẫn với quá trình xử lý thời gian chạy bình thường, chúng tôi nên kiểm tra tất cả các đường dẫn thực thi (ví dụ: nulltest và khối của nó). Có cả hai trách nhiệm này có nghĩa là phương pháp này đang thực hiện nhiều hơn một việc, vì vậy chúng ta đang vi phạm một phần nhỏ Nguyên tắc Trách nhiệm Đơn lẻ.

Perhaps worst of all, we do not know whether MyServiceImplis the right object in all cases. I implied as much in the comment. Why does the class with this method have to know the global context? Can we *ever* really know the right object to use here? Is it even possible for one type to be right for all possible contexts?  
Có lẽ tệ nhất là chúng ta không biết liệu MyServiceImplis có phải là đối tượng phù hợp trong mọi trường hợp hay không. Tôi ngụ ý rất nhiều trong bình luận. Tại sao lớp với phương pháp này phải biết bối cảnh toàn cầu? Chúng ta có thể thực sự biết đối tượng phù hợp để sử dụng ở đây không? Liệu một loại có thể phù hợp với tất cả các ngữ cảnh có thể không?

One occurrence of LAZY-INITIALIZATION isn’t a serious problem, of course. However, there are normally many instances of little setup idioms like this in applications. Hence, the global setup *strategy* (if there is one) is *scattered* across the application, with little modularity and often signiﬁcant duplication.  
Tất nhiên, một lần xuất hiện LAZY-KHỞI TẠO không phải là một vấn đề nghiêm trọng. Tuy nhiên, thường có nhiều trường hợp thành ngữ thiết lập nhỏ như thế này trong các ứng dụng. Do đó, chiến lược thiết lập toàn cầu (nếu có) nằm rải rác trong ứng dụng, với ít tính mô-đun và thường trùng lặp đáng kể.

If we are *diligent* about building well-formed and robust systems, we should never let little, *convenient* idioms lead to modularity breakdown. The startup process of object con-struction and wiring is no exception. We should modularize this process separately from the normal runtime logic and we should make sure that we have a global, consistent strat-egy for resolving our major dependencies.  
Nếu chúng ta siêng năng xây dựng các hệ thống mạnh mẽ và được hình thành tốt, thì chúng ta không bao giờ nên để những thành ngữ nhỏ, tiện lợi dẫn đến sự cố mô đun. Quá trình khởi động xây dựng đối tượng và nối dây cũng không ngoại lệ. Chúng ta nên mô đun hóa quy trình này tách biệt với logic thời gian chạy thông thường và chúng ta nên đảm bảo rằng chúng ta có một chiến lược toàn cầu, nhất quán để giải quyết các phụ thuộc chính của mình.

[**Separation of Main**](#_page_383_0)  
Tách chính

One way to separate construction from use is simply to move all aspects of construction to main, or modules called by main, and to design the rest of the system assuming that all objects have been constructed and wired up appropriately. (See Figure 11-1.)  
Một cách để tách việc xây dựng khỏi việc sử dụng đơn giản là chuyển tất cả các khía cạnh của việc xây dựng sang chính hoặc các mô-đun được gọi bởi chính và thiết kế phần còn lại của hệ thống với giả định rằng tất cả các đối tượng đã được xây dựng và nối dây một cách thích hợp. (Xem Hình 11-1.)

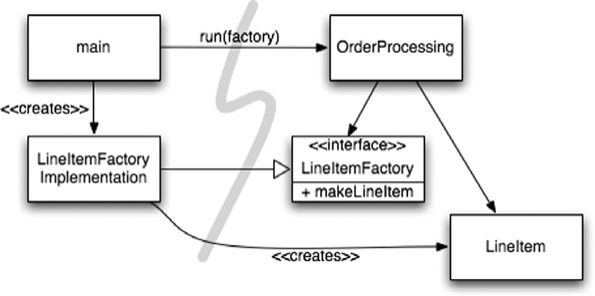
The ﬂow of control is easy to follow. The mainfunction builds the objects necessary for the system, then passes them to the application, which simply uses them. Notice the direction of the dependency arrows crossing the barrier between mainand the application. They all go one direction, pointing away from main. This means that the application has no knowledge of mainor of the construction process. It simply expects that everything has been built properly.  
Quy trình kiểm soát rất dễ theo dõi. Chức năng chính xây dựng các đối tượng cần thiết cho hệ thống, sau đó chuyển chúng đến ứng dụng, ứng dụng này chỉ sử dụng chúng. Lưu ý hướng mũi tên phụ thuộc đi qua rào cản giữa chính và ứng dụng. Tất cả đều đi về một hướng, chỉ ra khỏi chính. Điều này có nghĩa là ứng dụng không có kiến thức về mainor của quá trình xây dựng. Nó chỉ đơn giản mong đợi rằng mọi thứ đã được xây dựng đúng cách.

[**Factories**](#_page_383_0)  
nhà máy

Sometimes, of course, we need to make the application responsible for *when* an object gets created. For example, in an order processing system the application must create the  
Tất nhiên, đôi khi chúng ta cần làm cho ứng dụng chịu trách nhiệm khi một đối tượng được tạo. Ví dụ, trong một hệ thống xử lý đơn đặt hàng, ứng dụng phải tạo

1. [Mezzaros07].

<https://www.facebook.com/groups/phattrienlaptrinh>

156 **Chapter 11: Systems**  
156 Chương 11: Hệ thống

**Figure 11-1** Separating construction in main()  
Hình 11-1 Cấu trúc phân tách trong hàm main()

LineIteminstances to add to an Order. In this case we can use the ABSTRACT FACTORY2 pattern to give the application control of *when* to build the LineItems, but keep the details of that construction separate from the application code. (See Figure 11-2.)  
LineIteminstances để thêm vào Đơn hàng. Trong trường hợp này, chúng ta có thể sử dụng mẫu ABSTRACT FACTORY2 để cung cấp cho ứng dụng quyền kiểm soát thời điểm xây dựng Mục hàng, nhưng giữ các chi tiết của cấu trúc đó tách biệt với mã ứng dụng. (Xem Hình 11-2.)

**Figure 11-2  
Hình 11-2**

Separation construction with factory  
Công trình ngăn cách với nhà xưởng

Again notice that all the dependencies point from main toward the OrderProcessing application. This means that the application is decoupled from the details of how to build a LineItem. That capability is held in the LineItemFactoryImplementation, which is on the mainside of the line. And yet the application is in complete control of when the LineIteminstances get built and can even provide application-speciﬁc constructor arguments.  
Một lần nữa lưu ý rằng tất cả các điểm phụ thuộc từ chính hướng tới ứng dụng OrderProcessing. Điều này có nghĩa là ứng dụng được tách rời khỏi các chi tiết về cách xây dựng Mục hàng. Khả năng đó được giữ trong LineItemFactoryImplementation, nằm ở phần chính của dòng. Tuy nhiên, ứng dụng hoàn toàn kiểm soát thời điểm LineIteminstances được xây dựng và thậm chí có thể cung cấp các đối số hàm tạo dành riêng cho ứng dụng.

2. [GOF].

<https://www.facebook.com/groups/phattrienlaptrinh>

**Scaling Up** 157  
Mở rộng quy mô 157

[**Dependency Injection**](#_page_383_0)  
Tiêm phụ thuộc

A powerful mechanism for separating construction from use is *Dependency Injection* (DI), the application of *Inversion of Control* (IoC) to dependency management.3 Inversion of Control moves secondary responsibilities from an object to other objects that are dedicated to the purpose, thereby supporting the *Single Responsibility Principle.* In the context of dependency management, an object should not take responsibility for instantiating depen-dencies itself. Instead, it should pass this responsibility to another “authoritative” mecha-nism, thereby inverting the control. Because setup is a global concern, this authoritative mechanism will usually be either the “main” routine or a special-purpose *container.*  
Một cơ chế mạnh mẽ để tách việc xây dựng khỏi việc sử dụng là Dependency Injection (DI), ứng dụng của Inversion of Control (IoC) để quản lý sự phụ thuộc.3 Inversion of Control chuyển các trách nhiệm phụ từ một đối tượng sang các đối tượng khác dành riêng cho mục đích, do đó hỗ trợ nguyên tắc trách nhiệm duy nhất. Trong bối cảnh quản lý phụ thuộc, một đối tượng không nên chịu trách nhiệm khởi tạo chính các phụ thuộc. Thay vào đó, nó nên chuyển trách nhiệm này cho một cơ chế “có thẩm quyền” khác, do đó đảo ngược quyền kiểm soát. Bởi vì thiết lập là mối quan tâm toàn cầu, nên cơ chế có thẩm quyền này thường sẽ là quy trình “chính” hoặc bộ chứa có mục đích đặc biệt.

JNDI lookups are a “partial” implementation of DI, where an object asks a directory server to provide a “service” matching a particular name.  
Tra cứu JNDI là một triển khai DI "một phần", trong đó một đối tượng yêu cầu máy chủ thư mục cung cấp "dịch vụ" khớp với một tên cụ thể.

MyService myService = (MyService)(jndiContext.lookup(“NameOfMyService”));  
MyService myService = (MyService)(jndiContext.lookup(“NameOfMyService”));

The invoking object doesn’t control what kind of object is actually returned (as long it implements the appropriate interface, of course), but the invoking object still actively resolves the dependency.  
Đối tượng gọi không kiểm soát loại đối tượng nào thực sự được trả về (tất nhiên là miễn là nó triển khai giao diện phù hợp), nhưng đối tượng gọi vẫn tích cực giải quyết sự phụ thuộc.

True Dependency Injection goes one step further. The class takes no direct steps to resolve its dependencies; it is completely passive. Instead, it provides setter methods or constructor arguments (or both) that are used to *inject* the dependencies. During the con-struction process, the DI container instantiates the required objects (usually on demand) and uses the constructor arguments or setter methods provided to wire together the depen-dencies. Which dependent objects are actually used is speciﬁed through a conﬁguration ﬁle or programmatically in a special-purpose construction module.  
True Dependency Injection tiến thêm một bước. Lớp không thực hiện các bước trực tiếp để giải quyết các phụ thuộc của nó; nó hoàn toàn thụ động. Thay vào đó, nó cung cấp các phương thức setter hoặc đối số hàm tạo (hoặc cả hai) được sử dụng để đưa vào các phụ thuộc. Trong quá trình xây dựng, bộ chứa DI khởi tạo các đối tượng được yêu cầu (thường là theo yêu cầu) và sử dụng các đối số hàm tạo hoặc phương thức setter được cung cấp để kết nối các phần phụ thuộc lại với nhau. Những đối tượng phụ thuộc nào được sử dụng thực sự được chỉ định thông qua tệp cấu hình hoặc được lập trình trong mô-đun xây dựng có mục đích đặc biệt.

The Spring Framework provides the best known DI container for Java.4 You deﬁne which objects to wire together in an XML conﬁguration ﬁle, then you ask for particular objects by name in Java code. We will look at an example shortly.  
Spring Framework cung cấp bộ chứa DI nổi tiếng nhất cho Java.4 Bạn xác định các đối tượng nào sẽ kết nối với nhau trong một tệp cấu hình XML, sau đó bạn yêu cầu các đối tượng cụ thể theo tên trong mã Java. Chúng ta sẽ xem xét một ví dụ ngay sau đây.

But what about the virtues of LAZY-INITIALIZATION? This idiom is still sometimes useful with DI. First, most DI containers won’t construct an object until needed. Second, many of these containers provide mechanisms for invoking factories or for constructing proxies, which could be used for LAZY-EVALUATION and similar *optimizations.*5  
Nhưng còn ưu điểm của LƯỜI KHỞI TẠO thì sao? Thành ngữ này đôi khi vẫn hữu ích với DI. Đầu tiên, hầu hết các bộ chứa DI sẽ không xây dựng một đối tượng cho đến khi cần thiết. Thứ hai, nhiều vùng chứa này cung cấp các cơ chế để gọi các nhà máy hoặc để xây dựng các proxy, có thể được sử dụng để ĐÁNH GIÁ LƯỜI và các tối ưu hóa tương tự.5

[**Scaling Up**](#_page_383_0)  
Mở rộng quy mô

Cities grow from towns, which grow from settlements. At ﬁrst the roads are narrow and practically nonexistent, then they are paved, then widened over time. Small buildings and  
Các thành phố phát triển từ các thị trấn, phát triển từ các khu định cư. Lúc đầu, những con đường hẹp và hầu như không tồn tại, sau đó chúng được trải nhựa, rồi mở rộng theo thời gian. tòa nhà nhỏ và

3. See, for example, [Fowler].  
3. Ví dụ, hãy xem [Fowler].

4. See [Spring]. There is also a Spring.NET framework.  
4. Xem [Mùa xuân]. Ngoài ra còn có một khuôn khổ Spring.NET.

5. Don’t forget that lazy instantiation/evaluation is just an optimization and perhaps premature!  
5. Đừng quên rằng việc khởi tạo/đánh giá lười biếng chỉ là một sự tối ưu hóa và có lẽ là quá sớm!

<https://www.facebook.com/groups/phattrienlaptrinh>

158 **Chapter 11: Systems**  
158 Chương 11: Hệ thống

empty plots are ﬁlled with larger buildings, some of which will eventually be replaced with skyscrapers.  
những mảnh đất trống được lấp đầy bằng những tòa nhà lớn hơn, một số trong đó cuối cùng sẽ được thay thế bằng những tòa nhà chọc trời.

At ﬁrst there are no services like power, water, sewage, and the Internet (gasp!). These services are also added as the population and building densities increase.  
Lúc đầu, không có các dịch vụ như điện, nước, nước thải và Internet (thở hổn hển!). Các dịch vụ này cũng được thêm vào khi mật độ dân số và tòa nhà tăng lên.

This growth is not without pain. How many times have you driven, bumper to bumper through a road “improvement” project and asked yourself, “Why didn’t they build it wide enough the ﬁrst time!?”  
Sự tăng trưởng này không phải là không có đau đớn. Đã bao nhiêu lần bạn lái xe qua một dự án “cải tạo” đường và tự hỏi: “Tại sao họ không xây đủ rộng ngay từ lần đầu tiên!?”

But it couldn’t have happened any other way. Who can justify the expense of a six-lane highway through the middle of a small town that anticipates growth? Who would *want* such a road through their town?  
Nhưng nó không thể xảy ra theo bất kỳ cách nào khác. Ai có thể biện minh cho chi phí của một đường cao tốc sáu làn xe chạy qua giữa một thị trấn nhỏ dự đoán sự phát triển? Ai sẽ muốn một con đường như vậy xuyên qua thị trấn của họ?

It is a myth that we can get systems “right the ﬁrst time.” Instead, we should imple-ment only today’s *stories,* then refactor and expand the system to implement new stories tomorrow. This is the essence of iterative and incremental agility. Test-driven develop-ment, refactoring, and the clean code they produce make this work at the code level.  
Việc chúng ta có thể có hệ thống “đúng ngay từ lần đầu tiên” là một chuyện hoang đường. Thay vào đó, chúng ta chỉ nên triển khai các câu chuyện của ngày hôm nay, sau đó cấu trúc lại và mở rộng hệ thống để triển khai các câu chuyện mới vào ngày mai. Đây là bản chất của sự linh hoạt lặp đi lặp lại và gia tăng. Quá trình phát triển dựa trên thử nghiệm, tái cấu trúc và mã sạch mà họ tạo ra giúp điều này hoạt động ở cấp độ mã.

But what about at the system level? Doesn’t the system architecture require preplan-ning? Certainly, *it* can’t grow incrementally from simple to complex*,* can it?  
Nhưng còn ở cấp độ hệ thống thì sao? Không phải kiến trúc hệ thống yêu cầu lập kế hoạch trước sao? Chắc chắn, nó không thể tăng dần từ đơn giản đến phức tạp, phải không?

*Software systems are unique compared to physical systems. Their architectures can grow incrementally,* ***if*** *we maintain the proper separation of concerns.  
Hệ thống phần mềm là duy nhất so với các hệ thống vật lý. Kiến trúc của chúng có thể phát triển dần dần, nếu chúng ta duy trì sự phân tách hợp lý các mối quan tâm.*

The ephemeral nature of software systems makes this possible, as we will see. Let us ﬁrst consider a counterexample of an architecture that doesn’t separate concerns adequately.  
Bản chất phù du của các hệ thống phần mềm khiến điều này trở nên khả thi, như chúng ta sẽ thấy. Đầu tiên chúng ta hãy xem xét một phản ví dụ về một kiến trúc không phân tách các mối quan tâm một cách thỏa đáng.

The original EJB1 and EJB2 architectures did not separate concerns appropriately and thereby imposed unnecessary barriers to organic growth. Consider an *Entity Bean* for a persistent Bankclass. An entity bean is an in-memory representation of relational data, in other words, a table row.  
Các kiến trúc EJB1 và EJB2 ban đầu không phân tách các mối quan tâm một cách thích hợp và do đó đặt ra các rào cản không cần thiết đối với tăng trưởng hữu cơ. Hãy xem xét một Entity Bean cho một Bankclass liên tục. Một bean thực thể là một biểu diễn trong bộ nhớ của dữ liệu quan hệ, nói cách khác, một hàng của bảng.

First, you had to deﬁne a local (in process) or remote (separate JVM) interface, which clients would use. Listing 11-1 shows a possible local interface:  
Đầu tiên, bạn phải xác định giao diện cục bộ (đang xử lý) hoặc giao diện từ xa (JVM riêng biệt) mà máy khách sẽ sử dụng. Liệt kê 11-1 cho thấy một giao diện cục bộ khả thi:

**Listing 11-1**

**An EJB2 local interface for a Bank EJB  
Giao diện cục bộ EJB2 cho Ngân hàng EJB**

package com.example.banking; import java.util.Collections; import javax.ejb.\*;

public interface BankLocal extends java.ejb.EJBLocalObject { String getStreetAddr1() throws EJBException;

String getStreetAddr2() throws EJBException; String getCity() throws EJBException; String getState() throws EJBException; String getZipCode() throws EJBException;

void setStreetAddr1(String street1) throws EJBException; void setStreetAddr2(String street2) throws EJBException; void setCity(String city) throws EJBException;

void setState(String state) throws EJBException;

<https://www.facebook.com/groups/phattrienlaptrinh>

**Scaling Up** 159  
Mở rộng quy mô 159

**Listing 11-1 (continued)**

**An EJB2 local interface for a Bank EJB  
Giao diện cục bộ EJB2 cho Ngân hàng EJB**

void setZipCode(String zip) throws EJBException; Collection getAccounts() throws EJBException;

void setAccounts(Collection accounts) throws EJBException; void addAccount(AccountDTO accountDTO) throws EJBException;

}

I have shown several attributes for the Bank’s address and a collection of accounts that the bank owns, each of which would have its data handled by a separate AccountEJB. Listing 11-2 shows the corresponding implementation class for the Bank bean.  
Tôi đã chỉ ra một số thuộc tính cho địa chỉ của Ngân hàng và tập hợp các tài khoản mà ngân hàng sở hữu, mỗi tài khoản sẽ có dữ liệu được xử lý bởi một AccountEJB riêng biệt. Liệt kê 11-2 cho thấy lớp triển khai tương ứng cho Bank bean.

**Listing 11-2**

**The corresponding EJB2 Entity Bean Implementation  
Triển khai Bean thực thể EJB2 tương ứng**

package com.example.banking; import java.util.Collections; import javax.ejb.\*;

public abstract class Bank implements javax.ejb.EntityBean { // Business logic...

public abstract String getStreetAddr1(); public abstract String getStreetAddr2(); public abstract String getCity(); public abstract String getState(); public abstract String getZipCode();

public abstract void setStreetAddr1(String street1); public abstract void setStreetAddr2(String street2); public abstract void setCity(String city);

public abstract void setState(String state); public abstract void setZipCode(String zip); public abstract Collection getAccounts();

public abstract void setAccounts(Collection accounts); public void addAccount(AccountDTO accountDTO) {

InitialContext context = new InitialContext();

AccountHomeLocal accountHome = context.lookup("AccountHomeLocal"); AccountLocal account = accountHome.create(accountDTO);

Collection accounts = getAccounts(); accounts.add(account);

}

// EJB container logic

public abstract void setId(Integer id); public abstract Integer getId();

public Integer ejbCreate(Integer id) { ... } public void ejbPostCreate(Integer id) { ... }

// The rest had to be implemented but were usually empty: public void setEntityContext(EntityContext ctx) {} public void unsetEntityContext() {}

public void ejbActivate() {} public void ejbPassivate() {} public void ejbLoad() {} public void ejbStore() {} public void ejbRemove() {}

}

<https://www.facebook.com/groups/phattrienlaptrinh>

160 **Chapter 11: Systems**  
160 Chương 11: Hệ thống

I haven’t shown the corresponding *LocalHome* interface, essentially a factory used to create objects, nor any of the possible Bank ﬁnder (query) methods you might add.  
Tôi chưa hiển thị giao diện LocalHome tương ứng, về cơ bản là một nhà máy được sử dụng để tạo các đối tượng, cũng như bất kỳ phương thức (truy vấn) tìm kiếm Ngân hàng nào mà bạn có thể thêm vào.

Finally, you had to write one or more XML deployment descriptors that specify the object-relational mapping details to a persistence store, the desired transactional behavior, security constraints, and so on*.*  
Cuối cùng, bạn phải viết một hoặc nhiều bộ mô tả triển khai XML chỉ định các chi tiết ánh xạ quan hệ đối tượng cho một kho lưu trữ bền vững, hành vi giao dịch mong muốn, các ràng buộc bảo mật, v.v.

The business logic is tightly coupled to the EJB2 application “container.” You must subclass container types and you must provide many lifecycle methods that are required by the container.  
Logic nghiệp vụ được liên kết chặt chẽ với “bộ chứa” ứng dụng EJB2. Bạn phải phân lớp các loại vùng chứa và bạn phải cung cấp nhiều phương thức vòng đời mà vùng chứa yêu cầu.

Because of this coupling to the heavyweight container, isolated unit testing is difﬁcult. It is necessary to mock out the container, which is hard, or waste a lot of time deploying EJBs and tests to a real server. Reuse outside of the EJB2 architecture is effectively impos-sible, due to the tight coupling.  
Do khớp nối này với hộp chứa nặng nên việc kiểm tra đơn vị bị cô lập gặp khó khăn. Cần phải mô phỏng vùng chứa, điều này rất khó hoặc lãng phí nhiều thời gian để triển khai EJB và kiểm tra máy chủ thực. Việc sử dụng lại bên ngoài kiến trúc EJB2 là điều không thể thực hiện được do khớp nối chặt chẽ.

Finally, even object-oriented programming is undermined. One bean cannot inherit from another bean. Notice the logic for adding a new account. It is common in EJB2 beans to deﬁne “data transfer objects” (DTOs) that are essentially “structs” with no behavior. This usually leads to redundant types holding essentially the same data, and it requires boilerplate code to copy data from one object to another.  
Cuối cùng, ngay cả lập trình hướng đối tượng cũng bị phá hoại. Một bean không thể kế thừa từ một bean khác. Lưu ý logic để thêm một tài khoản mới. Thông thường trong các hạt EJB2 định nghĩa “đối tượng truyền dữ liệu” (DTO) về cơ bản là “cấu trúc” không có hành vi. Điều này thường dẫn đến các loại dư thừa về cơ bản chứa cùng một dữ liệu và nó yêu cầu mã soạn sẵn để sao chép dữ liệu từ đối tượng này sang đối tượng khác.

[**Cross-Cutting Concerns**](#_page_383_0)  
Mối quan tâm xuyên suốt

The EJB2 architecture comes close to true separation of concerns in some areas. For example, the desired transactional, security, and some of the persistence behaviors are declared in the deployment descriptors, independently of the source code.  
Kiến trúc EJB2 tiến gần đến sự tách biệt thực sự của các mối quan tâm trong một số lĩnh vực. Ví dụ: giao dịch mong muốn, bảo mật và một số hành vi liên tục được khai báo trong bộ mô tả triển khai, độc lập với mã nguồn.

Note that *concerns* like persistence tend to cut across the natural object boundaries of a domain. You want to persist all your objects using generally the same strategy, for exam-ple*,* using a particular DBMS6 versus ﬂat ﬁles, following certain naming conventions for tables and columns, using consistent transactional semantics, and so on*.*  
Lưu ý rằng các mối quan tâm như tính bền bỉ có xu hướng vượt qua ranh giới đối tượng tự nhiên của một miền. Bạn muốn duy trì tất cả các đối tượng của mình bằng cách sử dụng chung một chiến lược, ví dụ, sử dụng một DBMS6 cụ thể so với các tệp phẳng, tuân theo các quy ước đặt tên nhất định cho các bảng và cột, sử dụng ngữ nghĩa giao dịch nhất quán, v.v.

In principle, you can reason about your persistence strategy in a modular, encapsulated way.Yet, in practice, you have to spread essentially the same code that implements the persis-tence strategy across many objects. We use the term *cross-cutting concerns* for concerns like these. Again, the persistence framework might be modular and our domain logic, in isola-tion, might be modular. The problem is the ﬁne-grained *intersection* of these domains.  
Về nguyên tắc, bạn có thể lập luận về chiến lược kiên trì của mình theo cách mô-đun, được gói gọn. Tuy nhiên, trên thực tế, về cơ bản, bạn phải trải rộng cùng một mã thực hiện chiến lược kiên trì trên nhiều đối tượng. Chúng tôi sử dụng thuật ngữ mối quan tâm xuyên suốt cho những mối quan tâm như thế này. Một lần nữa, khung kiên trì có thể là mô-đun và logic miền của chúng tôi, trong sự cô lập, có thể là mô-đun. Vấn đề là giao điểm chi tiết của các lĩnh vực này.

In fact, the way the EJB architecture handled persistence, security, and transactions*,* “anticipated” *aspect-oriented programming* (AOP),7 which is a general-purpose approach to restoring modularity for cross-cutting concerns.  
Trên thực tế, cách kiến trúc EJB xử lý tính bền bỉ, bảo mật và giao dịch, lập trình hướng khía cạnh (AOP) đã “dự kiến”7, đây là cách tiếp cận có mục đích chung để khôi phục tính mô đun cho các mối quan tâm xuyên suốt.

In AOP, modular constructs called *aspects* specify which points in the system should have their behavior modiﬁed in some consistent way to support a particular concern. This speciﬁcation is done using a succinct declarative or programmatic mechanism.  
Trong AOP, các cấu trúc mô-đun được gọi là các khía cạnh chỉ định những điểm nào trong hệ thống sẽ được sửa đổi hành vi của chúng theo một cách nhất quán nào đó để hỗ trợ một mối quan tâm cụ thể. Thông số kỹ thuật này được thực hiện bằng cách sử dụng một cơ chế lập trình hoặc khai báo ngắn gọn.

6. Database management system.  
6. Hệ quản trị cơ sở dữ liệu.

7. See [AOSD] for general information on aspects and [AspectJ]] and [Colyer] for AspectJ-speciﬁc information.  
7. Xem [AOSD] để biết thông tin chung về các khía cạnh và [AspectJ]] và [Colyer] để biết thông tin cụ thể về AspectJ.

<https://www.facebook.com/groups/phattrienlaptrinh>

**Java Proxies** 161  
Proxy Java 161

Using persistence as an example, you would declare which objects and attributes (or *patterns* thereof) should be persisted and then delegate the persistence tasks to your persis-tence framework. The behavior modiﬁcations are made *noninvasively*8 to the target code by the AOP framework. Let us look at three aspects or aspect-like mechanisms in Java.  
Sử dụng tính bền vững làm ví dụ, bạn sẽ khai báo những đối tượng và thuộc tính nào (hoặc mẫu của chúng) sẽ được duy trì và sau đó ủy quyền các tác vụ cần bền bỉ cho khung kiên trì của bạn. Các sửa đổi hành vi được thực hiện không xâm lấn8 đối với mã mục tiêu bằng khung AOP. Chúng ta hãy xem xét ba khía cạnh hoặc các cơ chế giống như khía cạnh trong Java.

[**Java Proxies**](#_page_383_0)  
Proxy Java

Java proxies are suitable for simple situations, such as wrapping method calls in individual objects or classes. However, the dynamic proxies provided in the JDK only work with interfaces. To proxy classes, you have to use a byte-code manipulation library, such as CGLIB, ASM, or Javassist.9  
Các proxy Java phù hợp với các tình huống đơn giản, chẳng hạn như gói các cuộc gọi phương thức trong các đối tượng hoặc lớp riêng lẻ. Tuy nhiên, các proxy động được cung cấp trong JDK chỉ hoạt động với các giao diện. Đối với các lớp proxy, bạn phải sử dụng thư viện thao tác mã byte, chẳng hạn như CGLIB, ASM hoặc Javassist.9

Listing 11-3 shows the skeleton for a JDK proxy to provide persistence support for our Bankapplication, covering only the methods for getting and setting the list of accounts.  
Liệt kê 11-3 hiển thị bộ khung cho proxy JDK để cung cấp hỗ trợ bền vững cho ứng dụng Ngân hàng của chúng ta, chỉ bao gồm các phương pháp để nhận và thiết lập danh sách tài khoản.

**Listing 11-3  
Liệt kê 11-3**

**JDK Proxy Example  
Ví dụ về JDK Proxy**

// Bank.java (suppressing package names...) import java.utils.\*;

// The abstraction of a bank. public interface Bank {

Collection<Account> getAccounts();

void setAccounts(Collection<Account> accounts); }

// BankImpl.java import java.utils.\*;

// The “Plain Old Java Object” (POJO) implementing the abstraction. public class BankImpl implements Bank {

private List<Account> accounts;

public Collection<Account> getAccounts() { return accounts;

}

public void setAccounts(Collection<Account> accounts) { this.accounts = new ArrayList<Account>();

for (Account account: accounts) { this.accounts.add(account);

} }

}

// BankProxyHandler.java import java.lang.reflect.\*; import java.util.\*;

8. Meaning no manual editing of the target source code is required. 9. See [CGLIB], [ASM], and [Javassist].  
8. Có nghĩa là không cần chỉnh sửa thủ công mã nguồn mục tiêu. 9. Xem [CGLIB], [ASM] và [Javassist].

<https://www.facebook.com/groups/phattrienlaptrinh>

162 **Chapter 11: Systems**  
162 Chương 11: Hệ thống

**Listing 11-3 (continued) JDK Proxy Example  
Liệt kê 11-3 (tiếp theo) Ví dụ về JDK Proxy**

// “InvocationHandler” required by the proxy API.

public class BankProxyHandler implements InvocationHandler { private Bank bank;

public BankHandler (Bank bank) { this.bank = bank;

}

// Method defined in InvocationHandler

public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {

String methodName = method.getName(); if (methodName.equals("getAccounts")) {

bank.setAccounts(getAccountsFromDatabase()); return bank.getAccounts();

} else if (methodName.equals("setAccounts")) { bank.setAccounts((Collection<Account>) args[0]); setAccountsToDatabase(bank.getAccounts()); return null;

} else { ...

} }

// Lots of details here:

protected Collection<Account> getAccountsFromDatabase() { ... }

protected void setAccountsToDatabase(Collection<Account> accounts) { ... } }

// Somewhere else...

Bank bank = (Bank) Proxy.newProxyInstance( Bank.class.getClassLoader(),

new Class[] { Bank.class },

new BankProxyHandler(new BankImpl()));

We deﬁned an interface Bank, which will be *wrapped* by the proxy, and a *Plain-Old Java Object* (POJO), BankImpl, that implements the business logic. (We will revisit POJOs shortly.)  
Chúng tôi đã xác định một Ngân hàng giao diện, sẽ được bao bọc bởi proxy và Đối tượng Java thuần túy cũ (POJO), BankImpl, thực hiện logic nghiệp vụ. (Chúng tôi sẽ sớm xem lại POJO.)

The Proxy API requires an InvocationHandlerobject that it calls to implement any Bankmethod calls made to the proxy. Our BankProxyHandleruses the Java reﬂection API to map the generic method invocations to the corresponding methods in BankImpl, and so on.  
API Proxy yêu cầu một đối tượng InvocationHandler mà nó gọi để triển khai bất kỳ lệnh gọi Bankmethod nào được thực hiện tới proxy. BankProxyHandler của chúng tôi sử dụng API phản ánh Java để ánh xạ các lời gọi phương thức chung tới các phương thức tương ứng trong BankImpl, v.v.

There is a *lot* of code here and it is relatively complicated, even for this simple case.10 Using one of the byte-manipulation libraries is similarly challenging. This code “volume”  
Có rất nhiều mã ở đây và nó tương đối phức tạp, ngay cả đối với trường hợp đơn giản này.10 Việc sử dụng một trong các thư viện thao tác byte cũng là một thách thức tương tự. Mã này "khối lượng"

10. For more detailed examples of the Proxy API and examples of its use, see, for example, [Goetz].  
10. Để biết thêm các ví dụ chi tiết về Proxy API và các ví dụ về việc sử dụng nó, hãy xem [Goetz].

<https://www.facebook.com/groups/phattrienlaptrinh>

**Pure JavaAOP Frameworks** 163  
Khung JavaAOP thuần túy 163

and complexity are two of the drawbacks of proxies. They make it hard to create clean code! Also, proxies don’t provide a mechanism for specifying system-wide execution “points” of interest, which is needed for a true AOP solution.11  
và độ phức tạp là hai trong số những nhược điểm của proxy. Chúng gây khó khăn cho việc tạo mã sạch! Ngoài ra, proxy không cung cấp cơ chế chỉ định các “điểm” quan tâm thực thi trên toàn hệ thống, điều cần thiết cho một giải pháp AOP thực sự.11

[**Pure JavaAOP Frameworks**](#_page_383_0)  
Khung JavaAOP thuần túy

Fortunately, most of the proxy boilerplate can be handled automatically by tools. Proxies are used internally in several Java frameworks, for example, Spring AOP and JBoss AOP, to implement aspects in pure Java.12 In Spring, you write your business logic as *Plain-Old Java Objects*. POJOs are purely focused on their domain. They have no dependencies on enterprise frameworks (or any other domains). Hence, they are conceptually simpler and easier to test drive. The relative simplicity makes it easier to ensure that you are imple-menting the corresponding user stories correctly and to maintain and evolve the code for future stories.  
May mắn thay, hầu hết các bản tóm tắt proxy có thể được xử lý tự động bằng các công cụ. Proxy được sử dụng nội bộ trong một số khung công tác Java, chẳng hạn như Spring AOP và JBoss AOP, để triển khai các khía cạnh trong Java thuần túy.12 Trong Spring, bạn viết logic nghiệp vụ của mình dưới dạng Đối tượng Java thuần túy. POJO hoàn toàn tập trung vào miền của họ. Chúng không phụ thuộc vào khung doanh nghiệp (hoặc bất kỳ miền nào khác). Do đó, chúng đơn giản hơn về mặt khái niệm và dễ lái thử hơn. Tính đơn giản tương đối giúp dễ dàng đảm bảo rằng bạn đang triển khai chính xác các câu chuyện của người dùng tương ứng cũng như duy trì và phát triển mã cho các câu chuyện trong tương lai.

You incorporate the required application infrastructure, including cross-cutting con-cerns like persistence, transactions, security, caching, failover, and so on, using declara-tive conﬁguration ﬁles or APIs. In many cases, you are actually specifying Spring or JBoss library aspects, where the framework handles the mechanics of using Java proxies or byte-code libraries transparently to the user. These declarations drive the dependency injection (DI) container, which instantiates the major objects and wires them together on demand*.*  
Bạn kết hợp cơ sở hạ tầng ứng dụng cần thiết, bao gồm các mối quan tâm xuyên suốt như tính bền vững, giao dịch, bảo mật, bộ nhớ đệm, chuyển đổi dự phòng, v.v. bằng cách sử dụng các tệp hoặc API cấu hình khai báo. Trong nhiều trường hợp, bạn đang thực sự chỉ định các khía cạnh của thư viện Spring hoặc JBoss, trong đó khung xử lý các cơ chế sử dụng proxy Java hoặc thư viện mã byte một cách minh bạch cho người dùng. Các khai báo này điều khiển bộ chứa tiêm phụ thuộc (DI), giúp khởi tạo các đối tượng chính và nối chúng lại với nhau theo yêu cầu.

Listing 11-4 shows a typical fragment of a Spring V2.5 conﬁguration ﬁle, app.xml13:  
Liệt kê 11-4 cho thấy một đoạn điển hình của tệp cấu hình Spring V2.5, app.xml13:

**Listing 11-4  
Liệt kê 11-4**

**Spring 2.X configuration file  
Tệp cấu hình Spring 2.X**

<beans> ...

<bean id="appDataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close" p:driverClassName="com.mysql.jdbc.Driver" p:url="jdbc:mysql://localhost:3306/mydb" p:username="me"/>

<bean id="bankDataAccessObject" class="com.example.banking.persistence.BankDataAccessObject" p:dataSource-ref="appDataSource"/>

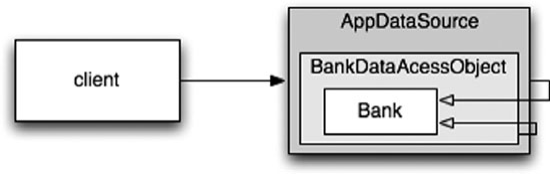
<bean id="bank"

11. AOP is sometimes confused with techniques used to implement it, such as method interception and “wrapping” through proxies. The real value of an AOP system is the ability to specify systemic behaviors in a concise and modular way.  
11. AOP đôi khi bị nhầm lẫn với các kỹ thuật được sử dụng để triển khai nó, chẳng hạn như chặn phương thức và “gói” thông qua proxy. Giá trị thực của hệ thống AOP là khả năng chỉ định các hành vi hệ thống theo cách ngắn gọn và mô đun.

12. See [Spring] and [JBoss]. “Pure Java” means without the use of AspectJ.  
12. Xem [Mùa xuân] và [JBoss]. “Java thuần túy” có nghĩa là không sử dụng AspectJ.

13. Adapted from <http://www.theserverside.com/tt/articles/article.tss?l=IntrotoSpring25>  
13. Chuyển thể từ http://www.theserverside.com/tt/articles/article.tss?l=IntrotoSpring25

<https://www.facebook.com/groups/phattrienlaptrinh>

164 **Chapter 11: Systems**  
164 Chương 11: Hệ thống

**Listing 11-4 (continued)**  
Liệt kê 11-4 (tiếp theo)

**Spring 2.X configuration file  
Tệp cấu hình Spring 2.X**

class="com.example.banking.model.Bank" p:dataAccessObject-ref="bankDataAccessObject"/> ...

</beans>

Each “bean” is like one part of a nested “Russian doll,” with a domain object for a Bankproxied (wrapped) by a data accessor object (DAO), which is itself proxied by a JDBC driver data source. (See Figure 11-3.)  
Mỗi “đậu” giống như một phần của “búp bê Nga” lồng nhau, với một đối tượng miền dành cho Bankproxied (được bao bọc) bởi một đối tượng truy cập dữ liệu (DAO), đối tượng này được ủy quyền bởi một nguồn dữ liệu trình điều khiển JDBC. (Xem Hình 11-3.)

**Figure 11-3  
Hình 11-3**

The “Russian doll” of decorators  
“Búp bê Nga” của những người thợ trang trí

The client believes it is invoking getAccounts()on a Bankobject, but it is actually talk-ing to the outermost of a set of nested DECORATOR14 objects that extend the basic behavior of the Bank POJO. We could add other decorators for transactions, caching, and so forth*.*  
Máy khách tin rằng nó đang gọi getAccounts() trên một Bankobject, nhưng nó thực sự đang nói chuyện với lớp ngoài cùng của một tập hợp các đối tượng DECORATOR14 lồng nhau mở rộng hành vi cơ bản của Bank POJO. Chúng tôi có thể thêm các bộ trang trí khác cho các giao dịch, bộ nhớ đệm, v.v.

In the application, a few lines are needed to ask the DI container for the top-level objects in the system, as speciﬁed in the XML ﬁle.  
Trong ứng dụng, cần có một vài dòng để yêu cầu bộ chứa DI cho các đối tượng cấp cao nhất trong hệ thống, như được chỉ định trong tệp XML.

XmlBeanFactory bf =

new XmlBeanFactory(new ClassPathResource("app.xml", getClass())); Bank bank = (Bank) bf.getBean("bank");

Because so few lines of Spring-speciﬁc Java code are required, *the application is almost completely decoupled from Spring*, eliminating all the tight-coupling problems of systems like EJB2.  
Do yêu cầu rất ít dòng mã Java dành riêng cho Spring, nên ứng dụng gần như được tách rời hoàn toàn khỏi Spring, loại bỏ tất cả các vấn đề liên kết chặt chẽ của các hệ thống như EJB2.

Although XML can be verbose and hard to read,15 the “policy” speciﬁed in these con-ﬁguration ﬁles is simpler than the complicated proxy and aspect logic that is hidden from view and created automatically. This type of architecture is so compelling that frame-works like Spring led to a complete overhaul of the EJB standard for version 3. EJB3  
Mặc dù XML có thể dài dòng và khó đọc,15 “chính sách” được chỉ định trong các tệp cấu hình này đơn giản hơn so với logic khía cạnh và proxy phức tạp được ẩn khỏi chế độ xem và được tạo tự động. Kiểu kiến trúc này hấp dẫn đến mức các công trình khung như Spring đã dẫn đến việc đại tu hoàn toàn tiêu chuẩn EJB cho phiên bản 3. EJB3

14. [GOF].

15. The example can be simpliﬁed using mechanisms that exploit *convention over conﬁguration* and Java 5 annotations to reduce the amount of explicit “wiring” logic required.  
15. Ví dụ này có thể được đơn giản hóa bằng cách sử dụng các cơ chế khai thác quy ước về cấu hình và các chú thích Java 5 để giảm số lượng logic “đi dây” rõ ràng cần thiết.

<https://www.facebook.com/groups/phattrienlaptrinh>

**Pure JavaAOP Frameworks** 165  
Khung JavaAOP thuần túy 165

largely follows the Spring model of declaratively supporting cross-cutting concerns using XML conﬁguration ﬁles and/or Java 5 annotations.  
phần lớn tuân theo mô hình Spring hỗ trợ khai báo các mối quan tâm xuyên suốt bằng cách sử dụng tệp cấu hình XML và/hoặc chú thích Java 5.

Listing 11-5 shows our Bank object rewritten in EJB316.  
Liệt kê 11-5 cho thấy đối tượng Ngân hàng của chúng ta được viết lại trong EJB316.

**Listing 11-5**

**An EBJ3 Bank EJB  
Ngân hàng EBJ3 EJB**

package com.example.banking.model; import javax.persistence.\*; import java.util.ArrayList; import java.util.Collection;

@Entity

@Table(name = "BANKS")

public class Bank implements java.io.Serializable { @Id @GeneratedValue(strategy=GenerationType.AUTO) private int id;

@Embeddable // An object “inlined” in Bank’s DB row public class Address {

protected String streetAddr1; protected String streetAddr2; protected String city; protected String state; protected String zipCode;

}

@Embedded

private Address address;

@OneToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER, mappedBy="bank")

private Collection<Account> accounts = new ArrayList<Account>();

public int getId() { return id;

}

public void setId(int id) { this.id = id;

}

public void addAccount(Account account) { account.setBank(this); accounts.add(account);

}

public Collection<Account> getAccounts() { return accounts;

}

16. Adapted from <http://www.onjava.com/pub/a/onjava/2006/05/17/standardizing-with-ejb3-java-persistence-api.html>

<https://www.facebook.com/groups/phattrienlaptrinh>

166 **Chapter 11: Systems**  
166 Chương 11: Hệ thống

**Listing 11-5 (continued) An EBJ3 Bank EJB**  
Liệt kê 11-5 (tiếp theo) Ngân hàng EBJ3 EJB

public void setAccounts(Collection<Account> accounts) { this.accounts = accounts;

} }

This code is much cleaner than the original EJB2 code. Some of the entity details are still here, contained in the annotations. However, because none of that information is out-side of the annotations, the code is clean, clear, and hence easy to test drive, maintain, and so on.  
Mã này sạch hơn nhiều so với mã EJB2 ban đầu. Một số chi tiết thực thể vẫn còn ở đây, có trong các chú thích. Tuy nhiên, do không có thông tin nào trong số đó nằm ngoài chú thích nên mã sạch sẽ, rõ ràng và do đó dễ chạy thử, bảo trì, v.v.

Some or all of the persistence information in the annotations can be moved to XML deployment descriptors, if desired, leaving a truly pure POJO. If the persistence mapping details won’t change frequently, many teams may choose to keep the annotations, but with far fewer harmful drawbacks compared to the EJB2 invasiveness.  
Một số hoặc tất cả thông tin lưu trữ trong các chú thích có thể được chuyển sang các bộ mô tả triển khai XML, nếu muốn, để lại một POJO thực sự thuần túy. Nếu các chi tiết ánh xạ liên tục không thay đổi thường xuyên, nhiều nhóm có thể chọn giữ lại các chú thích, nhưng có ít nhược điểm có hại hơn nhiều so với khả năng xâm lấn của EJB2.

[**AspectJ Aspects**](#_page_383_0)  
Khía cạnhJ Khía cạnh

Finally, the most full-featured tool for separating concerns through aspects is the AspectJ language,17 an extension of Java that provides “ﬁrst-class” support for aspects as modular-ity constructs. The pure Java approaches provided by Spring AOP and JBoss AOP are suf-ﬁcient for 80–90 percent of the cases where aspects are most useful. However, AspectJ provides a very rich and powerful tool set for separating concerns. The drawback of AspectJ is the need to adopt several new tools and to learn new language constructs and usage idioms.  
Cuối cùng, công cụ đầy đủ tính năng nhất để phân tách các mối quan tâm thông qua các khía cạnh là ngôn ngữ AspectJ,17 một phần mở rộng của Java cung cấp hỗ trợ “hạng nhất” cho các khía cạnh dưới dạng cấu trúc mô-đun. Các cách tiếp cận Java thuần túy do Spring AOP và JBoss AOP cung cấp là đủ cho 80–90 phần trăm các trường hợp mà các khía cạnh là hữu ích nhất. Tuy nhiên, AspectJ cung cấp một bộ công cụ rất phong phú và mạnh mẽ để phân tách các mối quan tâm. Hạn chế của AspectJ là cần áp dụng một số công cụ mới và học các cấu trúc ngôn ngữ mới và các thành ngữ sử dụng.

The adoption issues have been partially mitigated by a recently introduced “annota-tion form” of AspectJ, where Java 5 annotations are used to deﬁne aspects using pure Java code. Also, the Spring Framework has a number of features that make incorporation of annotation-based aspects much easier for a team with limited AspectJ experience.  
Các vấn đề về việc áp dụng đã được giảm thiểu một phần bởi một “biểu mẫu chú thích” được giới thiệu gần đây của AspectJ, trong đó các chú thích Java 5 được sử dụng để xác định các khía cạnh sử dụng mã Java thuần túy. Ngoài ra, Spring Framework có một số tính năng giúp kết hợp các khía cạnh dựa trên chú thích dễ dàng hơn nhiều đối với nhóm có kinh nghiệm AspectJ hạn chế.

A full discussion of AspectJ is beyond the scope of this book. See [AspectJ], [Colyer], and [Spring] for more information.  
Thảo luận đầy đủ về AspectJ nằm ngoài phạm vi của cuốn sách này. Xem [AspectJ], [Colyer] và [Spring] để biết thêm thông tin.

[**Test Drive the System Architecture**](#_page_383_0)  
Lái thử kiến trúc hệ thống

The power of separating concerns through aspect-like approaches can’t be overstated. If you can write your application’s domain logic using POJOs, decoupled from any architec-ture concerns at the code level, then it is possible to truly *test drive* your architecture. You can evolve it from simple to sophisticated, as needed, by adopting new technologies on  
Không thể cường điệu hóa sức mạnh của việc phân tách các mối quan tâm thông qua các cách tiếp cận giống như khía cạnh. Nếu bạn có thể viết logic miền của ứng dụng bằng cách sử dụng POJO, được tách rời khỏi bất kỳ mối quan tâm nào về kiến trúc ở cấp độ mã, thì bạn có thể thực sự kiểm tra kiến trúc của mình. Bạn có thể phát triển nó từ đơn giản đến phức tạp, khi cần, bằng cách áp dụng các công nghệ mới trên

17. See [AspectJ] and [Colyer].  
17. Xem [AspectJ] và [Colyer].

<https://www.facebook.com/groups/phattrienlaptrinh>

**Optimize Decision Making** 167  
Tối ưu hóa việc ra quyết định 167

demand. It is not necessary to do a *Big Design Up Front*18 (BDUF). In fact, BDUF is even harmful because it inhibits adapting to change, due to the psychological resistance to dis-carding prior effort and because of the way architecture choices inﬂuence subsequent thinking about the design.  
yêu cầu. Không cần thiết phải thực hiện Big Design Up Front18 (BDUF). Trên thực tế, BDUF thậm chí còn có hại vì nó cản trở việc thích nghi với sự thay đổi, do tâm lý chống lại việc từ bỏ nỗ lực trước đó và do cách các lựa chọn kiến trúc ảnh hưởng đến suy nghĩ tiếp theo về thiết kế.

Building architects have to do BDUF because it is not feasible to make radical archi-tectural changes to a large physical structure once construction is well underway.19 Although software has its own *physics*,20 it is economically feasible to make radical change, *if* the structure of the software separates its concerns effectively.  
Các kiến trúc sư xây dựng phải thực hiện BDUF vì không khả thi để thực hiện các thay đổi kiến trúc triệt để đối với một cấu trúc vật lý lớn một khi quá trình xây dựng đang được tiến hành tốt.19 Mặc dù phần mềm có tính chất vật lý riêng,20 nhưng việc thực hiện thay đổi triệt để là khả thi về mặt kinh tế, nếu cấu trúc của phần mềm phân tách mối quan tâm của nó một cách hiệu quả.

This means we can start a software project with a “naively simple” but nicely decou-pled architecture, delivering working user stories quickly, then adding more infrastructure as we scale up. Some of the world’s largest Web sites have achieved very high availability and performance, using sophisticated data caching, security, virtualization, and so forth, all done efﬁciently and ﬂexibly because the minimally coupled designs are appropriately *simple* at each level of abstraction and scope.  
Điều này có nghĩa là chúng tôi có thể bắt đầu một dự án phần mềm với kiến trúc “đơn giản ngây thơ” nhưng độc đáo, cung cấp các câu chuyện của người dùng đang hoạt động một cách nhanh chóng, sau đó thêm nhiều cơ sở hạ tầng hơn khi chúng tôi mở rộng quy mô. Một số trang Web lớn nhất thế giới đã đạt được tính khả dụng và hiệu suất rất cao, sử dụng bộ nhớ đệm dữ liệu tinh vi, bảo mật, ảo hóa, v.v., tất cả đều được thực hiện hiệu quả và linh hoạt bởi vì các thiết kế kết hợp tối thiểu đơn giản phù hợp ở từng mức độ trừu tượng và phạm vi.

Of course, this does not mean that we go into a project “rudderless.” We have some expectations of the general scope, goals, and schedule for the project, as well as the gen-eral structure of the resulting system. However, we must maintain the ability to change course in response to evolving circumstances.  
Tất nhiên, điều này không có nghĩa là chúng ta tham gia vào một dự án “không có bánh lái”. Chúng tôi có một số kỳ vọng về phạm vi, mục tiêu và lịch trình chung cho dự án, cũng như cấu trúc chung của hệ thống kết quả. Tuy nhiên, chúng ta phải duy trì khả năng thay đổi hướng đi để đáp ứng với hoàn cảnh phát triển.

The early EJB architecture is but one of many well-known APIs that are over-engi-neered and that compromise separation of concerns. Even well-designed APIs can be over-kill when they aren’t really needed. A good API should largely *disappear* from view most of the time, so the team expends the majority of its creative efforts focused on the user sto-ries being implemented. If not, then the architectural constraints will inhibit the efﬁcient delivery of optimal value to the customer.  
Kiến trúc EJB ban đầu chỉ là một trong nhiều API nổi tiếng được sử dụng quá nhiều công nghệ và làm ảnh hưởng đến việc phân tách các mối quan tâm. Ngay cả các API được thiết kế tốt cũng có thể bị tiêu diệt quá mức khi chúng không thực sự cần thiết. Hầu hết thời gian, một API tốt sẽ biến mất khỏi tầm nhìn, vì vậy nhóm dành phần lớn nỗ lực sáng tạo của mình để tập trung vào các câu chuyện người dùng đang được triển khai. Nếu không, thì các ràng buộc kiến trúc sẽ cản trở việc cung cấp hiệu quả giá trị tối ưu cho khách hàng.

To recap this long discussion,  
Để tóm tắt cuộc thảo luận dài này,

*An optimal system architecture consists of modularized domains of concern, each of which is implemented with Plain Old Java (or other) Objects. The different domains are inte-grated together with minimally invasive Aspects or Aspect-like tools. This architecture can be test-driven, just like the code.  
Một kiến trúc hệ thống tối ưu bao gồm các miền quan tâm được mô đun hóa, mỗi miền trong số đó được triển khai với các Đối tượng Java (hoặc khác) thuần túy. Các miền khác nhau được tích hợp cùng với các Khía cạnh xâm lấn tối thiểu hoặc các công cụ giống như Khía cạnh. Kiến trúc này có thể được điều khiển thử nghiệm, giống như mã.*

[**Optimize Decision Making**](#_page_383_0)  
Tối ưu hóa việc ra quyết định

Modularity and separation of concerns make decentralized management and decision making possible. In a sufﬁciently large system, whether it is a city or a software project, no one person can make all the decisions.  
Tính mô đun và sự tách biệt của các mối quan tâm làm cho việc quản lý phi tập trung và ra quyết định trở nên khả thi. Trong một hệ thống đủ lớn, cho dù đó là một thành phố hay một dự án phần mềm, không một người nào có thể đưa ra tất cả các quyết định.

18. Not to be confused with the good practice of up-front design, BDUF is the practice of designing *everything* up front before implementing anything at all.  
18. Đừng nhầm lẫn với thông lệ tốt về thiết kế trước, BDUF là thông lệ thiết kế mọi thứ trước khi thực hiện bất kỳ điều gì.

19. There is still a signiﬁcant amount of iterative exploration and discussion of details, even after construction starts. 20. The term *software physics* was ﬁrst used by [Kolence].  
19. Vẫn còn một số lượng đáng kể các cuộc thăm dò lặp đi lặp lại và thảo luận về các chi tiết, ngay cả sau khi bắt đầu xây dựng. 20. Thuật ngữ vật lý phần mềm lần đầu tiên được sử dụng bởi [Kolence].

<https://www.facebook.com/groups/phattrienlaptrinh>

168 **Chapter 11: Systems**  
168 Chương 11: Hệ thống

We all know it is best to give responsibilities to the most qualiﬁed persons. We often forget that it is also best to *postpone decisions until the last possible moment*. This isn’t lazy or irresponsible; it lets us make informed choices with the best possible information. A premature decision is a decision made with suboptimal knowledge. We will have that much less customer feedback, mental reﬂection on the project, and experience with our implementation choices if we decide too soon.  
Tất cả chúng ta đều biết rằng tốt nhất là giao trách nhiệm cho những người có năng lực nhất. Chúng ta thường quên rằng tốt nhất là trì hoãn các quyết định cho đến thời điểm cuối cùng có thể. Đây không phải là lười biếng hay vô trách nhiệm; nó cho phép chúng tôi đưa ra những lựa chọn sáng suốt với thông tin tốt nhất có thể. Một quyết định sớm là một quyết định được đưa ra với kiến thức dưới mức tối ưu. Chúng tôi sẽ có ít phản hồi của khách hàng hơn, phản ánh tinh thần về dự án và trải nghiệm với các lựa chọn triển khai của chúng tôi nếu chúng tôi quyết định quá sớm.

*The agility provided by a POJO system with modularized concerns allows us to make opti-mal, just-in-time decisions, based on the most recent knowledge. The complexity of these decisions is also reduced.*

[**Use Standards Wisely, When They Add *Demonstrable*Value**](#_page_383_0)  
Sử dụng các tiêu chuẩn một cách khôn ngoan, khi chúng thêm giá trị chứng minh

Building construction is a marvel to watch because of the pace at which new buildings are built (even in the dead of winter) and because of the extraordinary designs that are possi-ble with today’s technology. Construction is a mature industry with highly optimized parts, methods, and standards that have evolved under pressure for centuries.  
Việc xây dựng tòa nhà là một điều kỳ diệu đáng xem vì tốc độ xây dựng các tòa nhà mới (ngay cả trong cái chết của mùa đông) và vì những thiết kế đặc biệt có thể thực hiện được với công nghệ ngày nay. Xây dựng là một ngành trưởng thành với các bộ phận, phương pháp và tiêu chuẩn được tối ưu hóa cao đã phát triển dưới áp lực trong nhiều thế kỷ.

Many teams used the EJB2 architecture because it was a standard, even when lighter-weight and more straightforward designs would have been sufﬁcient. I have seen teams become obsessed with various *strongly hyped* standards and lose focus on implementing value for their customers.  
Nhiều nhóm đã sử dụng kiến trúc EJB2 vì nó là một tiêu chuẩn, ngay cả khi các thiết kế nhẹ hơn và đơn giản hơn là đủ. Tôi đã chứng kiến các nhóm bị ám ảnh bởi nhiều tiêu chuẩn được thổi phồng quá mức và mất tập trung vào việc thực hiện giá trị cho khách hàng của họ.

*Standards make it easier to reuse ideas and components, recruit people with relevant expe-rience, encapsulate good ideas, and wire components together. However, the process of creating standards can sometimes take too long for industry to wait, and some standards lose touch with the real needs of the adopters they are intended to serve.*

[**Systems Need Domain-Speciﬁc Languages**](#_page_383_0)  
Hệ thống cần ngôn ngữ dành riêng cho miền

Building construction, like most domains, has developed a rich language with a vocabu-lary, idioms, and patterns21 that convey essential information clearly and concisely. In soft-ware, there has been renewed interest recently in creating *Domain-Speciﬁc Languages* (DSLs),22 which are separate, small scripting languages or APIs in standard languages that permit code to be written so that it reads like a structured form of prose that a domain expert might write.  
Xây dựng công trình, giống như hầu hết các lĩnh vực, đã phát triển một ngôn ngữ phong phú với vốn từ vựng, thành ngữ và mẫu21 truyền tải thông tin cần thiết một cách rõ ràng và chính xác. Trong lĩnh vực phần mềm, gần đây người ta lại quan tâm đến việc tạo ra các Ngôn ngữ dành riêng cho miền (DSL),22 là các ngôn ngữ kịch bản nhỏ, riêng biệt hoặc các API trong các ngôn ngữ tiêu chuẩn cho phép mã được viết sao cho nó đọc giống như một dạng văn xuôi có cấu trúc. mà một chuyên gia tên miền có thể viết.

A good DSL minimizes the “communication gap” between a domain concept and the code that implements it, just as agile practices optimize the communications within a team and with the project’s stakeholders. If you are implementing domain logic in the  
Một DSL tốt sẽ giảm thiểu “khoảng cách giao tiếp” giữa khái niệm miền và mã triển khai nó, giống như các phương pháp linh hoạt tối ưu hóa giao tiếp trong nhóm và với các bên liên quan của dự án. Nếu bạn đang triển khai logic miền trong

21. The work of [Alexander] has been particularly inﬂuential on the software community. 22. See, for example, [DSL]. [JMock] is a good example of a JavaAPI that creates a DSL.  
21. Công trình của [Alexander] có ảnh hưởng đặc biệt đến cộng đồng phần mềm. 22. Ví dụ, xem [DSL]. [JMock] là một ví dụ điển hình về JavaAPI tạo DSL.

<https://www.facebook.com/groups/phattrienlaptrinh>

**Bibliography** 169  
Thư mục 169

same language that a domain expert uses, there is less risk that you will incorrectly trans-late the domain into the implementation.  
cùng một ngôn ngữ mà chuyên gia miền sử dụng, sẽ có ít rủi ro hơn là bạn sẽ chuyển miền không chính xác sang quá trình triển khai.

DSLs, when used effectively, raise the abstraction level above code idioms and design patterns. They allow the developer to reveal the intent of the code at the appropriate level of abstraction.  
DSL, khi được sử dụng hiệu quả, sẽ nâng mức độ trừu tượng lên trên các thành ngữ mã và mẫu thiết kế. Chúng cho phép nhà phát triển tiết lộ ý định của mã ở mức độ trừu tượng thích hợp.

*Domain-Speciﬁc Languages allow all levels of abstraction and all domains in the applica-tion to be expressed as POJOs, from high-level policy to low-level details.*

[**Conclusion**](#_page_383_0)  
Phần kết luận

Systems must be clean too. An invasive architecture overwhelms the domain logic and impacts agility. When the domain logic is obscured, quality suffers because bugs ﬁnd it easier to hide and stories become harder to implement. If agility is compromised, produc-tivity suffers and the beneﬁts of TDD are lost.  
Hệ thống cũng phải sạch sẽ. Kiến trúc xâm lấn lấn át logic miền và tác động đến sự nhanh nhẹn. Khi logic miền bị che khuất, chất lượng sẽ bị ảnh hưởng bởi vì các lỗi sẽ dễ dàng che giấu hơn và các câu chuyện trở nên khó thực hiện hơn. Nếu tính linh hoạt bị tổn hại, năng suất sẽ bị ảnh hưởng và lợi ích của TDD sẽ bị mất.

At all levels of abstraction, the intent should be clear. This will only happen if you write POJOs and you use aspect-like mechanisms to incorporate other implementation concerns noninvasively.  
Ở tất cả các cấp độ trừu tượng, ý định phải rõ ràng. Điều này sẽ chỉ xảy ra nếu bạn viết POJO và bạn sử dụng các cơ chế giống như khía cạnh để kết hợp các mối quan tâm triển khai khác một cách không xâm phạm.

Whether you are designing systems or individual modules, never forget to *use the simplest thing that can possibly work*.  
Cho dù bạn đang thiết kế các hệ thống hay các mô-đun riêng lẻ, đừng bao giờ quên sử dụng thứ đơn giản nhất có thể hoạt động được.

[**Bibliography**](#_page_383_0)  
Thư mục

**[Alexander]:** Christopher Alexander, *A Timeless Way of Building,* Oxford University Press, New York, 1979.

**[AOSD]:** Aspect-Oriented Software Development port, <http://aosd.net>

**[ASM]:** ASM Home Page, <http://asm.objectweb.org/>

**[AspectJ]:** <http://eclipse.org/aspectj>

**[CGLIB]:** Code Generation Library, <http://cglib.sourceforge.net/>

**[Colyer]:** Adrian Colyer, Andy Clement, George Hurley, Mathew Webster, *Eclipse AspectJ,* Person Education, Inc., Upper Saddle River, NJ, 2005.

**[DSL]:** Domain-speciﬁc programming language, [http://en.wikipedia.org/wiki/Domain-speciﬁc\_programming\_language](http://en.wikipedia.org/wiki/Domainspecific_programming_language)

**[Fowler]:** Inversion of Control Containers and the Dependency Injection pattern, <http://martinfowler.com/articles/injection.html>

<https://www.facebook.com/groups/phattrienlaptrinh>

170 **Chapter 11: Systems**  
170 Chương 11: Hệ thống

**[Goetz]:** Brian Goetz, *Java Theory and Practice: Decorating with Dynamic Proxie*s, <http://www.ibm.com/developerworks/java/library/j-jtp08305.html>

**[Javassist]:** Javassist Home Page, <http://www.csg.is.titech.ac.jp/~chiba/javassist/>

**[JBoss]:** JBoss Home Page, <http://jboss.org>

**[JMock]:** JMock—A Lightweight Mock Object Library for Java, <http://jmock.org>

**[Kolence]:** Kenneth W. Kolence, Software physics and computer performance measure-ments, *Proceedings of the ACM annual conference—Volume 2*, Boston, Massachusetts, pp. 1024–1040, 1972.

**[Spring]:** *The Spring Framework*, <http://www.springframework.org>

**[Mezzaros07]:** *XUnit Patterns*, Gerard Mezzaros, Addison-Wesley, 2007.

**[GOF]:** *Design Patterns: Elements of Reusable Object Oriented Software*, Gamma et al., Addison-Wesley, 1996.

<https://www.facebook.com/groups/phattrienlaptrinh>

[**12**](#_page_383_0)

[**Emergence**](#_page_383_0)  
Sự xuất hiện

by Jeff Langr  
bởi Jeff Langr

[**Getting Clean via Emergent Design**](#_page_383_0)  
Làm sạch thông qua thiết kế mới nổi

What if there were four simple rules that you could follow that would help you create good designs as you worked? What if by following these rules you gained insights into the struc-ture and design of your code, making it easier to apply principles such as SRP and DIP? What if these four rules facilitated the *emergence* of good designs?  
Điều gì sẽ xảy ra nếu có bốn quy tắc đơn giản mà bạn có thể tuân theo để giúp bạn tạo ra những thiết kế đẹp khi làm việc? Điều gì sẽ xảy ra nếu bằng cách tuân theo các quy tắc này, bạn đã hiểu rõ hơn về cấu trúc và thiết kế mã của mình, giúp việc áp dụng các nguyên tắc như SRP và DIP trở nên dễ dàng hơn? Điều gì sẽ xảy ra nếu bốn quy tắc này tạo điều kiện cho sự xuất hiện của những thiết kế tốt?

Many of us feel that Kent Beck’s four rules of *Simple Design*1 are of signiﬁcant help in creating well-designed software.  
Nhiều người trong chúng ta cảm thấy rằng bốn quy tắc của Thiết kế đơn giản1 của Kent Beck giúp ích rất nhiều trong việc tạo ra phần mềm được thiết kế tốt.

1. [XPE].

171

<https://www.facebook.com/groups/phattrienlaptrinh>

172 **Chapter 12: Emergence**  
172 Chương 12: Xuất Thế

According to Kent, a design is “simple” if it follows these rules:  
Theo Kent, một thiết kế là “đơn giản” nếu nó tuân theo các quy tắc sau:

• Runs all the tests  
• Chạy tất cả các bài kiểm tra

• Contains no duplication  
• Không chứa bản sao

• Expresses the intent of the programmer  
• Thể hiện ý đồ của người lập trình

• Minimizes the number of classes and methods  
• Giảm thiểu số lượng lớp và phương thức

The rules are given in order of importance.  
Các quy tắc được đưa ra theo thứ tự quan trọng.

[**Simple Design Rule 1: Runs All the Tests**](#_page_383_0)  
Quy tắc thiết kế đơn giản 1: Chạy tất cả các bài kiểm tra

First and foremost, a design must produce a system that acts as intended. A system might have a perfect design on paper, but if there is no simple way to verify that the system actu-ally works as intended, then all the paper effort is questionable.  
Đầu tiên và quan trọng nhất, một thiết kế phải tạo ra một hệ thống hoạt động như dự kiến. Một hệ thống có thể có một thiết kế hoàn hảo trên giấy tờ, nhưng nếu không có cách đơn giản nào để xác minh rằng hệ thống đó thực sự hoạt động như dự kiến, thì mọi nỗ lực trên giấy tờ đều đáng nghi ngờ.

A system that is comprehensively tested and passes all of its tests all of the time is a test-able system. That’s an obvious statement, but an important one. Systems that aren’t testable aren’t veriﬁable. Arguably, a system that cannot be veriﬁed should never be deployed.  
Một hệ thống được kiểm tra toàn diện và luôn vượt qua tất cả các bài kiểm tra của nó là một hệ thống có thể kiểm tra. Đó là một tuyên bố rõ ràng, nhưng là một tuyên bố quan trọng. Các hệ thống không thể kiểm tra được sẽ không thể kiểm chứng được. Có thể cho rằng, một hệ thống không thể được xác minh sẽ không bao giờ được triển khai.

Fortunately, making our systems testable pushes us toward a design where our classes are small and single purpose. It’s just easier to test classes that conform to the SRP. The more tests we write, the more we’ll continue to push toward things that are simpler to test. So making sure our system is fully testable helps us create better designs.  
May mắn thay, việc làm cho các hệ thống của chúng tôi có thể kiểm tra được đã thúc đẩy chúng tôi hướng tới một thiết kế trong đó các lớp của chúng tôi nhỏ và có mục đích duy nhất. Việc kiểm tra các lớp phù hợp với SRP sẽ dễ dàng hơn. Càng viết nhiều bài kiểm tra, chúng tôi càng tiếp tục hướng tới những thứ đơn giản hơn để kiểm tra. Vì vậy, đảm bảo hệ thống của chúng tôi có thể kiểm tra đầy đủ sẽ giúp chúng tôi tạo ra các thiết kế tốt hơn.

Tight coupling makes it difﬁcult to write tests. So, similarly, the more tests we write, the more we use principles like DIP and tools like dependency injection, interfaces, and abstraction to minimize coupling. Our designs improve even more.  
Khớp nối chặt chẽ khiến việc viết bài kiểm tra trở nên khó khăn. Vì vậy, tương tự, càng viết nhiều bài kiểm tra, chúng tôi càng sử dụng nhiều nguyên tắc như DIP và các công cụ như tiêm phụ thuộc, giao diện và trừu tượng hóa để giảm thiểu khớp nối. thiết kế của chúng tôi cải thiện nhiều hơn nữa.

Remarkably, following a simple and obvious rule that says we need to have tests and run them continuously impacts our system’s adherence to the primary OO goals of low coupling and high cohesion. Writing tests leads to better designs.  
Đáng chú ý, việc tuân theo một quy tắc đơn giản và rõ ràng cho biết chúng tôi cần phải có các thử nghiệm và chạy chúng liên tục sẽ tác động đến việc hệ thống của chúng tôi tuân thủ các mục tiêu OO chính là khớp nối thấp và gắn kết cao. Viết bài kiểm tra dẫn đến thiết kế tốt hơn.

[**Simple Design Rules 2–4: Refactoring**](#_page_383_0)  
Quy tắc thiết kế đơn giản 2–4: Tái cấu trúc

Once we have tests, we are empowered to keep our code and classes clean. We do this by incrementally refactoring the code. For each few lines of code we add, we pause and reﬂect on the new design. Did we just degrade it? If so, we clean it up and run our tests to demon-strate that we haven’t broken anything. *The fact that we have these tests eliminates the fear that cleaning up the code will break it!*  
Khi chúng tôi có các bài kiểm tra, chúng tôi được trao quyền để giữ cho mã và các lớp của chúng tôi sạch sẽ. Chúng tôi làm điều này bằng cách tái cấu trúc mã dần dần. Đối với mỗi vài dòng mã chúng tôi thêm vào, chúng tôi tạm dừng và suy nghĩ về thiết kế mới. Có phải chúng ta vừa làm suy giảm nó? Nếu vậy, chúng tôi dọn sạch nó và chạy thử nghiệm để chứng minh rằng chúng tôi không làm hỏng bất kỳ thứ gì. Thực tế là chúng tôi có các bài kiểm tra này giúp loại bỏ nỗi sợ rằng việc dọn sạch mã sẽ phá vỡ nó!

During this refactoring step, we can apply anything from the entire body of knowledge about good software design. We can increase cohesion, decrease coupling, separate con-cerns, modularize system concerns, shrink our functions and classes, choose better names, and so on. This is also where we apply the ﬁnal three rules of simple design: Eliminate duplication, ensure expressiveness, and minimize the number of classes and methods.  
Trong bước tái cấu trúc này, chúng ta có thể áp dụng mọi thứ từ toàn bộ kiến thức về thiết kế phần mềm tốt. Chúng ta có thể tăng tính liên kết, giảm khớp nối, phân tách các mối quan tâm, mô đun hóa các mối quan tâm của hệ thống, thu gọn các chức năng và lớp của chúng ta, chọn các tên tốt hơn, v.v. Đây cũng là nơi chúng tôi áp dụng ba quy tắc cuối cùng của thiết kế đơn giản: Loại bỏ trùng lặp, đảm bảo tính biểu cảm và giảm thiểu số lượng lớp và phương thức.

<https://www.facebook.com/groups/phattrienlaptrinh>

**No Duplication** 173  
Không Trùng lặp 173

[**No Duplication**](#_page_383_0)  
Không trùng lặp

Duplication is the primary enemy of a well-designed system. It represents additional work, additional risk, and additional unnecessary complexity. Duplication manifests itself in many forms. Lines of code that look exactly alike are, of course, duplication. Lines of code that are similar can often be massaged to look even more alike so that they can be more easily refactored. And duplication can exist in other forms such as duplication of implementation. For example, we might have two methods in a collection class:  
Sao chép là kẻ thù chính của một hệ thống được thiết kế tốt. Nó đại diện cho công việc bổ sung, rủi ro bổ sung và sự phức tạp không cần thiết bổ sung. Sự trùng lặp thể hiện dưới nhiều hình thức. Tất nhiên, các dòng mã trông giống hệt nhau là trùng lặp. Các dòng mã giống nhau thường có thể được xoa bóp để trông giống nhau hơn để chúng có thể dễ dàng tái cấu trúc hơn. Và sự trùng lặp có thể tồn tại ở các dạng khác như trùng lặp về cách thực hiện. Ví dụ, chúng ta có thể có hai phương thức trong một lớp sưu tập:

int size() {} boolean isEmpty() {}

We could have separate implementations for each method. The isEmptymethod could track a boolean, while sizecould track a counter. Or, we can eliminate this duplication by tying isEmpty to the deﬁnition of size:  
Chúng tôi có thể có các triển khai riêng cho từng phương pháp. Phương thức isEmpty có thể theo dõi một boolean, trong khi sizecó thể theo dõi một bộ đếm. Hoặc, chúng ta có thể loại bỏ sự trùng lặp này bằng cách gắn isEmpty với định nghĩa về kích thước:

boolean isEmpty() { return 0 == size();

}

Creating a clean system requires the will to eliminate duplication, even in just a few lines of code. For example, consider the following code:  
Việc tạo ra một hệ thống sạch đòi hỏi phải có ý chí để loại bỏ sự trùng lặp, thậm chí chỉ trong một vài dòng mã. Ví dụ: hãy xem xét đoạn mã sau:

public void scaleToOneDimension(

float desiredDimension, float imageDimension) {

if (Math.abs(desiredDimension - imageDimension) < errorThreshold) return;

float scalingFactor = desiredDimension / imageDimension; scalingFactor = (float)(Math.floor(scalingFactor \* 100) \* 0.01f);

RenderedOp newImage = ImageUtilities.getScaledImage( image, scalingFactor, scalingFactor);

image.dispose(); System.gc(); image = newImage;

}

public synchronized void rotate(int degrees) { RenderedOp newImage = ImageUtilities.getRotatedImage(

image, degrees); image.dispose(); System.gc();

image = newImage;

}

To keep this system clean, we should eliminate the small amount of duplication between the scaleToOneDimension and rotate methods:  
Để giữ cho hệ thống này sạch sẽ, chúng ta nên loại bỏ một lượng nhỏ trùng lặp giữa các phương thức scaleToOneDimension và xoay:

public void scaleToOneDimension(

float desiredDimension, float imageDimension) {

if (Math.abs(desiredDimension - imageDimension) < errorThreshold) return;

float scalingFactor = desiredDimension / imageDimension; scalingFactor = (float)(Math.floor(scalingFactor \* 100) \* 0.01f);

<https://www.facebook.com/groups/phattrienlaptrinh>

174 **Chapter 12: Emergence**  
174 Chương 12: Xuất hiện

**replaceImage(ImageUtilities.getScaledImage( image, scalingFactor, scalingFactor));**

}

public synchronized void rotate(int degrees) { **replaceImage(ImageUtilities.getRotatedImage(image, degrees));**

}

**private void replaceImage(RenderedOp newImage) { image.dispose();**

**System.gc(); image = newImage;**

**}**

As we extract commonality at this very tiny level, we start to recognize violations of SRP. So we might move a newly extracted method to another class. That elevates its visibility. Someone else on the team may recognize the opportunity to further abstract the new method and reuse it in a different context. This “reuse in the small” can cause system com-plexity to shrink dramatically. Understanding how to achieve reuse in the small is essential to achieving reuse in the large.  
Khi chúng tôi trích xuất điểm chung ở cấp độ rất nhỏ này, chúng tôi bắt đầu nhận ra các vi phạm SRP. Vì vậy, chúng tôi có thể chuyển một phương thức mới được trích xuất sang một lớp khác. Điều đó nâng cao khả năng hiển thị của nó. Một người khác trong nhóm có thể nhận ra cơ hội để tiếp tục trừu tượng hóa phương thức mới và sử dụng lại nó trong một ngữ cảnh khác. Việc “tái sử dụng nhỏ” này có thể khiến độ phức tạp của hệ thống giảm đi đáng kể. Hiểu cách đạt được tái sử dụng trong quy mô nhỏ là điều cần thiết để đạt được tái sử dụng trong quy mô lớn.

The TEMPLATE METHOD2 pattern is a common technique for removing higher-level duplication. For example:  
Mẫu TEMPLATE METHOD2 là một kỹ thuật phổ biến để loại bỏ trùng lặp cấp cao hơn. Ví dụ:

public class VacationPolicy {

public void accrueUSDivisionVacation() {

// code to calculate vacation based on hours worked to date // ...

// code to ensure vacation meets US minimums // ...

// code to apply vaction to payroll record // ...

}

public void accrueEUDivisionVacation() {

// code to calculate vacation based on hours worked to date // ...

// code to ensure vacation meets EU minimums // ...

// code to apply vaction to payroll record // ...

} }

The code across accrueUSDivisionVacationand accrueEuropeanDivisionVacationis largely the same, with the exception of calculating legal minimums. That bit of the algorithm changes based on the employee type.  
Mã trên cộng dồnUSDivisionVacationvà tích lũyEuropeanDivisionVacation phần lớn giống nhau, ngoại trừ việc tính toán mức tối thiểu hợp pháp. Phần thuật toán đó thay đổi dựa trên loại nhân viên.

We can eliminate the obvious duplication by applying the TEMPLATE METHOD pattern.  
Chúng ta có thể loại bỏ sự trùng lặp rõ ràng bằng cách áp dụng mẫu PHƯƠNG PHÁP TEMPLATE.

abstract public class VacationPolicy { public void accrueVacation() {

**calculateBaseVacationHours();**

2. [GOF].

<https://www.facebook.com/groups/phattrienlaptrinh>

**Expressive** 175  
biểu cảm 175

**alterForLegalMinimums(); applyToPayroll();**

}

private void calculateBaseVacationHours() { /\* ... \*/ }; abstract protected void alterForLegalMinimums(); private void applyToPayroll() { /\* ... \*/ };

}

public class USVacationPolicy extends VacationPolicy { @Override protected void alterForLegalMinimums() {

// US specific logic }

}

public class EUVacationPolicy extends VacationPolicy { @Override protected void alterForLegalMinimums() {

// EU specific logic }

}

The subclasses ﬁll in the “hole” in the accrueVacationalgorithm, supplying the only bits of information that are not duplicated.  
Các lớp con điền vào “lỗ hổng” trong thuật toán tích lũyVacational, cung cấp các bit thông tin duy nhất không bị trùng lặp.

[**Expressive**](#_page_383_0)  
biểu cảm

Most of us have had the experience of working on convoluted code. Many of us have pro-duced some convoluted code ourselves. It’s easy to write code that *we* understand, because at the time we write it we’re deep in an understanding of the problem we’re trying to solve. Other maintainers of the code aren’t going to have so deep an understanding.  
Hầu hết chúng ta đều đã có kinh nghiệm làm việc với mã phức tạp. Nhiều người trong chúng ta đã tự tạo ra một số mã phức tạp. Thật dễ dàng để viết mã mà chúng tôi hiểu, bởi vì tại thời điểm chúng tôi viết mã, chúng tôi hiểu sâu về vấn đề mà chúng tôi đang cố gắng giải quyết. Những người duy trì mã khác sẽ không hiểu sâu như vậy.

The majority of the cost of a software project is in long-term maintenance. In order to minimize the potential for defects as we introduce change, it’s critical for us to be able to understand what a system does. As systems become more complex, they take more and more time for a developer to understand, and there is an ever greater opportunity for a mis-understanding. Therefore, code should clearly express the intent of its author. The clearer the author can make the code, the less time others will have to spend understanding it. This will reduce defects and shrink the cost of maintenance.  
Phần lớn chi phí của một dự án phần mềm là bảo trì dài hạn. Để giảm thiểu khả năng xảy ra lỗi khi chúng tôi đưa ra thay đổi, điều quan trọng đối với chúng tôi là có thể hiểu chức năng của một hệ thống. Khi các hệ thống trở nên phức tạp hơn, nhà phát triển sẽ mất nhiều thời gian hơn để hiểu chúng và càng có nhiều khả năng dẫn đến hiểu sai. Do đó, mã phải thể hiện rõ ràng ý định của tác giả. Tác giả có thể tạo mã càng rõ ràng thì những người khác sẽ càng mất ít thời gian để hiểu nó. Điều này sẽ làm giảm lỗi và giảm chi phí bảo trì.

You can express yourself by choosing good names. We want to be able to hear a class or function name and not be surprised when we discover its responsibilities.  
Bạn có thể thể hiện bản thân bằng cách chọn những cái tên hay. Chúng tôi muốn có thể nghe tên một lớp hoặc chức năng và không ngạc nhiên khi chúng tôi phát hiện ra trách nhiệm của nó.

You can also express yourself by keeping your functions and classes small. Small classes and functions are usually easy to name, easy to write, and easy to understand.  
Bạn cũng có thể thể hiện bản thân bằng cách giữ cho các chức năng và lớp của bạn nhỏ. Các lớp và hàm nhỏ thường dễ đặt tên, dễ viết và dễ hiểu.

You can also express yourself by using standard nomenclature. Design patterns, for example, are largely about communication and expressiveness. By using the standard pattern names, such as COMMAND or VISITOR, in the names of the classes that imple-ment those patterns, you can succinctly describe your design to other developers.  
Bạn cũng có thể thể hiện bản thân bằng cách sử dụng danh pháp tiêu chuẩn. Ví dụ, các mẫu thiết kế phần lớn là về giao tiếp và biểu cảm. Bằng cách sử dụng các tên mẫu tiêu chuẩn, chẳng hạn như LỆNH hoặc KHÁCH, trong tên của các lớp triển khai các mẫu đó, bạn có thể mô tả ngắn gọn thiết kế của mình cho các nhà phát triển khác.

Well-written unit tests are also expressive. A primary goal of tests is to act as docu-mentation by example. Someone reading our tests should be able to get a quick under-standing of what a class is all about.  
Các bài kiểm tra đơn vị được viết tốt cũng rất biểu cảm. Mục tiêu chính của các bài kiểm tra là hoạt động như tài liệu bằng ví dụ. Ai đó đọc các bài kiểm tra của chúng tôi sẽ có thể hiểu nhanh về nội dung của một lớp học.

<https://www.facebook.com/groups/phattrienlaptrinh>

176 **Chapter 12: Emergence**  
176 Chương 12: Xuất Thế

But the most important way to be expressive is to *try*. All too often we get our code working and then move on to the next problem without giving sufﬁcient thought to making that code easy for the next person to read. Remember, the most likely next person to read the code will be you.  
Nhưng cách thể hiện quan trọng nhất là phải thử. Chúng ta thường xuyên làm cho mã của mình hoạt động và sau đó chuyển sang vấn đề tiếp theo mà không suy nghĩ đầy đủ để làm cho mã đó dễ đọc cho người tiếp theo. Hãy nhớ rằng, rất có thể người tiếp theo đọc mã sẽ là bạn.

So take a little pride in your workmanship. Spend a little time with each of your func-tions and classes. Choose better names, split large functions into smaller functions, and generally just take care of what you’ve created. Care is a precious resource.  
Vì vậy, hãy tự hào một chút về tay nghề của bạn. Dành một ít thời gian cho từng chức năng và lớp học của bạn. Chọn tên hay hơn, chia các chức năng lớn thành các chức năng nhỏ hơn và nói chung chỉ quan tâm đến những gì bạn đã tạo. Chăm sóc là một nguồn tài nguyên quý giá.

[**Minimal Classes and Methods**](#_page_383_0)  
Các lớp và phương thức tối thiểu

Even concepts as fundamental as elimination of duplication, code expressiveness, and the SRP can be taken too far. In an effort to make our classes and methods small, we might create too many tiny classes and methods. So this rule suggests that we also keep our func-tion and class counts low.  
Ngay cả những khái niệm cơ bản như loại bỏ trùng lặp, tính biểu cảm của mã và SRP cũng có thể bị đưa đi quá xa. Trong nỗ lực làm cho các lớp và phương thức của chúng ta trở nên nhỏ, chúng ta có thể tạo ra quá nhiều lớp và phương thức nhỏ. Vì vậy, quy tắc này gợi ý rằng chúng tôi cũng giữ cho số lượng chức năng và lớp học của mình ở mức thấp.

High class and method counts are sometimes the result of pointless dogmatism. Con-sider, for example, a coding standard that insists on creating an interface for each and every class. Or consider developers who insist that ﬁelds and behavior must always be sep-arated into data classes and behavior classes. Such dogma should be resisted and a more pragmatic approach adopted.  
Số lượng phương thức và lớp cao đôi khi là kết quả của chủ nghĩa giáo điều vô nghĩa. Ví dụ, hãy xem xét một tiêu chuẩn viết mã nhấn mạnh vào việc tạo ra một giao diện cho mỗi và mọi lớp. Hoặc xem xét các nhà phát triển nhấn mạnh rằng các trường và hành vi phải luôn được tách biệt thành các lớp dữ liệu và các lớp hành vi. Giáo điều như vậy nên được chống lại và áp dụng một cách tiếp cận thực dụng hơn.

Our goal is to keep our overall system small while we are also keeping our functions and classes small. Remember, however, that this rule is the lowest priority of the four rules of Simple Design. So, although it’s important to keep class and function count low, it’s more important to have tests, eliminate duplication, and express yourself.  
Mục tiêu của chúng tôi là giữ cho hệ thống tổng thể của chúng tôi nhỏ trong khi chúng tôi cũng giữ cho các hàm và lớp của mình nhỏ. Tuy nhiên, hãy nhớ rằng quy tắc này là ưu tiên thấp nhất trong bốn quy tắc của Thiết kế đơn giản. Vì vậy, mặc dù điều quan trọng là phải giữ cho số lượng lớp và chức năng ở mức thấp, nhưng điều quan trọng hơn là phải có các bài kiểm tra, loại bỏ sự trùng lặp và thể hiện bản thân.

[**Conclusion**](#_page_383_0)  
Phần kết luận

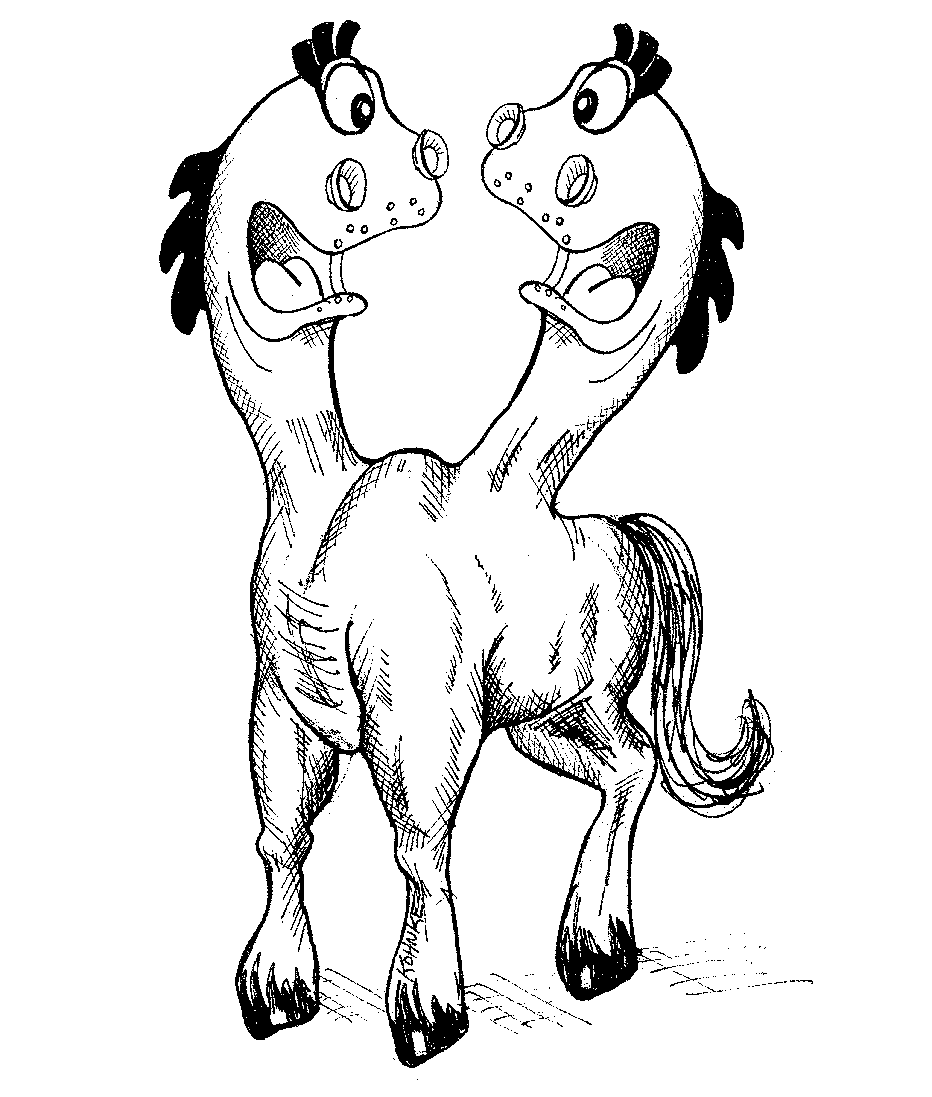
Is there a set of simple practices that can replace experience? Clearly not. On the other hand, the practices described in this chapter and in this book are a crystallized form of the many decades of experience enjoyed by the authors. Following the practice of simple design can and does encourage and enable developers to adhere to good principles and patterns that otherwise take years to learn.  
Có một tập hợp các thực hành đơn giản nào có thể thay thế kinh nghiệm không? Rõ ràng không. Mặt khác, những phương pháp thực hành được mô tả trong chương này và trong cuốn sách này là một dạng kết tinh của nhiều thập kỷ kinh nghiệm mà các tác giả đã trải qua. Việc tuân theo thực hành thiết kế đơn giản có thể và thực sự khuyến khích và cho phép các nhà phát triển tuân thủ các nguyên tắc và mẫu tốt mà nếu không thì phải mất nhiều năm mới học được.

[**Bibliography**](#_page_383_0)  
Thư mục

**[XPE]:** *Extreme Programming Explained: Embrace Change,* Kent Beck, Addison-Wesley, 1999.

**[GOF]:** *Design Patterns: Elements of Reusable Object Oriented Software,* Gamma et al., Addison-Wesley, 1996.

<https://www.facebook.com/groups/phattrienlaptrinh>



[**13**](#_page_383_0)

[**Concurrency**](#_page_383_0)  
đồng thời

by Brett L. Schuchert

*“Objects are abstractions of processing. Threads are abstractions of schedule.”  
“Các đối tượng là sự trừu tượng hóa của quá trình xử lý. Chủ đề là trừu tượng của lịch trình.*

—James O. Coplien1

1. Private correspondence.  
1. Thư từ riêng tư.

177

<https://www.facebook.com/groups/phattrienlaptrinh>

178 **Chapter 13: Concurrency**  
178 Chương 13: Đồng thời

Writing clean concurrent programs is hard—very hard. It is much easier to write code that executes in a single thread. It is also easy to write multithreaded code that looks ﬁne on the surface but is broken at a deeper level. Such code works ﬁne until the system is placed under stress.  
Viết các chương trình đồng thời rõ ràng là khó - rất khó. Việc viết mã thực thi trong một luồng sẽ dễ dàng hơn nhiều. Nó cũng dễ dàng để viết mã đa luồng có vẻ tốt trên bề mặt nhưng bị hỏng ở cấp độ sâu hơn. Mã như vậy hoạt động tốt cho đến khi hệ thống bị căng thẳng.

In this chapter we discuss the need for concurrent programming, and the difﬁculties it presents. We then present several recommendations for dealing with those difﬁculties, and writing clean concurrent code. Finally, we conclude with issues related to testing concurrent code.  
Trong chương này, chúng ta thảo luận về nhu cầu lập trình đồng thời và những khó khăn mà nó gây ra. Sau đó, chúng tôi trình bày một số khuyến nghị để giải quyết những khó khăn đó và viết mã đồng thời rõ ràng. Cuối cùng, chúng tôi kết luận với các vấn đề liên quan đến kiểm tra mã đồng thời.

Clean Concurrency is a complex topic, worthy of a book by itself. Our strategy in *this* book is to present an overview here and provide a more detailed tutorial in “Concurrency II” on page 317. If you are just curious about concurrency, then this chapter will sufﬁce for you now. If you have a need to understand concurrency at a deeper level, then you should read through the tutorial as well.  
Đồng thời sạch là một chủ đề phức tạp, bản thân nó xứng đáng là một cuốn sách. Chiến lược của chúng tôi trong cuốn sách này là trình bày tổng quan ở đây và cung cấp hướng dẫn chi tiết hơn trong “Đồng thời II” ở trang 317. Nếu bạn chỉ tò mò về đồng thời, thì chương này sẽ đủ cho bạn ngay bây giờ. Nếu bạn có nhu cầu hiểu về đồng thời ở mức độ sâu hơn, thì bạn cũng nên đọc qua hướng dẫn.

[**Why Concurrency?**](#_page_383_0)  
Tại sao đồng thời?

Concurrency is a decoupling strategy. It helps us decouple *what* gets done from *when* it gets done. In single-threaded applications *what* and *when* are so strongly coupled that the state of the entire application can often be determined by looking at the stack backtrace. A programmer who debugs such a system can set a breakpoint, or a sequence of breakpoints, and *know* the state of the system by which breakpoints are hit.  
Đồng thời là một chiến lược tách rời. Nó giúp chúng ta tách biệt giữa việc hoàn thành và việc hoàn thành. Trong các ứng dụng đơn luồng, cái gì và khi nào được kết hợp chặt chẽ đến mức trạng thái của toàn bộ ứng dụng thường có thể được xác định bằng cách nhìn vào dấu vết ngược của ngăn xếp. Một lập trình viên gỡ lỗi một hệ thống như vậy có thể đặt một điểm ngắt hoặc một chuỗi các điểm ngắt và biết trạng thái của hệ thống mà các điểm ngắt được nhấn.

Decoupling *what* from *when* can dramatically improve both the throughput and struc-tures of an application. From a structural point of view the application looks like many lit-tle collaborating computers rather than one big main loop. This can make the system easier to understand and offers some powerful ways to separate concerns.  
Việc tách rời những gì khỏi thời điểm có thể cải thiện đáng kể cả thông lượng và cấu trúc của một ứng dụng. Từ quan điểm cấu trúc, ứng dụng trông giống như nhiều máy tính cộng tác nhỏ hơn là một vòng lặp chính lớn. Điều này có thể làm cho hệ thống dễ hiểu hơn và cung cấp một số cách hiệu quả để phân tách các mối quan tâm.

Consider, for example, the standard “Servlet” model of Web applications. These sys-tems run under the umbrella of a Web or EJB container that *partially* manages concur-rency for you. The servlets are executed asynchronously whenever Web requests come in. The servlet programmer does not have to manage all the incoming requests. *In principle,* each servlet execution lives in its own little world and is decoupled from all the other serv-let executions.  
Ví dụ, hãy xem xét mô hình “Servlet” tiêu chuẩn của các ứng dụng Web. Các hệ thống này chạy dưới sự bảo trợ của bộ chứa Web hoặc EJB quản lý một phần tính đồng thời cho bạn. Các servlet được thực thi không đồng bộ bất cứ khi nào có yêu cầu Web. Người lập trình servlet không phải quản lý tất cả các yêu cầu đến. Về nguyên tắc, mỗi lần thực thi servlet tồn tại trong thế giới nhỏ bé của riêng nó và được tách rời khỏi tất cả các lần thực thi servlet khác.

Of course if it were that easy, this chapter wouldn’t be necessary. In fact, the decou-pling provided by Web containers is far less than perfect. Servlet programmers have to be very aware, and very careful, to make sure their concurrent programs are correct. Still, the structural beneﬁts of the servlet model are signiﬁcant.  
Tất nhiên nếu nó dễ dàng như vậy thì chương này sẽ không cần thiết. Trên thực tế, khả năng tách rời được cung cấp bởi các bộ chứa Web kém hoàn hảo hơn nhiều. Người lập trình servlet phải rất ý thức và rất cẩn thận để đảm bảo rằng các chương trình đồng thời của họ là chính xác. Tuy nhiên, những lợi ích cấu trúc của mô hình servlet là rất quan trọng.

But structure is not the only motive for adopting concurrency. Some systems have response time and throughput constraints that require hand-coded concurrent solutions. For example, consider a single-threaded information aggregator that acquires information from many different Web sites and merges that information into a daily summary. Because  
Nhưng cấu trúc không phải là động cơ duy nhất để áp dụng đồng thời. Một số hệ thống có các ràng buộc về thời gian phản hồi và thông lượng yêu cầu các giải pháp đồng thời được mã hóa bằng tay. Ví dụ: hãy xem xét một trình tổng hợp thông tin đơn luồng thu thập thông tin từ nhiều trang Web khác nhau và hợp nhất thông tin đó thành một bản tóm tắt hàng ngày. Bởi vì

<https://www.facebook.com/groups/phattrienlaptrinh>

**Why Concurrency?** 179  
Tại sao đồng thời? 179

this system is single threaded, it hits each Web site in turn, always ﬁnishing one before starting the next. The daily run needs to execute in less than 24 hours. However, as more and more Web sites are added, the time grows until it takes more than 24 hours to gather all the data. The single-thread involves a lot of waiting at Web sockets for I/O to complete. We could improve the performance by using a multithreaded algorithm that hits more than one Web site at a time.  
hệ thống này là một luồng đơn, nó truy cập lần lượt vào từng trang Web, luôn hoàn thành một trang trước khi bắt đầu trang tiếp theo. Quá trình chạy hàng ngày cần thực hiện trong vòng chưa đầy 24 giờ. Tuy nhiên, khi ngày càng có nhiều trang Web được thêm vào, thời gian sẽ tăng lên cho đến khi phải mất hơn 24 giờ để thu thập tất cả dữ liệu. Chuỗi đơn bao gồm rất nhiều thời gian chờ đợi tại các ổ cắm Web để I/O hoàn tất. Chúng tôi có thể cải thiện hiệu suất bằng cách sử dụng thuật toán đa luồng truy cập nhiều trang Web cùng một lúc.

Or consider a system that handles one user at a time and requires only one second of time per user. This system is fairly responsive for a few users, but as the number of users increases, the system’s response time increases. No user wants to get in line behind 150 others! We could improve the response time of this system by handling many users concurrently.  
Hoặc xem xét một hệ thống xử lý một người dùng tại một thời điểm và chỉ yêu cầu một giây thời gian cho mỗi người dùng. Hệ thống này khá nhạy đối với một số ít người dùng, nhưng khi số lượng người dùng tăng lên, thời gian phản hồi của hệ thống cũng tăng lên. Không người dùng nào muốn xếp hàng sau 150 người khác! Chúng tôi có thể cải thiện thời gian phản hồi của hệ thống này bằng cách xử lý đồng thời nhiều người dùng.

Or consider a system that interprets large data sets but can only give a complete solu-tion after processing all of them. Perhaps each data set could be processed on a different computer, so that many data sets are being processed in parallel.  
Hoặc xem xét một hệ thống diễn giải các tập dữ liệu lớn nhưng chỉ có thể đưa ra giải pháp hoàn chỉnh sau khi xử lý tất cả chúng. Có lẽ mỗi bộ dữ liệu có thể được xử lý trên một máy tính khác, do đó nhiều bộ dữ liệu đang được xử lý song song.

[**Myths and Misconceptions**](#_page_383_0)  
Thần thoại và quan niệm sai lầm

And so there are compelling reasons to adopt concurrency. However, as we said before, concurrency is *hard*. If you aren’t very careful, you can create some very nasty situations. Consider these common myths and misconceptions:  
Và do đó, có những lý do thuyết phục để áp dụng đồng thời. Tuy nhiên, như chúng tôi đã nói trước đây, đồng thời là khó khăn. Nếu bạn không cẩn thận, bạn có thể tạo ra một số tình huống rất khó chịu. Hãy xem xét những huyền thoại và quan niệm sai lầm phổ biến này:

• *Concurrency always improves performance.*  
• Đồng thời luôn cải thiện hiệu suất.

Concurrency can *sometimes* improve performance, but only when there is a lot of wait time that can be shared between multiple threads or multiple processors. Neither situ-ation is trivial.  
Đồng thời đôi khi có thể cải thiện hiệu suất, nhưng chỉ khi có nhiều thời gian chờ có thể được chia sẻ giữa nhiều luồng hoặc nhiều bộ xử lý. Cả hai tình huống đều không tầm thường.

• *Design does not change when writing concurrent programs.*  
• Thiết kế không thay đổi khi viết các chương trình đồng thời.

In fact, the design of a concurrent algorithm can be remarkably different from the design of a single-threaded system. The decoupling of *what* from *when* usually has a huge effect on the structure of the system.  
Trên thực tế, thiết kế của thuật toán đồng thời có thể khác biệt đáng kể so với thiết kế của hệ thống đơn luồng. Việc tách rời cái gì và cái gì thường có ảnh hưởng rất lớn đến cấu trúc của hệ thống.

• *Understanding concurrency issues is not important when working with a container such as a Web or EJB container.*  
• Việc hiểu các vấn đề tương tranh không quan trọng khi làm việc với một bộ chứa như bộ chứa Web hoặc EJB.

In fact, you’d better know just what your container is doing and how to guard against the issues of concurrent update and deadlock described later in this chapter.  
Trên thực tế, tốt hơn hết bạn nên biết vùng chứa của mình đang làm gì và cách bảo vệ chống lại các vấn đề về cập nhật đồng thời và bế tắc được mô tả sau trong chương này.

Here are a few more balanced sound bites regarding writing concurrent software:  
Dưới đây là một số âm thanh cân bằng hơn liên quan đến việc viết phần mềm đồng thời:

• *Concurrency incurs some overhead,* both in performance as well as writing additional code.  
• Đồng thời phát sinh một số chi phí, cả về hiệu suất cũng như viết mã bổ sung.

• *Correct concurrency is complex,* even for simple problems.  
• Đồng thời chính xác là phức tạp, ngay cả đối với các vấn đề đơn giản.

<https://www.facebook.com/groups/phattrienlaptrinh>

180 **Chapter 13: Concurrency**  
180 Chương 13: Đồng thời

• *Concurrency bugs aren’t usually repeatable,* so they are often ignored as one-offs2 instead of the true defects they are.  
• Các lỗi đồng thời thường không lặp lại, vì vậy chúng thường bị bỏ qua như một lần2 thay vì các lỗi thực sự của chúng.

• *Concurrency often requires a fundamental change in design strategy*.  
• Đồng thời thường đòi hỏi một sự thay đổi cơ bản trong chiến lược thiết kế.

[**Challenges**](#_page_459_0)  
thử thách

What makes concurrent programming so difﬁcult? Consider the following trivial class:  
Điều gì làm cho lập trình đồng thời trở nên khó khăn như vậy? Hãy xem xét lớp tầm thường sau:

public class X {

private int lastIdUsed;

public int getNextId() { return ++lastIdUsed;

} }

Let’s say we create an instance of X, set the lastIdUsedﬁeld to 42, and then share the instance between two threads. Now suppose that both of those threads call the method getNextId(); there are three possible outcomes:  
Giả sử chúng ta tạo một thể hiện của X, đặt lastIdUsedfield thành 42, sau đó chia sẻ thể hiện giữa hai luồng. Bây giờ, giả sử rằng cả hai luồng đó đều gọi phương thức getNextId(); có ba kết quả có thể xảy ra:

• Thread one gets the value 43, thread two gets the value 44, lastIdUsed is 44. • Thread one gets the value 44, thread two gets the value 43, lastIdUsed is 44. • Thread one gets the value 43, thread two gets the value 43, lastIdUsed is 43.  
• Luồng một nhận giá trị 43, luồng hai nhận giá trị 44, lastIdUsed là 44. • Luồng một nhận giá trị 44, luồng hai nhận giá trị 43, lastIdUsed là 44. • Luồng một nhận giá trị 43, luồng hai nhận giá trị giá trị 43, lastIdUsed là 43.

The surprising third result3 occurs when the two threads step on each other. This hap-pens because there are many possible paths that the two threads can take through that one line of Java code, and some of those paths generate incorrect results. How many different paths are there? To really answer that question, we need to understand what the Just-In-Time Compiler does with the generated byte-code, and understand what the Java memory model considers to be atomic.  
Kết quả đáng ngạc nhiên thứ ba3 xảy ra khi hai sợi dây giẫm lên nhau. Điều này xảy ra vì có nhiều đường dẫn khả thi mà hai luồng có thể đi qua một dòng mã Java đó và một số đường dẫn đó tạo ra kết quả không chính xác. Có bao nhiêu con đường khác nhau? Để thực sự trả lời câu hỏi đó, chúng ta cần hiểu Trình biên dịch Just-In-Time làm gì với mã byte được tạo và hiểu mô hình bộ nhớ Java coi là gì.

A quick answer, working with just the generated byte-code, is that there are 12,870 different possible execution paths4 for those two threads executing within the getNextId method. If the type of lastIdUsedis changed from intto long, the number of possible paths increases to 2,704,156. Of course most of those paths generate valid results. The problem is that *some of them don’t*.  
Một câu trả lời nhanh, chỉ hoạt động với mã byte được tạo, là có 12.870 đường dẫn thực thi khả thi khác nhau4 cho hai luồng thực thi đó trong phương thức getNextId. Nếu loại lastIdUsedis thay đổi từ intto long, số đường dẫn có thể tăng lên 2.704.156. Tất nhiên, hầu hết các đường dẫn đó tạo ra kết quả hợp lệ. Vấn đề là một số trong số họ không.

[**Concurrency Defense Principles**](#_page_459_0)  
Nguyên tắc bảo vệ đồng thời

What follows is a series of principles and techniques for defending your systems from the problems of concurrent code.  
Phần tiếp theo là một loạt các nguyên tắc và kỹ thuật để bảo vệ hệ thống của bạn khỏi các vấn đề về mã đồng thời.

2. Cosmic-rays, glitches, and so on.  
2. Tia vũ trụ, trục trặc, v.v.

3. See “Digging Deeper” on page 323.  
3. Xem phần “Đào sâu hơn” ở trang 323.

4. See “Possible Paths of Execution” on page 321.  
4. Xem “Các con đường thực thi khả thi” trên trang 321.

<https://www.facebook.com/groups/phattrienlaptrinh>

**Concurrency Defense Principles** 181  
Nguyên tắc bảo vệ tương tranh 181

[**Single Responsibility Principle**](#_page_459_0)  
Nguyên tắc trách nhiệm duy nhất

The SRP5 states that a given method/class/component should have a single reason to change. Concurrency design is complex enough to be a reason to change in it’s own right and therefore deserves to be separated from the rest of the code. Unfortunately, it is all too common for concurrency implementation details to be embedded directly into other pro-duction code. Here are a few things to consider:  
SRP5 tuyên bố rằng một phương thức/lớp/thành phần nhất định phải có một lý do duy nhất để thay đổi. Thiết kế đồng thời đủ phức tạp để trở thành một lý do để thay đổi theo đúng nghĩa của nó và do đó xứng đáng được tách biệt khỏi phần còn lại của mã. Thật không may, việc các chi tiết triển khai đồng thời được nhúng trực tiếp vào mã sản xuất khác là điều quá phổ biến. Đây là một vài thứ đáng xem xét:

• *Concurrency-related code has its own life cycle of development,* change, and tuning.  
• Mã liên quan đến đồng thời có vòng đời phát triển, thay đổi và điều chỉnh riêng.

• *Concurrency-related code has its own challenges,* which are different from and often more difﬁcult than nonconcurrency-related code.  
• Mã liên quan đến đồng thời có những thách thức riêng của nó, khác với và thường khó hơn mã không liên quan đến tương tranh.

• The number of ways in which miswritten concurrency-based code can fail makes it challenging enough without the added burden of surrounding application code.  
• Số cách mà mã dựa trên đồng thời bị viết sai có thể bị lỗi khiến nó đủ thách thức mà không phải chịu thêm gánh nặng của mã ứng dụng xung quanh.

**Recommendation**: *Keep your concurrency-related code separate from other code*.6  
Khuyến nghị: Giữ mã liên quan đến đồng thời của bạn tách biệt với mã khác.6

[**Corollary: Limit the Scope of Data**](#_page_459_0)  
Hệ quả: Giới hạn phạm vi dữ liệu

As we saw, two threads modifying the same ﬁeld of a shared object can interfere with each other, causing unexpected behavior. One solution is to use the synchronizedkeyword to protect a *critical section* in the code that uses the shared object. It is important to restrict the number of such critical sections. The more places shared data can get updated, the more likely:  
Như chúng ta đã thấy, hai luồng sửa đổi cùng một trường của một đối tượng được chia sẻ có thể can thiệp lẫn nhau, gây ra hành vi không mong muốn. Một giải pháp là sử dụng từ khóa được đồng bộ hóa để bảo vệ phần quan trọng trong mã sử dụng đối tượng được chia sẻ. Điều quan trọng là phải hạn chế số lượng các phần quan trọng như vậy. Càng nhiều nơi dữ liệu được chia sẻ có thể được cập nhật thì càng có nhiều khả năng:

• You will forget to protect one or more of those places—effectively breaking all code that modiﬁes that shared data.  
• Bạn sẽ quên bảo vệ một hoặc nhiều nơi trong số đó—phá vỡ hiệu quả tất cả các mã sửa đổi dữ liệu được chia sẻ đó.

• There will be duplication of effort required to make sure everything is effectively guarded (violation of DRY7).  
• Sẽ có sự trùng lặp về nỗ lực cần thiết để đảm bảo mọi thứ được bảo vệ hiệu quả (vi phạm DRY7).

• It will be difﬁcult to determine the source of failures, which are already hard enough to ﬁnd.  
• Sẽ rất khó để xác định nguồn gốc của sự cố, vốn đã khó tìm ra rồi.

**Recommendation**: *Take data encapsulation to heart; severely limit the access of any data that may be shared.*  
Khuyến nghị: Hãy quan tâm đến việc đóng gói dữ liệu; hạn chế nghiêm ngặt quyền truy cập của bất kỳ dữ liệu nào có thể được chia sẻ.

[**Corollary: Use Copies of Data**](#_page_459_0)  
Hệ quả: Sử dụng Bản sao Dữ liệu

A good way to avoid shared data is to avoid sharing the data in the ﬁrst place. In some sit-uations it is possible to copy objects and treat them as read-only. In other cases it might be possible to copy objects, collect results from multiple threads in these copies and then merge the results in a single thread.  
Một cách tốt để tránh chia sẻ dữ liệu là tránh chia sẻ dữ liệu ngay từ đầu. Trong một số tình huống, có thể sao chép các đối tượng và coi chúng là chỉ đọc. Trong các trường hợp khác, có thể sao chép các đối tượng, thu thập kết quả từ nhiều luồng trong các bản sao này và sau đó hợp nhất các kết quả trong một luồng.

5. [PPP]  
5. [PPP]

6. See “Client/Server Example” on page 317. 7. [PRAG].  
6. Xem “Ví dụ Máy khách/Máy chủ” trên trang 317. 7. [PRAG].

<https://www.facebook.com/groups/phattrienlaptrinh>

182 **Chapter 13: Concurrency**  
182 Chương 13: Đồng thời

If there is an easy way to avoid sharing objects, the resulting code will be far less likely to cause problems. You might be concerned about the cost of all the extra object creation. It is worth experimenting to ﬁnd out if this is in fact a problem. However, if using copies of objects allows the code to avoid synchronizing, the savings in avoiding the intrinsic lock will likely make up for the additional creation and garbage collection overhead.  
Nếu có một cách dễ dàng để tránh chia sẻ các đối tượng, thì mã kết quả sẽ ít gây ra sự cố hơn nhiều. Bạn có thể lo lắng về chi phí của tất cả việc tạo đối tượng bổ sung. Rất đáng để thử nghiệm để tìm hiểu xem đây có thực sự là một vấn đề hay không. Tuy nhiên, nếu việc sử dụng các bản sao của đối tượng cho phép mã tránh đồng bộ hóa, thì khoản tiết kiệm được trong việc tránh khóa nội tại sẽ có thể bù đắp cho chi phí tạo bổ sung và thu gom rác.

[**Corollary:Threads Should Be as Independent as Possible**](#_page_459_0)  
Hệ quả: Chủ đề nên càng độc lập càng tốt

Consider writing your threaded code such that each thread exists in its own world, sharing no data with any other thread. Each thread processes one client request, with all of its required data coming from an unshared source and stored as local variables. This makes each of those threads behave as if it were the only thread in the world and there were no synchronization requirements.  
Cân nhắc viết mã luồng của bạn sao cho mỗi luồng tồn tại trong thế giới riêng của nó, không chia sẻ dữ liệu với bất kỳ luồng nào khác. Mỗi luồng xử lý một yêu cầu của khách hàng, với tất cả dữ liệu cần thiết của nó đến từ một nguồn không chia sẻ và được lưu trữ dưới dạng các biến cục bộ. Điều này làm cho mỗi luồng đó hoạt động như thể nó là luồng duy nhất trên thế giới và không có yêu cầu đồng bộ hóa.

For example, classes that subclass from HttpServletreceive all of their information as parameters passed in to the doGetand doPostmethods. This makes each Servletact as if it has its own machine. So long as the code in the Servletuses only local variables, there is no chance that the Servletwill cause synchronization problems. Of course, most applications using Servletseventually run into shared resources such as database connections.  
Ví dụ: các lớp phân lớp từ HttpServletnhận tất cả thông tin của chúng dưới dạng các tham số được truyền vào các phương thức doGet và doPost. Điều này làm cho mỗi Servletact như thể nó có máy riêng. Miễn là mã trong Servlet chỉ sử dụng các biến cục bộ, không có khả năng Servlet sẽ gây ra sự cố đồng bộ hóa. Tất nhiên, hầu hết các ứng dụng sử dụng Servlets cuối cùng chạy vào các tài nguyên được chia sẻ, chẳng hạn như các kết nối cơ sở dữ liệu.

**Recommendation**: *Attempt to partition data into independent subsets than can be operated on by independent threads, possibly in different processors.*  
Khuyến nghị: Cố gắng phân vùng dữ liệu thành các tập hợp con độc lập mà các luồng độc lập có thể vận hành trên đó, có thể trong các bộ xử lý khác nhau.

[**Know Your Library**](#_page_459_0)  
Biết thư viện của bạn

Java 5 offers many improvements for concurrent development over previous versions. There are several things to consider when writing threaded code in Java 5:  
Java 5 cung cấp nhiều cải tiến để phát triển đồng thời so với các phiên bản trước. Có một số điều cần xem xét khi viết mã luồng trong Java 5:

• Use the provided thread-safe collections.  
• Sử dụng các bộ sưu tập thread-safe được cung cấp.

• Use the executor framework for executing unrelated tasks. • Use nonblocking solutions when possible.  
• Sử dụng khung thực thi để thực thi các tác vụ không liên quan. • Sử dụng các giải pháp không chặn khi có thể.

• Several library classes are not thread safe.  
• Một số lớp thư viện không phải là luồng an toàn.

[**Thread-Safe Collections**](#_page_459_0)  
Bộ sưu tập chủ đề an toàn

When Java was young, Doug Lea wrote the seminal book8 *Concurrent Programming in Java*. Along with the book he developed several thread-safe collections, which later became part of the JDK in the java.util.concurrentpackage. The collections in that pack-age are safe for multithreaded situations and they perform well. In fact, the  
Khi Java còn trẻ, Doug Lea đã viết cuốn sách nổi tiếng8 Lập trình đồng thời trong Java. Cùng với cuốn sách, ông đã phát triển một số bộ sưu tập thread-safe, sau này trở thành một phần của JDK trong gói java.util.concurrent. Các bộ sưu tập trong độ tuổi gói đó an toàn cho các tình huống đa luồng và chúng hoạt động tốt. Trên thực tế

8. [Lea99].

<https://www.facebook.com/groups/phattrienlaptrinh>

**Know Your Execution Models** 183  
Biết các mô hình thực thi của bạn 183

ConcurrentHashMapimplementation performs better than HashMapin nearly all situations. It also allows for simultaneous concurrent reads and writes, and it has methods supporting common composite operations that are otherwise not thread safe. If Java 5 is the deploy-ment environment, start with ConcurrentHashMap.  
ConcurrentHashMapimplementation hoạt động tốt hơn HashMapin trong hầu hết các trường hợp. Nó cũng cho phép đọc và ghi đồng thời đồng thời, và nó có các phương thức hỗ trợ các hoạt động tổng hợp phổ biến mà nếu không thì không an toàn cho luồng. Nếu Java 5 là môi trường triển khai, hãy bắt đầu với ConcurrentHashMap.

There are several other kinds of classes added to support advanced concurrency design. Here are a few examples:  
Có một số loại lớp khác được thêm vào để hỗ trợ thiết kế đồng thời nâng cao. Dưới đây là một vài ví dụ:

**ReentrantLock Semaphore  
ReentrantLock Semaphore**

**CountDownLatch  
Đếm NgượcChốt**

A lock that can be acquired in one method and released in another. An implementation of the classic semaphore, a lock with a count.  
Một khóa có thể được lấy theo một phương pháp và được giải phóng theo phương pháp khác. Một triển khai của semaphore cổ điển, một khóa có số đếm.

A lock that waits for a number of events before releasing all threads waiting on it. This allows all threads to have a fair chance of starting at about the same time.  
Khóa chờ một số sự kiện trước khi giải phóng tất cả các chuỗi đang chờ trên đó. Điều này cho phép tất cả các chuỗi có cơ hội bắt đầu gần như giống nhau.

**Recommendation**: *Review the classes available to you. In the case of Java, become familiar with java.util.concurrent, java.util.concurrent.atomic, java.util.concurrent.locks.*  
Khuyến nghị: Xem lại các lớp học có sẵn cho bạn. Trong trường hợp của Java, hãy làm quen với java.util.concurrent, java.util.concurrent.atomic, java.util.concurrent.locks.

[**Know Your Execution Models**](#_page_459_0)  
Biết các mô hình thực thi của bạn

There are several different ways to partition behavior in a concurrent application. To dis-cuss them we need to understand some basic deﬁnitions.  
Có một số cách khác nhau để phân vùng hành vi trong một ứng dụng đồng thời. Để thảo luận về chúng, chúng ta cần hiểu một số định nghĩa cơ bản.

Bound Resources  
tài nguyên ràng buộc

Mutual Exclusion  
Loại trừ lẫn nhau

Starvation  
chết đói

Deadlock  
Bế tắc

Livelock  
ổ khóa

Resources of a ﬁxed size or number used in a concurrent environ-ment. Examples include database connections and ﬁxed-size read/ write buffers.  
Tài nguyên có kích thước hoặc số lượng cố định được sử dụng trong môi trường đồng thời. Ví dụ bao gồm các kết nối cơ sở dữ liệu và bộ đệm đọc/ghi kích thước cố định.

Only one thread can access shared data or a shared resource at a time.  
Tại một thời điểm, chỉ một luồng có thể truy cập dữ liệu được chia sẻ hoặc tài nguyên được chia sẻ.

One thread or a group of threads is prohibited from proceeding for an excessively long time or forever. For example, always let-ting fast-running threads through ﬁrst could starve out longer run-ning threads if there is no end to the fast-running threads.  
Một luồng hoặc một nhóm luồng bị cấm tiếp tục trong một thời gian quá dài hoặc mãi mãi. Ví dụ: luôn cho phép các luồng chạy nhanh chạy qua đầu tiên có thể bỏ trống các luồng đang chạy dài hơn nếu không có kết thúc đối với các luồng chạy nhanh.

Two or more threads waiting for each other to ﬁnish. Each thread has a resource that the other thread requires and neither can ﬁnish until it gets the other resource.  
Hai hoặc nhiều luồng đang đợi nhau để kết thúc. Mỗi luồng có một tài nguyên mà luồng kia yêu cầu và không có luồng nào có thể kết thúc cho đến khi nó nhận được tài nguyên kia.

Threads in lockstep, each trying to do work but ﬁnding another “in the way.” Due to resonance, threads continue trying to make progress but are unable to for an excessively long time— or forever.  
Các chủ đề theo từng bước, mỗi chủ đề cố gắng thực hiện công việc nhưng lại tìm thấy một chủ đề khác “ngáng đường”. Do cộng hưởng, các luồng tiếp tục cố gắng đạt được tiến bộ nhưng không thể trong một thời gian quá dài— hoặc mãi mãi.

Given these deﬁnitions, we can now discuss the various execution models used in concurrent programming.  
Với những định nghĩa này, bây giờ chúng ta có thể thảo luận về các mô hình thực thi khác nhau được sử dụng trong lập trình đồng thời.

<https://www.facebook.com/groups/phattrienlaptrinh>

184 **Chapter 13: Concurrency**  
184 Chương 13: Đồng thời

[**Producer-Consumer9**](#_page_459_0)  
Nhà sản xuất-Người tiêu dùng9

One or more producer threads create some work and place it in a buffer or queue. One or more consumer threads acquire that work from the queue and complete it. The queue between the producers and consumers is a *bound resource*. This means producers must wait for free space in the queue before writing and consumers must wait until there is something in the queue to consume. Coordination between the producers and consumers via the queue involves producers and consumers signaling each other. The producers write to the queue and signal that the queue is no longer empty. Consumers read from the queue and signal that the queue is no longer full. Both potentially wait to be notiﬁed when they can continue.  
Một hoặc nhiều luồng của nhà sản xuất tạo một số công việc và đặt nó vào bộ đệm hoặc hàng đợi. Một hoặc nhiều chủ đề người tiêu dùng có được công việc đó từ hàng đợi và hoàn thành nó. Hàng đợi giữa nhà sản xuất và người tiêu dùng là một tài nguyên ràng buộc. Điều này có nghĩa là các nhà sản xuất phải chờ không gian trống trong hàng đợi trước khi viết và người tiêu dùng phải đợi cho đến khi có thứ gì đó trong hàng đợi để tiêu thụ. Sự phối hợp giữa người sản xuất và người tiêu dùng thông qua hàng đợi liên quan đến việc người sản xuất và người tiêu dùng báo hiệu cho nhau. Các nhà sản xuất ghi vào hàng đợi và báo hiệu rằng hàng đợi không còn trống nữa. Người tiêu dùng đọc từ hàng đợi và báo hiệu rằng hàng đợi không còn đầy nữa. Cả hai đều có khả năng chờ đợi để được thông báo khi họ có thể tiếp tục.

[**Readers-Writers10**](#_page_459_0)  
Độc giả-Nhà văn10

When you have a shared resource that primarily serves as a source of information for read-ers, but which is occasionally updated by writers, throughput is an issue. Emphasizing throughput can cause starvation and the accumulation of stale information. Allowing updates can impact throughput. Coordinating readers so they do not read something a writer is updating and vice versa is a tough balancing act. Writers tend to block many read-ers for a long period of time, thus causing throughput issues.  
Khi bạn có một tài nguyên được chia sẻ chủ yếu đóng vai trò là nguồn thông tin cho người đọc, nhưng thỉnh thoảng được người viết cập nhật, thông lượng sẽ là một vấn đề. Nhấn mạnh thông lượng có thể gây ra tình trạng đói và tích lũy thông tin cũ. Việc cho phép cập nhật có thể ảnh hưởng đến thông lượng. Điều phối người đọc để họ không đọc thứ gì đó mà người viết đang cập nhật và ngược lại là một hành động cân bằng khó khăn. Người viết có xu hướng chặn nhiều người đọc trong một thời gian dài, do đó gây ra các vấn đề về thông lượng.

The challenge is to balance the needs of both readers and writers to satisfy correct operation, provide reasonable throughput and avoiding starvation. A simple strategy makes writers wait until there are no readers before allowing the writer to perform an update. If there are continuous readers, however, the writers will be starved. On the other hand, if there are frequent writers and they are given priority, throughput will suffer. Find-ing that balance and avoiding concurrent update issues is what the problem addresses.  
Thách thức là cân bằng nhu cầu của cả người đọc và người viết để đáp ứng hoạt động chính xác, cung cấp thông lượng hợp lý và tránh chết đói. Một chiến lược đơn giản khiến người viết đợi cho đến khi không có người đọc trước khi cho phép người viết thực hiện cập nhật. Tuy nhiên, nếu có người đọc liên tục, người viết sẽ bị bỏ đói. Mặt khác, nếu có người viết thường xuyên và họ được ưu tiên, thông lượng sẽ bị ảnh hưởng. Tìm kiếm sự cân bằng đó và tránh các sự cố cập nhật đồng thời là vấn đề giải quyết.

[**Dining Philosophers11**](#_page_459_0)  
Nhà triết học ăn uống11

Imagine a number of philosophers sitting around a circular table. A fork is placed to the left of each philosopher. There is a big bowl of spaghetti in the center of the table. The philosophers spend their time thinking unless they get hungry. Once hungry, they pick up the forks on either side of them and eat. A philosopher cannot eat unless he is holding two forks. If the philosopher to his right or left is already using one of the forks he needs, he must wait until that philosopher ﬁnishes eating and puts the forks back down. Once a philosopher eats, he puts both his forks back down on the table and waits until he is hungry again.  
Hãy tưởng tượng một số triết gia ngồi quanh một chiếc bàn tròn. Một cái nĩa được đặt ở bên trái của mỗi triết gia. Có một bát mì spaghetti lớn ở giữa bàn. Các triết gia dành thời gian suy nghĩ trừ khi họ đói. Khi đói, họ cầm dĩa ở hai bên và ăn. Một triết gia không thể ăn trừ khi anh ta đang cầm hai cái nĩa. Nếu nhà triết học ở bên phải hoặc bên trái của anh ta đã sử dụng một trong những chiếc nĩa mà anh ta cần, thì anh ta phải đợi cho đến khi nhà triết học đó ăn xong và đặt nĩa trở lại. Khi một triết gia ăn xong, anh ta đặt cả hai chiếc nĩa của mình trở lại bàn và đợi cho đến khi anh ta đói trở lại.

Replace philosophers with threads and forks with resources and this problem is simi-lar to many enterprise applications in which processes compete for resources. Unless care-fully designed, systems that compete in this way can experience deadlock, livelock, throughput, and efﬁciency degradation.  
Thay thế các triết gia bằng các luồng và nhánh bằng tài nguyên và vấn đề này tương tự như nhiều ứng dụng doanh nghiệp trong đó các quy trình cạnh tranh để giành tài nguyên. Trừ khi được thiết kế cẩn thận, các hệ thống cạnh tranh theo cách này có thể bị bế tắc, bế tắc, thông lượng và suy giảm hiệu quả.

9. <http://en.wikipedia.org/wiki/Producer-consumer>

10. <http://en.wikipedia.org/wiki/Readers-writers_problem>

11. <http://en.wikipedia.org/wiki/Dining_philosophers_problem>

<https://www.facebook.com/groups/phattrienlaptrinh>

**Keep Synchronized Sections Small** 185  
Giữ các phần được đồng bộ hóa ở mức nhỏ 185

Most concurrent problems you will likely encounter will be some variation of these three problems. Study these algorithms and write solutions using them on your own so that when you come across concurrent problems, you’ll be more prepared to solve the problem.  
Hầu hết các vấn đề đồng thời mà bạn có thể gặp phải sẽ là một số biến thể của ba vấn đề này. Nghiên cứu các thuật toán này và tự viết các giải pháp bằng cách sử dụng chúng để khi gặp các vấn đề đồng thời, bạn sẽ sẵn sàng hơn để giải quyết vấn đề.

**Recommendation**: *Learn these basic algorithms and understand their solutions.*  
Khuyến nghị: Tìm hiểu các thuật toán cơ bản này và hiểu giải pháp của chúng.

[**Beware Dependencies Between Synchronized Methods**](#_page_459_0)  
Coi chừng sự phụ thuộc giữa các phương thức được đồng bộ hóa

Dependencies between synchronized methods cause subtle bugs in concurrent code. The Java language has the notion of synchronized, which protects an individual method. How-ever, if there is more than one synchronized method on the same shared class, then your system may be written incorrectly.12  
Sự phụ thuộc giữa các phương thức được đồng bộ hóa gây ra các lỗi tinh vi trong mã đồng thời. Ngôn ngữ Java có khái niệm đồng bộ hóa, bảo vệ một phương thức riêng lẻ. Tuy nhiên, nếu có nhiều hơn một phương thức được đồng bộ hóa trên cùng một lớp dùng chung, thì hệ thống của bạn có thể bị viết sai.12

**Recommendation**:*Avoid using more than one method on a shared object.*  
Khuyến nghị:Tránh sử dụng nhiều phương thức trên một đối tượng được chia sẻ.

There will be times when you must use more than one method on a shared object. When this is the case, there are three ways to make the code correct:  
Sẽ có lúc bạn phải sử dụng nhiều hơn một phương thức trên một đối tượng được chia sẻ. Trong trường hợp này, có ba cách để làm cho mã chính xác:

• **Client-Based Locking**—Have the client lock the server before calling the ﬁrst method and make sure the lock’s extent includes code calling the last method.  
• Khóa dựa trên máy khách—Yêu cầu máy khách khóa máy chủ trước khi gọi phương thức đầu tiên và đảm bảo phạm vi khóa bao gồm mã gọi phương thức cuối cùng.

• **Server-Based Locking**—Within the server create a method that locks the server, calls all the methods, and then unlocks. Have the client call the new method.  
• Khóa dựa trên máy chủ—Trong máy chủ, hãy tạo một phương thức khóa máy chủ, gọi tất cả các phương thức rồi mở khóa. Yêu cầu khách hàng gọi phương thức mới.

• **Adapted Server**—create an intermediary that performs the locking. This is an exam-ple of server-based locking, where the original server cannot be changed.  
• Máy chủ được điều chỉnh—tạo một trung gian thực hiện khóa. Đây là một ví dụ về khóa dựa trên máy chủ, trong đó máy chủ ban đầu không thể thay đổi.

[**Keep Synchronized Sections Small**](#_page_459_0)  
Giữ các phần được đồng bộ hóa ở mức nhỏ

The synchronized keyword introduces a lock. All sections of code guarded by the same lock are guaranteed to have only one thread executing through them at any given time. Locks are expensive because they create delays and add overhead. So we don’t want to litter our code with synchronizedstatements. On the other hand, critical sec-tions13 must be guarded. So we want to design our code with as few critical sections as possible.  
Từ khóa được đồng bộ hóa giới thiệu một khóa. Tất cả các phần mã được bảo vệ bởi cùng một khóa được đảm bảo chỉ có một luồng thực thi thông qua chúng tại bất kỳ thời điểm nào. Khóa đắt tiền vì chúng tạo ra sự chậm trễ và thêm chi phí. Vì vậy, chúng tôi không muốn xả rác mã của mình bằng các câu lệnh được đồng bộ hóa. Mặt khác, các phần quan trọng13 phải được bảo vệ. Vì vậy, chúng tôi muốn thiết kế mã của mình với càng ít phần quan trọng càng tốt.

Some naive programmers try to achieve this by making their critical sections very large. However, extending synchronization beyond the minimal critical section increases contention and degrades performance.14  
Một số lập trình viên ngây thơ cố gắng đạt được điều này bằng cách làm cho các phần quan trọng của họ trở nên rất lớn. Tuy nhiên, việc mở rộng đồng bộ hóa vượt quá phần quan trọng tối thiểu sẽ làm tăng sự tranh chấp và làm giảm hiệu suất.14

**Recommendation**: *Keep your synchronized sections as small as possible.*  
Khuyến nghị: Giữ các phần được đồng bộ hóa của bạn càng nhỏ càng tốt.

12. See “Dependencies Between Methods Can Break Concurrent Code” on page 329.  
12. Xem “Sự phụ thuộc giữa các phương thức có thể phá vỡ mã đồng thời” ở trang 329.

13. A critical section is any section of code that must be protected from simultaneous use for the program to be correct. 14. See “Increasing Throughput” on page 333.  
13. Phần quan trọng là bất kỳ phần mã nào phải được bảo vệ khỏi việc sử dụng đồng thời để chương trình hoạt động chính xác. 14. Xem “Tăng thông lượng” trên trang 333.

<https://www.facebook.com/groups/phattrienlaptrinh>

186 **Chapter 13: Concurrency**  
186 Chương 13: Đồng thời

[**Writing Correct Shut-Down Code Is Hard**](#_page_459_0)  
Viết đúng mã tắt là khó

Writing a system that is meant to stay live and run forever is different from writing some-thing that works for awhile and then shuts down gracefully.  
Viết một hệ thống có nghĩa là tồn tại và chạy mãi mãi khác với viết một cái gì đó hoạt động trong một thời gian và sau đó tắt một cách duyên dáng.

Graceful shutdown can be hard to get correct. Common problems involve deadlock,15 with threads waiting for a signal to continue that never comes.  
Tắt máy duyên dáng có thể khó để có được chính xác. Các sự cố phổ biến liên quan đến bế tắc,15 với các luồng đang chờ tín hiệu tiếp tục mà không bao giờ đến.

For example, imagine a system with a parent thread that spawns several child threads and then waits for them all to ﬁnish before it releases its resources and shuts down. What if one of the spawned threads is deadlocked? The parent will wait forever, and the system will never shut down.  
Ví dụ, hãy tưởng tượng một hệ thống với một luồng cha tạo ra một số luồng con và sau đó đợi tất cả chúng kết thúc trước khi giải phóng tài nguyên của nó và tắt. Điều gì sẽ xảy ra nếu một trong các luồng được sinh ra bị bế tắc? Cha mẹ sẽ đợi mãi mãi và hệ thống sẽ không bao giờ tắt.

Or consider a similar system that has been *instructed* to shut down. The parent tells all the spawned children to abandon their tasks and ﬁnish. But what if two of the children were operating as a producer/consumer pair. Suppose the producer receives the signal from the parent and quickly shuts down. The consumer might have been expecting a mes-sage from the producer and be blocked in a state where it cannot receive the shutdown sig-nal. It could get stuck waiting for the producer and never ﬁnish, preventing the parent from ﬁnishing as well.  
Hoặc xem xét một hệ thống tương tự đã được hướng dẫn tắt. Cha mẹ nói với tất cả những đứa trẻ được sinh ra từ bỏ nhiệm vụ của chúng và hoàn thành. Nhưng điều gì sẽ xảy ra nếu hai đứa trẻ hoạt động như một cặp nhà sản xuất/người tiêu dùng. Giả sử nhà sản xuất nhận được tín hiệu từ cha mẹ và nhanh chóng tắt. Người tiêu dùng có thể đã mong đợi một thông báo từ nhà sản xuất và bị chặn ở trạng thái không thể nhận được tín hiệu tắt máy. Nó có thể gặp khó khăn khi chờ đợi nhà sản xuất và không bao giờ hoàn thành, khiến cha mẹ cũng không thể hoàn thành.

Situations like this are not at all uncommon. So if you must write concurrent code that involves shutting down gracefully, expect to spend much of your time getting the shut-down to happen correctly.  
Những tình huống như thế này không phải là hiếm. Vì vậy, nếu bạn phải viết mã đồng thời liên quan đến việc tắt máy một cách duyên dáng, bạn sẽ phải dành nhiều thời gian để việc tắt máy diễn ra chính xác.

**Recommendation**:*Think about shut-down early and get it working early. It’s going to take longer than you expect. Review existing algorithms because this is probably harder than you think.*  
Khuyến nghị: Hãy suy nghĩ về việc tắt sớm và làm cho nó hoạt động sớm. Nó sẽ mất nhiều thời gian hơn bạn mong đợi. Xem lại các thuật toán hiện có vì điều này có thể khó hơn bạn nghĩ.

[**Testing Threaded Code**](#_page_459_0)  
Kiểm tra mã luồng

Proving that code is correct is impractical. Testing does not guarantee correctness. How-ever, good testing can minimize risk. This is all true in a single-threaded solution. As soon as there are two or more threads using the same code and working with shared data, things get substantially more complex.  
Chứng minh rằng mã đó là chính xác là không thực tế. Thử nghiệm không đảm bảo tính chính xác. Tuy nhiên, thử nghiệm tốt có thể giảm thiểu rủi ro. Điều này hoàn toàn đúng trong một giải pháp đơn luồng. Ngay khi có hai hoặc nhiều luồng sử dụng cùng một mã và làm việc với dữ liệu được chia sẻ, mọi thứ trở nên phức tạp hơn nhiều.

**Recommendation**: *Write tests that have the potential to expose problems and then run them frequently, with different programatic conﬁgurations and system conﬁgurations and load. If tests ever fail, track down the failure. Don’t ignore a failure just because the tests pass on a subsequent run.*  
Khuyến nghị: Viết các bài kiểm tra có khả năng phát hiện ra các vấn đề và sau đó chạy chúng thường xuyên, với các cấu hình chương trình và cấu hình hệ thống và tải khác nhau. Nếu các bài kiểm tra thất bại, hãy theo dõi lỗi. Đừng bỏ qua lỗi chỉ vì các bài kiểm tra vượt qua lần chạy tiếp theo.

That is a whole lot to take into consideration. Here are a few more ﬁne-grained recommendations:  
Đó là rất nhiều để xem xét. Dưới đây là một vài khuyến nghị chi tiết hơn:

• Treat spurious failures as candidate threading issues. • Get your nonthreaded code working ﬁrst.  
• Xử lý các lỗi giả mạo như các vấn đề phân luồng ứng viên. • Làm cho mã không phân luồng của bạn hoạt động trước.

15. See “Deadlock” on page 335.  
15. Xem “Bế tắc” ở trang 335.

<https://www.facebook.com/groups/phattrienlaptrinh>

**Testing Threaded Code** 187  
Kiểm tra Mã luồng 187

• Make your threaded code pluggable. • Make your threaded code tunable.  
• Làm cho mã luồng của bạn có thể cắm được. • Làm cho mã luồng của bạn có thể điều chỉnh được.

• Run with more threads than processors. • Run on different platforms.  
• Chạy với nhiều luồng hơn bộ xử lý. • Chạy trên các nền tảng khác nhau.

• Instrument your code to try and force failures.  
• Công cụ mã của bạn để thử và buộc thất bại.

[**Treat Spurious Failures as Candidate Threading Issues**](#_page_459_0)  
Xử lý các lỗi giả mạo như các vấn đề phân luồng của ứng viên

Threaded code causes things to fail that “simply cannot fail.” Most developers do not have an intuitive feel for how threading interacts with other code (authors included). Bugs in threaded code might exhibit their symptoms once in a thousand, or a million, executions. Attempts to repeat the systems can be frustratingly.This often leads developers to write off the failure as a cosmic ray, a hardware glitch, or some other kind of “one-off.” It is best to assume that one-offs do not exist. The longer these “one-offs” are ignored, the more code is built on top of a potentially faulty approach.  
Mã theo luồng khiến mọi thứ bị lỗi mà “đơn giản là không thể bị lỗi”. Hầu hết các nhà phát triển không có cảm giác trực quan về cách luồng tương tác với mã khác (bao gồm cả tác giả). Các lỗi trong mã luồng có thể biểu hiện các triệu chứng của chúng một lần trong một nghìn hoặc một triệu lần thực thi. Việc cố gắng lặp lại các hệ thống có thể gây khó chịu. Điều này thường khiến các nhà phát triển coi lỗi là do tia vũ trụ, trục trặc phần cứng hoặc một số loại "chỉ xảy ra một lần" khác. Tốt nhất là giả định rằng một lần không tồn tại. Những “lần xuất hiện một lần” này bị bỏ qua càng lâu thì càng có nhiều mã được xây dựng dựa trên cách tiếp cận có khả năng bị lỗi.

**Recommendation**: *Do not ignore system failures as one-offs.*  
Khuyến nghị: Đừng bỏ qua lỗi hệ thống chỉ xảy ra một lần.

[**Get Your Nonthreaded Code Working First**](#_page_459_0)  
Làm cho mã không theo luồng của bạn hoạt động trước

This may seem obvious, but it doesn’t hurt to reinforce it. Make sure code works outside of its use in threads. Generally, this means creating POJOs that are called by your threads. The POJOs are not thread aware, and can therefore be tested outside of the threaded envi-ronment. The more of your system you can place in such POJOs, the better.  
Điều này có vẻ hiển nhiên, nhưng củng cố nó cũng không hại gì. Hãy chắc chắn rằng mã hoạt động bên ngoài việc sử dụng nó trong chủ đề. Nói chung, điều này có nghĩa là tạo các POJO được gọi bởi chủ đề của bạn. Các POJO không nhận biết luồng và do đó có thể được kiểm tra bên ngoài môi trường luồng. Bạn có thể đặt càng nhiều hệ thống của mình vào các POJO như vậy thì càng tốt.

**Recommendation**: *Do not try to chase down nonthreading bugs and threading bugs at the same time. Make sure your code works outside of threads*.  
Khuyến nghị: Đừng cố gắng đuổi theo các lỗi không phân luồng và các lỗi phân luồng cùng một lúc. Đảm bảo mã của bạn hoạt động bên ngoài luồng.

[**Make Your Threaded Code Pluggable**](#_page_459_0)  
Làm cho mã luồng của bạn có thể cắm được

Write the concurrency-supporting code such that it can be run in several conﬁgurations:  
Viết mã hỗ trợ đồng thời sao cho nó có thể chạy trong một số cấu hình:

• One thread, several threads, varied as it executes  
• Một luồng, nhiều luồng, thay đổi khi nó thực thi

• Threaded code interacts with something that can be both real or a test double. • Execute with test doubles that run quickly, slowly, variable.  
• Mã luồng tương tác với một cái gì đó có thể là thực hoặc thử nghiệm kép. • Thực hiện với các bài thi chạy nhanh, chậm, biến thiên.

• Conﬁgure tests so they can run for a number of iterations.  
• Định cấu hình các bài kiểm tra để chúng có thể chạy trong một số lần lặp lại.

**Recommendation**: *Make your thread-based code especially pluggable so that you can run it in various conﬁgurations.*  
Khuyến nghị: Làm cho mã dựa trên luồng của bạn đặc biệt có thể cắm được để bạn có thể chạy mã đó trong các cấu hình khác nhau.

[**Make Your Threaded Code Tunable**](#_page_459_0)  
Làm cho mã luồng của bạn có thể điều chỉnh được

Getting the right balance of threads typically requires trial an error. Early on, ﬁnd ways to time the performance of your system under different conﬁgurations. Allow the number of  
Để có được sự cân bằng phù hợp của các luồng thường yêu cầu thử lỗi. Ngay từ đầu, hãy tìm cách tính thời gian hiệu suất của hệ thống của bạn theo các cấu hình khác nhau. Cho phép số lượng

<https://www.facebook.com/groups/phattrienlaptrinh>

188 **Chapter 13: Concurrency**  
188 Chương 13: Đồng thời

threads to be easily tuned. Consider allowing it to change while the system is running. Consider allowing self-tuning based on throughput and system utilization.  
chủ đề để được dễ dàng điều chỉnh. Cân nhắc cho phép nó thay đổi trong khi hệ thống đang chạy. Cân nhắc cho phép tự điều chỉnh dựa trên thông lượng và mức sử dụng hệ thống.

[**Run with More Threads Than Processors**](#_page_459_0)  
Chạy với nhiều luồng hơn bộ xử lý

Things happen when the system switches between tasks. To encourage task swapping, run with more threads than processors or cores. The more frequently your tasks swap, the more likely you’ll encounter code that is missing a critical section or causes deadlock.  
Mọi thứ xảy ra khi hệ thống chuyển đổi giữa các tác vụ. Để khuyến khích hoán đổi tác vụ, hãy chạy với nhiều luồng hơn bộ xử lý hoặc lõi. Nhiệm vụ của bạn hoán đổi càng thường xuyên, bạn càng có nhiều khả năng gặp phải mã bị thiếu phần quan trọng hoặc gây ra bế tắc.

[**Run on Different Platforms**](#_page_459_0)  
Chạy trên các nền tảng khác nhau

In the middle of 2007 we developed a course on concurrent programming. The course development ensued primarily under OS X. The class was presented using Windows XP running under a VM. Tests written to demonstrate failure conditions did not fail as fre-quently in an XP environment as they did running on OS X.  
Vào giữa năm 2007, chúng tôi đã phát triển một khóa học về lập trình đồng thời. Quá trình phát triển khóa học diễn ra chủ yếu dưới OS X. Lớp học được trình bày bằng Windows XP chạy trong máy ảo. Các thử nghiệm được viết để chứng minh các điều kiện lỗi không thường xuyên xảy ra lỗi trong môi trường XP như khi chúng chạy trên OS X.

In all cases the code under test was known to be incorrect. This just reinforced the fact that different operating systems have different threading policies, each of which impacts the code’s execution. Multithreaded code behaves differently in different environments.16 You should run your tests in every potential deployment environment.  
Trong mọi trường hợp, mã được kiểm tra được biết là không chính xác. Điều này chỉ củng cố thực tế là các hệ điều hành khác nhau có các chính sách phân luồng khác nhau, mỗi chính sách đều ảnh hưởng đến việc thực thi mã. Mã đa luồng hoạt động khác nhau trong các môi trường khác nhau.16 Bạn nên chạy thử nghiệm của mình trong mọi môi trường triển khai tiềm năng.

**Recommendation**: *Run your threaded code on all target platforms early and often.*  
Đề xuất: Chạy mã luồng của bạn trên tất cả các nền tảng mục tiêu sớm và thường xuyên.

[**Instrument Your Code to Try and Force Failures**](#_page_459_0)  
Cung cấp mã của bạn để thử và buộc thất bại

It is normal for ﬂaws in concurrent code to hide. Simple tests often don’t expose them. Indeed, they often hide during normal processing. They might show up once every few hours, or days, or weeks!  
Việc ẩn các lỗi trong mã đồng thời là điều bình thường. Các bài kiểm tra đơn giản thường không phơi bày chúng. Thật vậy, chúng thường ẩn trong quá trình xử lý bình thường. Chúng có thể xuất hiện cứ sau vài giờ, vài ngày hoặc vài tuần!

The reason that threading bugs can be infrequent, sporadic, and hard to repeat, is that only a very few pathways out of the many thousands of possible pathways through a vul-nerable section actually fail. So the probability that a failing pathway is taken can be star-tlingly low. This makes detection and debugging very difﬁcult.  
Lý do mà các lỗi phân luồng có thể không thường xuyên, rời rạc và khó lặp lại là do chỉ có một số rất ít đường dẫn trong số hàng ngàn đường dẫn có thể có thông qua một phần dễ bị tổn thương thực sự thất bại. Vì vậy, xác suất để thực hiện một lộ trình thất bại có thể thấp đến kinh ngạc. Điều này làm cho việc phát hiện và sửa lỗi rất khó khăn.

How might you increase your chances of catching such rare occurrences? You can instrument your code and force it to run in different orderings by adding calls to methods like Object.wait(), Object.sleep(), Object.yield() and Object.priority().  
Làm thế nào bạn có thể tăng cơ hội bắt gặp những sự cố hiếm gặp như vậy? Bạn có thể cung cấp công cụ cho mã của mình và buộc mã chạy theo các thứ tự khác nhau bằng cách thêm lệnh gọi vào các phương thức như Object.wait(), Object.sleep(), Object.yield() và Object.priority().

Each of these methods can affect the order of execution, thereby increasing the odds of detecting a ﬂaw. It’s better when broken code fails as early and as often as possible.  
Mỗi phương pháp này có thể ảnh hưởng đến thứ tự thực hiện, do đó làm tăng khả năng phát hiện ra lỗ hổng. Sẽ tốt hơn nếu mã bị lỗi bị lỗi càng sớm và càng thường xuyên càng tốt.

There are two options for code instrumentation:  
Có hai tùy chọn cho thiết bị đo mã:

• Hand-coded • Automated  
• Mã hóa thủ công • Tự động

16. Did you know that the threading model in Java does not guarantee preemptive threading? Modern OS’s support preemptive threading, so you get that “for free.” Even so, it not guaranteed by the JVM.  
16. Bạn có biết rằng mô hình phân luồng trong Java không đảm bảo phân luồng ưu tiên? Hệ điều hành hiện đại hỗ trợ phân luồng ưu tiên, vì vậy bạn nhận được “miễn phí”. Mặc dù vậy, nó không được đảm bảo bởi JVM.

<https://www.facebook.com/groups/phattrienlaptrinh>

**Testing Threaded Code** 189  
Kiểm tra Mã luồng 189

[**Hand-Coded**](#_page_459_0)  
mã hóa bằng tay

You can insert calls to wait(), sleep(), yield(), and priority()in your code by hand. It might be just the thing to do when you’re testing a particularly thorny piece of code.  
Bạn có thể chèn lệnh gọi wait(), sleep(), yield() và priority() vào mã của mình bằng tay. Nó có thể chỉ là điều cần làm khi bạn đang thử nghiệm một đoạn mã đặc biệt hóc búa.

Here is an example of doing just that:  
Đây là một ví dụ về việc làm đó:

public synchronized String nextUrlOrNull() { if(hasNext()) {

String url = urlGenerator.next(); Thread.yield(); // inserted for testing. updateHasNext();

return url; }

return null; }

The inserted call to yield()will change the execution pathways taken by the code and possibly cause the code to fail where it did not fail before. If the code does break, it was not because you added a call to yield().17 Rather, your code was broken and this simply made the failure evident.  
Lệnh gọi hàm yield() được chèn vào sẽ thay đổi lộ trình thực thi của mã và có thể khiến mã bị lỗi ở vị trí trước đây nó không bị lỗi. Nếu mã bị hỏng, thì đó không phải là do bạn đã thêm lệnh gọi đến hàm yield().17 Thay vào đó, mã của bạn đã bị hỏng và điều này chỉ đơn giản là làm cho lỗi trở nên rõ ràng.

There are many problems with this approach:  
Có nhiều vấn đề với cách tiếp cận này:

• You have to manually ﬁnd appropriate places to do this.  
• Bạn phải tự tìm những nơi thích hợp để làm việc này.

• How do you know where to put the call and what kind of call to use?  
• Làm thế nào để bạn biết nên thực hiện cuộc gọi ở đâu và nên sử dụng loại cuộc gọi nào?

• Leaving such code in a production environment unnecessarily slows the code down.  
• Để mã như vậy trong môi trường sản xuất sẽ làm chậm mã một cách không cần thiết.

• It’s a shotgun approach. You may or may not ﬁnd ﬂaws. Indeed, the odds aren’t with you.  
• Đó là một cách tiếp cận súng ngắn. Bạn có thể hoặc không thể tìm thấy những sai sót. Thật vậy, tỷ lệ cược không phải với bạn.

What we need is a way to do this during testing but not in production. We also need to easily mix up conﬁgurations between different runs, which results in increased chances of ﬁnding errors in the aggregate.  
Những gì chúng tôi cần là một cách để làm điều này trong quá trình thử nghiệm chứ không phải trong quá trình sản xuất. Chúng ta cũng cần dễ dàng kết hợp các cấu hình giữa các lần chạy khác nhau, điều này dẫn đến tăng khả năng tìm thấy lỗi trong tổng hợp.

Clearly, if we divide our system up into POJOs that know nothing of threading and classes that control the threading, it will be easier to ﬁnd appropriate places to instrument the code. Moreover, we could create many different test jigs that invoke the POJOs under different regimes of calls to sleep, yield, and so on.  
Rõ ràng, nếu chúng ta chia hệ thống của mình thành các POJO không biết gì về phân luồng và các lớp kiểm soát phân luồng, thì sẽ dễ dàng hơn để tìm các vị trí thích hợp để tạo công cụ mã. Hơn nữa, chúng tôi có thể tạo nhiều đồ gá thử nghiệm khác nhau để gọi POJO theo các chế độ lệnh gọi ngủ, năng suất khác nhau, v.v.

[**Automated**](#_page_459_0)  
tự động

You could use tools like an Aspect-Oriented Framework, CGLIB, or ASM to programmat-ically instrument your code. For example, you could use a class with a single method:  
Bạn có thể sử dụng các công cụ như Khung hướng theo khía cạnh, CGLIB hoặc ASM để lập trình mã của bạn. Ví dụ: bạn có thể sử dụng một lớp với một phương thức duy nhất:

public class ThreadJigglePoint { public static void jiggle() { }  
lớp công khai ThreadJigglePoint { public static void jiggle() { }

}

17. This is not strictly the case. Since the JVM does not guarantee preemptive threading, a particular algorithm might always work on an OS that does not preempt threads. The reverse is also possible but for different reasons.  
17. Điều này không hoàn toàn đúng. Vì JVM không đảm bảo phân luồng ưu tiên, nên một thuật toán cụ thể có thể luôn hoạt động trên một hệ điều hành không ưu tiên luồng. Điều ngược lại cũng có thể xảy ra nhưng vì những lý do khác nhau.

<https://www.facebook.com/groups/phattrienlaptrinh>

190 **Chapter 13: Concurrency**  
190 Chương 13: Đồng thời

You can add calls to this in various places within your code:  
Bạn có thể thêm các cuộc gọi này ở những nơi khác nhau trong mã của mình:

public synchronized String nextUrlOrNull() { if(hasNext()) {

ThreadJiglePoint.jiggle(); String url = urlGenerator.next(); ThreadJiglePoint.jiggle(); updateHasNext(); ThreadJiglePoint.jiggle(); return url;

}

return null; }

Now you use a simple aspect that randomly selects among doing nothing, sleeping, or yielding.  
Giờ đây, bạn sử dụng một khía cạnh đơn giản chọn ngẫu nhiên trong số không làm gì, ngủ hoặc đầu hàng.

Or imagine that the ThreadJigglePointclass has two implementations. The ﬁrst imple-ments jiggleto do nothing and is used in production. The second generates a random number to choose between sleeping, yielding, or just falling through. If you run your tests a thousand times with random jiggling, you may root out some ﬂaws. If the tests pass, at least you can say you’ve done due diligence. Though a bit simplistic, this could be a rea-sonable option in lieu of a more sophisticated tool.  
Hoặc tưởng tượng rằng lớp ThreadJigglePoint có hai cách triển khai. Thiết bị đầu tiên jiggle không làm gì cả và được sử dụng trong sản xuất. Thứ hai tạo ra một số ngẫu nhiên để lựa chọn giữa ngủ, năng suất hoặc chỉ vượt qua. Nếu bạn chạy thử nghiệm hàng nghìn lần với những tiếng lắc lư ngẫu nhiên, bạn có thể tìm ra một số sai sót. Nếu các bài kiểm tra vượt qua, ít nhất bạn có thể nói rằng bạn đã thực hiện thẩm định. Mặc dù hơi đơn giản, nhưng đây có thể là một lựa chọn hợp lý thay cho một công cụ phức tạp hơn.

There is a tool called ConTest,18 developed by IBM that does something similar, but it does so with quite a bit more sophistication.  
Có một công cụ gọi là ConTest,18 do IBM phát triển có chức năng tương tự nhưng tinh vi hơn một chút.

The point is to jiggle the code so that threads run in different orderings at different times. The combination of well-written tests and jiggling can dramatically increase the chance ﬁnding errors.  
Vấn đề là lắc mã để các luồng chạy theo các thứ tự khác nhau vào các thời điểm khác nhau. Sự kết hợp giữa các bài kiểm tra được viết tốt và trò đùa có thể làm tăng đáng kể cơ hội tìm ra lỗi.

**Recommendation**: *Use jiggling strategies to ferret out errors.*  
Khuyến nghị: Sử dụng các chiến lược lắc lư để tìm ra lỗi.

[**Conclusion**](#_page_459_0)  
Phần kết luận

Concurrent code is difﬁcult to get right. Code that is simple to follow can become night-marish when multiple threads and shared data get into the mix. If you are faced with writ-ing concurrent code, you need to write clean code with rigor or else face subtle and infrequent failures.  
Mã đồng thời là khó khăn để có được đúng. Mã đơn giản để làm theo có thể trở nên khó hiểu khi nhiều luồng và dữ liệu được chia sẻ trộn lẫn với nhau. Nếu bạn phải đối mặt với việc viết mã đồng thời, bạn cần phải viết mã rõ ràng một cách chặt chẽ, nếu không sẽ gặp phải những lỗi nhỏ và không thường xuyên.

First and foremost, follow the Single Responsibility Principle. Break your system into POJOs that separate thread-aware code from thread-ignorant code. Make sure when you are testing your thread-aware code, you are only testing it and nothing else. This suggests that your thread-aware code should be small and focused.  
Đầu tiên và quan trọng nhất, hãy tuân theo Nguyên tắc Trách nhiệm duy nhất. Chia hệ thống của bạn thành các POJO tách mã nhận biết luồng khỏi mã không biết luồng. Đảm bảo rằng khi bạn đang kiểm tra mã nhận biết luồng của mình, bạn chỉ đang kiểm tra mã đó và không có gì khác. Điều này cho thấy rằng mã nhận biết luồng của bạn phải nhỏ và tập trung.

Know the possible sources of concurrency issues: multiple threads operating on shared data, or using a common resource pool. Boundary cases, such as shutting down cleanly or ﬁnishing the iteration of a loop, can be especially thorny.  
Biết các nguồn có thể xảy ra sự cố đồng thời: nhiều luồng hoạt động trên dữ liệu được chia sẻ hoặc sử dụng nhóm tài nguyên chung. Các trường hợp ranh giới, chẳng hạn như tắt hoàn toàn hoặc hoàn thành việc lặp lại một vòng lặp, có thể đặc biệt khó khăn.

18. <http://www.alphaworks.ibm.com/tech/contest>

<https://www.facebook.com/groups/phattrienlaptrinh>

**Bibliography** 191  
Thư mục 191

Learn your library and know the fundamental algorithms. Understand how some of the features offered by the library support solving problems similar to the fundamental algorithms.  
Tìm hiểu thư viện của bạn và biết các thuật toán cơ bản. Hiểu cách một số tính năng do thư viện cung cấp hỗ trợ giải các bài toán tương tự như các thuật toán cơ bản.

Learn how to ﬁnd regions of code that must be locked and lock them. Do not lock regions of code that do not need to be locked. Avoid calling one locked section from another. This requires a deep understanding of whether something is or is not shared. Keep the amount of shared objects and the scope of the sharing as narrow as possible. Change designs of the objects with shared data to accommodate clients rather than forcing clients to manage shared state.  
Tìm hiểu cách tìm các vùng mã phải được khóa và khóa chúng. Không khóa các vùng mã không cần khóa. Tránh gọi một phần bị khóa từ một phần khác. Điều này đòi hỏi một sự hiểu biết sâu sắc về việc một cái gì đó được chia sẻ hay không. Giữ số lượng đối tượng được chia sẻ và phạm vi chia sẻ càng hẹp càng tốt. Thay đổi thiết kế của các đối tượng có dữ liệu được chia sẻ để phù hợp với máy khách thay vì buộc máy khách phải quản lý trạng thái được chia sẻ.

Issues will crop up. The ones that do not crop up early are often written off as a one-time occurrence. These so-called one-offs typically only happen under load or at seem-ingly random times. Therefore, you need to be able to run your thread-related code in many conﬁgurations on many platforms repeatedly and continuously. Testability, which comes naturally from following the Three Laws of TDD, implies some level of plug-ability, which offers the support necessary to run code in a wider range of conﬁgurations.  
Các vấn đề sẽ tăng lên. Những thứ không mọc lên sớm thường được coi là chỉ xảy ra một lần. Cái gọi là một lần này thường chỉ xảy ra khi tải hoặc vào những thời điểm dường như ngẫu nhiên. Do đó, bạn cần có khả năng chạy mã liên quan đến chuỗi của mình trong nhiều cấu hình trên nhiều nền tảng lặp đi lặp lại và liên tục. Khả năng kiểm tra, xuất phát tự nhiên từ việc tuân theo Ba định luật của TDD, bao hàm một số mức độ khả năng cắm, cung cấp sự hỗ trợ cần thiết để chạy mã trong phạm vi cấu hình rộng hơn.

You will greatly improve your chances of ﬁnding erroneous code if you take the time to instrument your code. You can either do so by hand or using some kind of automated technology. Invest in this early.You want to be running your thread-based code as long as possible before you put it into production.  
Bạn sẽ cải thiện đáng kể cơ hội tìm ra mã sai nếu bạn dành thời gian để chỉnh sửa mã của mình. Bạn có thể làm như vậy bằng tay hoặc sử dụng một số loại công nghệ tự động. Hãy đầu tư vào điều này sớm. Bạn muốn chạy mã dựa trên chuỗi của mình càng lâu càng tốt trước khi đưa mã vào sản xuất.

If you take a clean approach, your chances of getting it right increase drastically.  
Nếu bạn thực hiện một cách tiếp cận rõ ràng, cơ hội của bạn để làm cho đúng sẽ tăng lên đáng kể.

[**Bibliography**](#_page_459_0)  
Thư mục

**[Lea99]:** *Concurrent Programming in Java: Design Principles and Patterns*, 2d. ed., Doug Lea, Prentice Hall, 1999.

**[PPP]:** *Agile Software Development: Principles, Patterns, and Practices*, Robert C. Martin, Prentice Hall, 2002.

**[PRAG]:** *The Pragmatic Programmer*, Andrew Hunt, Dave Thomas, Addison-Wesley, 2000.

<https://www.facebook.com/groups/phattrienlaptrinh>

*This page intentionally left blank*

<https://www.facebook.com/groups/phattrienlaptrinh>

[**14**](#_page_459_0)

[**Successive Reﬁnement**](#_page_459_0)  
Cải tiến liên tiếp

Case Study of a Command-Line Argument Parser  
Nghiên cứu điển hình về trình phân tích cú pháp đối số dòng lệnh

This chapter is a case study in successive reﬁnement. You will see a module that started well but did not scale. Then you will see how the module was refactored and cleaned.  
Chương này là một nghiên cứu trường hợp trong sự sàng lọc liên tiếp. Bạn sẽ thấy một mô-đun bắt đầu tốt nhưng không mở rộng quy mô. Sau đó, bạn sẽ thấy mô-đun đã được tái cấu trúc và làm sạch như thế nào.

Most of us have had to parse command-line arguments from time to time. If we don’t have a convenient utility, then we simply walk the array of strings that is passed into the mainfunction. There are several good utilities available from various sources,  
Hầu hết chúng ta thỉnh thoảng phải phân tích các đối số dòng lệnh. Nếu chúng ta không có một tiện ích thuận tiện, thì chúng ta chỉ cần di chuyển mảng chuỗi được truyền vào hàm chính. Có một số tiện ích tốt có sẵn từ nhiều nguồn khác nhau,

193

<https://www.facebook.com/groups/phattrienlaptrinh>

194 **Chapter 14: Successive Reﬁnement**  
194 Chương 14: Tinh luyện liên tiếp

but none of them do exactly what I want. So, of course, I decided to write my own. I call it: Args.  
nhưng không ai trong số họ làm chính xác những gì tôi muốn. Vì vậy, tất nhiên, tôi quyết định viết của riêng mình. Tôi gọi nó là: Args.

Argsis very simple to use. You simply construct the Argsclass with the input argu-ments and a format string, and then query the Argsinstance for the values of the argu-ments. Consider the following simple example:  
Argsis rất đơn giản để sử dụng. Bạn chỉ cần xây dựng Lớp đối số với các đối số đầu vào và một chuỗi định dạng, sau đó truy vấn Trường đối số để biết giá trị của các đối số. Xét ví dụ đơn giản sau:

**Listing 14-1**

**Simple use of Args**

public static void main(String[] args) { try {

Args arg = new Args("l,p#,d\*", args); boolean logging = arg.getBoolean('l'); int port = arg.getInt('p');

String directory = arg.getString('d'); executeApplication(logging, port, directory);

} catch (ArgsException e) {

System.out.printf("Argument error: %s\n", e.errorMessage()); }

}

You can see how simple this is. We just create an instance of the Argsclass with two parameters. The ﬁrst parameter is the format, or *schema,* string: "l,p#,d\*." It deﬁnes three command-line arguments. The ﬁrst, –l, is a boolean argument. The second, -p, is an integer argument. The third, -d, is a string argument. The second parameter to the Argsconstructor is simply the array of command-line argument passed into main.  
Bạn có thể thấy điều này đơn giản như thế nào. Chúng tôi chỉ tạo một thể hiện của Argsclass với hai tham số. Tham số đầu tiên là định dạng hoặc lược đồ, chuỗi: "l,p#,d\*." Nó định nghĩa ba đối số dòng lệnh. Đầu tiên, –l, là một đối số boolean. Thứ hai, -p, là một đối số nguyên. Thứ ba, -d, là một đối số chuỗi. Tham số thứ hai cho Argsconstructor chỉ đơn giản là mảng đối số dòng lệnh được truyền vào chính.

If the constructor returns without throwing an ArgsException, then the incoming command-line was parsed, and the Argsinstance is ready to be queried. Methods like getBoolean, getInteger, and getStringallow us to access the values of the arguments by their names.  
Nếu hàm tạo trả về mà không ném ArgsException, thì dòng lệnh đến đã được phân tích cú pháp và Argsinstance đã sẵn sàng để được truy vấn. Các phương thức như getBoolean, getInteger và getString cho phép chúng ta truy cập các giá trị của đối số theo tên của chúng.

If there is a problem, either in the format string or in the command-line arguments themselves, an ArgsException will be thrown. A convenient description of what went wrong can be retrieved from the errorMessage method of the exception.  
Nếu có vấn đề, trong chuỗi định dạng hoặc trong chính các đối số dòng lệnh, một ArgsException sẽ được đưa ra. Một mô tả thuận tiện về những gì đã sai có thể được lấy từ phương thức errorMessage của ngoại lệ.

[**Args Implementation**](#_page_459_0)  
Thực hiện đối số

Listing 14-2 is the implementation of the Argsclass. Please read it very carefully. I worked hard on the style and structure and hope it is worth emulating.  
Liệt kê 14-2 là triển khai của Argsclass. Xin vui lòng đọc nó rất cẩn thận. Tôi đã làm việc chăm chỉ về phong cách và cấu trúc và hy vọng nó đáng để mô phỏng.

**Listing 14-2 Args.java  
Liệt kê 14-2 Args.java**

package com.objectmentor.utilities.args;

import static com.objectmentor.utilities.args.ArgsException.ErrorCode.\*; import java.util.\*;

public class Args {

private Map<Character, ArgumentMarshaler> marshalers;

<https://www.facebook.com/groups/phattrienlaptrinh>

**Args Implementation** 195  
Args thực hiện 195

**Listing 14-2 (continued) Args.java**  
Liệt kê 14-2 (tiếp theo) Args.java

private Set<Character> argsFound;

private ListIterator<String> currentArgument;

public Args(String schema, String[] args) throws ArgsException { marshalers = new HashMap<Character, ArgumentMarshaler>(); argsFound = new HashSet<Character>();

parseSchema(schema); parseArgumentStrings(Arrays.asList(args));

}

private void parseSchema(String schema) throws ArgsException { for (String element : schema.split(","))

if (element.length() > 0) parseSchemaElement(element.trim());

}

private void parseSchemaElement(String element) throws ArgsException { char elementId = element.charAt(0);

String elementTail = element.substring(1); validateSchemaElementId(elementId);

if (elementTail.length() == 0)

marshalers.put(elementId, new BooleanArgumentMarshaler()); else if (elementTail.equals("\*"))

marshalers.put(elementId, new StringArgumentMarshaler()); else if (elementTail.equals("#"))

marshalers.put(elementId, new IntegerArgumentMarshaler()); else if (elementTail.equals("##"))

marshalers.put(elementId, new DoubleArgumentMarshaler()); else if (elementTail.equals("[\*]"))

marshalers.put(elementId, new StringArrayArgumentMarshaler()); else

throw new ArgsException(INVALID\_ARGUMENT\_FORMAT, elementId, elementTail); }

private void validateSchemaElementId(char elementId) throws ArgsException { if (!Character.isLetter(elementId))

throw new ArgsException(INVALID\_ARGUMENT\_NAME, elementId, null); }

private void parseArgumentStrings(List<String> argsList) throws ArgsException {

for (currentArgument = argsList.listIterator(); currentArgument.hasNext();) {

String argString = currentArgument.next(); if (argString.startsWith("-")) {

parseArgumentCharacters(argString.substring(1)); } else {

currentArgument.previous(); break;

} }

}

<https://www.facebook.com/groups/phattrienlaptrinh>

196 **Chapter 14: Successive Reﬁnement**  
196 Chương 14: Tinh luyện liên tiếp

**Listing 14-2 (continued) Args.java  
Liệt kê 14-2 (tiếp theo) Args.java**

private void parseArgumentCharacters(String argChars) throws ArgsException { for (int i = 0; i < argChars.length(); i++)

parseArgumentCharacter(argChars.charAt(i)); }

private void parseArgumentCharacter(char argChar) throws ArgsException { ArgumentMarshaler m = marshalers.get(argChar);

if (m == null) {

throw new ArgsException(UNEXPECTED\_ARGUMENT, argChar, null); } else {

argsFound.add(argChar); try {

m.set(currentArgument); } catch (ArgsException e) {

e.setErrorArgumentId(argChar); throw e;

} }

}

public boolean has(char arg) { return argsFound.contains(arg);

}

public int nextArgument() {

return currentArgument.nextIndex(); }

public boolean getBoolean(char arg) {

return BooleanArgumentMarshaler.getValue(marshalers.get(arg)); }

public String getString(char arg) {

return StringArgumentMarshaler.getValue(marshalers.get(arg)); }

public int getInt(char arg) {

return IntegerArgumentMarshaler.getValue(marshalers.get(arg)); }

public double getDouble(char arg) {

return DoubleArgumentMarshaler.getValue(marshalers.get(arg)); }

public String[] getStringArray(char arg) {

return StringArrayArgumentMarshaler.getValue(marshalers.get(arg)); }

}

Notice that you can read this code from the top to the bottom without a lot of jumping around or looking ahead. The one thing you may have had to look ahead for is the deﬁni-tion of ArgumentMarshaler, which I left out intentionally. Having read this code carefully,  
Lưu ý rằng bạn có thể đọc đoạn mã này từ trên xuống dưới mà không cần phải nhảy xung quanh hoặc nhìn về phía trước. Một điều mà bạn có thể phải tìm kiếm trước đó là định nghĩa của Argument Marshaller, mà tôi đã cố ý bỏ qua. Sau khi đọc mã này một cách cẩn thận,

<https://www.facebook.com/groups/phattrienlaptrinh>

**Args Implementation** 197  
Thực hiện Args 197

you should understand what the ArgumentMarshalerinterface is and what its derivatives do. I’ll show a few of them to you now (Listing 14-3 through Listing 14-6).  
bạn nên hiểu giao diện ArgumentMarshaler là gì và các dẫn xuất của nó làm gì. Bây giờ tôi sẽ cho bạn xem một vài trong số chúng (Liệt kê 14-3 đến Liệt kê 14-6).

**Listing 14-3 ArgumentMarshaler.java**  
Liệt kê 14-3 Đối sốMarshaler.java

public interface ArgumentMarshaler {

void set(Iterator<String> currentArgument) throws ArgsException; }

**Listing 14-4 BooleanArgumentMarshaler.java  
Liệt kê 14-4 BooleanArgumentMarshaler.java**

public class BooleanArgumentMarshaler implements ArgumentMarshaler { private boolean booleanValue = false;

public void set(Iterator<String> currentArgument) throws ArgsException { booleanValue = true;

}

public static boolean getValue(ArgumentMarshaler am) {

if (am != null && am instanceof BooleanArgumentMarshaler) return ((BooleanArgumentMarshaler) am).booleanValue;

else

return false; }

}

**Listing 14-5 StringArgumentMarshaler.java**  
Liệt kê 14-5 StringArgumentMarshaler.java

import static com.objectmentor.utilities.args.ArgsException.ErrorCode.\*;

public class StringArgumentMarshaler implements ArgumentMarshaler { private String stringValue = "";

public void set(Iterator<String> currentArgument) throws ArgsException { try {

stringValue = currentArgument.next(); } catch (NoSuchElementException e) {

throw new ArgsException(MISSING\_STRING); }

}

public static String getValue(ArgumentMarshaler am) {

if (am != null && am instanceof StringArgumentMarshaler) return ((StringArgumentMarshaler) am).stringValue;

else

return ""; }

}

<https://www.facebook.com/groups/phattrienlaptrinh>

198 **Chapter 14: Successive Reﬁnement**  
198 Chương 14: Tinh Luyện Liên Tiếp

**Listing 14-6 IntegerArgumentMarshaler.java  
Liệt kê 14-6 IntegerArgumentMarshaler.java**

import static com.objectmentor.utilities.args.ArgsException.ErrorCode.\*;

public class IntegerArgumentMarshaler implements ArgumentMarshaler { private int intValue = 0;

public void set(Iterator<String> currentArgument) throws ArgsException { String parameter = null;

try {

parameter = currentArgument.next(); intValue = Integer.parseInt(parameter);

} catch (NoSuchElementException e) {

throw new ArgsException(MISSING\_INTEGER); } catch (NumberFormatException e) {

throw new ArgsException(INVALID\_INTEGER, parameter); }

}

public static int getValue(ArgumentMarshaler am) {

if (am != null && am instanceof IntegerArgumentMarshaler) return ((IntegerArgumentMarshaler) am).intValue;

else return 0;

} }

The other ArgumentMarshalerderivatives simply replicate this pattern for doublesand String arrays and would serve to clutter this chapter. I’ll leave them to you as an exercise.  
Các dẫn xuất ArgumentMarshaler khác chỉ đơn giản sao chép mẫu này cho mảng doublesand String và sẽ làm lộn xộn chương này. Tôi sẽ để lại chúng cho bạn như một bài tập.

One other bit of information might be troubling you: the deﬁnition of the error code constants. They are in the ArgsException class (Listing 14-7).  
Một thông tin khác có thể khiến bạn lo lắng: định nghĩa về hằng số mã lỗi. Chúng nằm trong lớp ArgsException (Liệt kê 14-7).

**Listing 14-7 ArgsException.java**  
Liệt kê 14-7 ArgsException.java

import static com.objectmentor.utilities.args.ArgsException.ErrorCode.\*;

public class ArgsException extends Exception { private char errorArgumentId = '\0'; private String errorParameter = null; private ErrorCode errorCode = OK;

public ArgsException() {}

public ArgsException(String message) {super(message);}

public ArgsException(ErrorCode errorCode) { this.errorCode = errorCode;

}

public ArgsException(ErrorCode errorCode, String errorParameter) { this.errorCode = errorCode;

this.errorParameter = errorParameter; }

<https://www.facebook.com/groups/phattrienlaptrinh>

**Args Implementation** 199  
Args thực hiện 199

**Listing 14-7 (continued) ArgsException.java  
Liệt kê 14-7 (tiếp theo) ArgsException.java**

public ArgsException(ErrorCode errorCode,

char errorArgumentId, String errorParameter) { this.errorCode = errorCode;

this.errorParameter = errorParameter; this.errorArgumentId = errorArgumentId;

}

public char getErrorArgumentId() { return errorArgumentId;

}

public void setErrorArgumentId(char errorArgumentId) { this.errorArgumentId = errorArgumentId;

}

public String getErrorParameter() { return errorParameter;

}

public void setErrorParameter(String errorParameter) { this.errorParameter = errorParameter;

}

public ErrorCode getErrorCode() { return errorCode;

}

public void setErrorCode(ErrorCode errorCode) { this.errorCode = errorCode;

}

public String errorMessage() { switch (errorCode) {

case OK:

return "TILT: Should not get here."; case UNEXPECTED\_ARGUMENT:

return String.format("Argument -%c unexpected.", errorArgumentId); case MISSING\_STRING:

return String.format("Could not find string parameter for -%c.", errorArgumentId);

case INVALID\_INTEGER:

return String.format("Argument -%c expects an integer but was '%s'.", errorArgumentId, errorParameter);

case MISSING\_INTEGER:

return String.format("Could not find integer parameter for -%c.", errorArgumentId);

case INVALID\_DOUBLE:

return String.format("Argument -%c expects a double but was '%s'.", errorArgumentId, errorParameter);

case MISSING\_DOUBLE:

return String.format("Could not find double parameter for -%c.", errorArgumentId);

case INVALID\_ARGUMENT\_NAME:

return String.format("'%c' is not a valid argument name.", errorArgumentId);

<https://www.facebook.com/groups/phattrienlaptrinh>

200 **Chapter 14: Successive Reﬁnement**  
200 Chương 14: Tinh Luyện Liên Tiếp

**Listing 14-7 (continued) ArgsException.java**  
Liệt kê 14-7 (tiếp theo) ArgsException.java

case INVALID\_ARGUMENT\_FORMAT:

return String.format("'%s' is not a valid argument format.", errorParameter);

}

return ""; }

public enum ErrorCode {

OK, INVALID\_ARGUMENT\_FORMAT, UNEXPECTED\_ARGUMENT, INVALID\_ARGUMENT\_NAME, MISSING\_STRING,

MISSING\_INTEGER, INVALID\_INTEGER, MISSING\_DOUBLE, INVALID\_DOUBLE}

}

It’s remarkable how much code is required to ﬂesh out the details of this simple con-cept. One of the reasons for this is that we are using a particularly wordy language. Java, being a statically typed language, requires a lot of words in order to satisfy the type sys-tem. In a language like Ruby, Python, or Smalltalk, this program is much smaller.1  
Điều đáng chú ý là cần bao nhiêu mã để đưa ra các chi tiết của khái niệm đơn giản này. Một trong những lý do cho điều này là chúng tôi đang sử dụng một ngôn ngữ đặc biệt dài dòng. Java, là một ngôn ngữ được gõ tĩnh, yêu cầu rất nhiều từ để đáp ứng kiểu hệ thống. Trong một ngôn ngữ như Ruby, Python hoặc Smalltalk, chương trình này nhỏ hơn nhiều.1

Please read the code over one more time. Pay special attention to the way things are named, the size of the functions, and the formatting of the code. If you are an experienced programmer, you may have some quibbles here and there with various parts of the style or structure. Overall, however, I hope you conclude that this program is nicely written and has a clean structure.  
Vui lòng đọc mã hơn một lần nữa. Đặc biệt chú ý đến cách mọi thứ được đặt tên, kích thước của các chức năng và định dạng của mã. Nếu bạn là một lập trình viên có kinh nghiệm, bạn có thể có một số vấn đề ở đây và ở đó với các phần khác nhau của phong cách hoặc cấu trúc. Tuy nhiên, nhìn chung, tôi hy vọng bạn kết luận rằng chương trình này được viết độc đáo và có cấu trúc rõ ràng.

For example, it should be obvious how you would add a new argument type, such as a date argument or a complex number argument, and that such an addition would require a trivial amount of effort. In short, it would simply require a new derivative of Argument-Marshaler, a new getXXXfunction, and a new case statement in the parseSchemaElement function. There would also probably be a new ArgsException.ErrorCodeand a new error message.  
Ví dụ: rõ ràng là bạn sẽ thêm một loại đối số mới như thế nào, chẳng hạn như đối số ngày tháng hoặc đối số số phức, và việc bổ sung như vậy sẽ đòi hỏi một lượng nỗ lực không đáng kể. Tóm lại, nó chỉ đơn giản yêu cầu một dẫn xuất mới của Argument-Marshaler, một hàm getXXX mới và một câu lệnh tình huống mới trong hàm parseSchemaElement. Cũng có thể sẽ có một ArgsException.ErrorCode mới và một thông báo lỗi mới.

[**How Did I Do This?**](#_page_459_0)  
Tôi đã làm điều này như thế nào?

Let me set your mind at rest. I did not simply write this program from beginning to end in its current form. More importantly, I am not expecting you to be able to write clean and elegant programs in one pass. If we have learned anything over the last couple of decades, it is that programming is a craft more than it is a science. To write clean code, you must ﬁrst write dirty code *and then clean it*.  
Hãy để tôi đặt tâm trí của bạn nghỉ ngơi. Tôi không chỉ viết chương trình này từ đầu đến cuối ở dạng hiện tại. Quan trọng hơn, tôi không mong đợi bạn có thể viết các chương trình rõ ràng và thanh lịch chỉ trong một lần. Nếu chúng ta đã học được bất cứ điều gì trong vài thập kỷ qua, đó là lập trình là một nghề thủ công hơn là khoa học. Để viết mã sạch, trước tiên bạn phải viết mã bẩn và sau đó làm sạch nó.

This should not be a surprise to you. We learned this truth in grade school when our teachers tried (usually in vain) to get us to write rough drafts of our compositions. The process, they told us, was that we should write a rough draft, then a second draft, then sev-eral subsequent drafts until we had our ﬁnal version. Writing clean compositions, they tried to tell us, is a matter of successive reﬁnement.  
Điều này không nên là một bất ngờ cho bạn. Chúng tôi đã học được sự thật này ở trường tiểu học khi các giáo viên của chúng tôi cố gắng (thường là vô ích) để bắt chúng tôi viết bản thảo sơ bộ các sáng tác của mình. Họ nói với chúng tôi rằng quy trình là chúng tôi nên viết một bản nháp thô, sau đó là bản nháp thứ hai, rồi vài bản nháp tiếp theo cho đến khi chúng tôi có phiên bản cuối cùng. Họ cố gắng nói với chúng tôi rằng việc viết các tác phẩm rõ ràng là một vấn đề của sự trau chuốt liên tục.

1. I recently rewrote this module in Ruby. It was 1/7th the size and had a subtly better structure.  
1. Gần đây tôi đã viết lại mô-đun này bằng Ruby. Nó có kích thước bằng 1/7 và có cấu trúc tốt hơn một chút.

<https://www.facebook.com/groups/phattrienlaptrinh>

**Args:The Rough Draft** 201  
Lập luận:Bản nháp thô 201

Most freshman programmers (like most grade-schoolers) don’t follow this advice par-ticularly well. They believe that the primary goal is to get the program working. Once it’s “working,” they move on to the next task, leaving the “working” program in whatever state they ﬁnally got it to “work.” Most seasoned programmers know that this is professional suicide.  
Hầu hết các lập trình viên năm nhất (giống như hầu hết các học sinh phổ thông) không đặc biệt làm tốt lời khuyên này. Họ tin rằng mục tiêu chính là làm cho chương trình hoạt động. Sau khi nó “hoạt động”, họ chuyển sang nhiệm vụ tiếp theo, để chương trình “hoạt động” ở bất kỳ trạng thái nào mà cuối cùng họ đã đưa nó vào trạng thái “hoạt động”. Hầu hết các lập trình viên dày dạn kinh nghiệm đều biết rằng đây là hành vi tự sát nghề nghiệp.

[**Args:The Rough Draft**](#_page_459_0)  
Lập luận:Bản nháp thô

Listing 14-8 shows an earlier version of the Args class. It “works.” And it’s messy.  
Liệt kê 14-8 cho thấy một phiên bản cũ hơn của lớp Args. Nó “hoạt động”. Và nó lộn xộn.

**Listing 14-8  
Liệt kê 14-8**

**Args.java (first draft)**

import java.text.ParseException; import java.util.\*;

public class Args { private String schema; private String[] args;

private boolean valid = true;

private Set<Character> unexpectedArguments = new TreeSet<Character>(); private Map<Character, Boolean> booleanArgs =

new HashMap<Character, Boolean>();

private Map<Character, String> stringArgs = new HashMap<Character, String>(); private Map<Character, Integer> intArgs = new HashMap<Character, Integer>(); private Set<Character> argsFound = new HashSet<Character>();

private int currentArgument; private char errorArgumentId = '\0';

private String errorParameter = "TILT"; private ErrorCode errorCode = ErrorCode.OK;

private enum ErrorCode {

OK, MISSING\_STRING, MISSING\_INTEGER, INVALID\_INTEGER, UNEXPECTED\_ARGUMENT}

public Args(String schema, String[] args) throws ParseException { this.schema = schema;

this.args = args; valid = parse();

}

private boolean parse() throws ParseException { if (schema.length() == 0 && args.length == 0)

return true; parseSchema(); try {

parseArguments();

} catch (ArgsException e) { }

return valid; }

private boolean parseSchema() throws ParseException { for (String element : schema.split(",")) {

<https://www.facebook.com/groups/phattrienlaptrinh>

202 **Chapter 14: Successive Reﬁnement**  
202 Chương 14: Tinh luyện liên tiếp

**Listing 14-8 (continued) Args.java (first draft)**  
Liệt kê 14-8 (tiếp theo) Args.java (bản thảo đầu tiên)

if (element.length() > 0) {

String trimmedElement = element.trim(); parseSchemaElement(trimmedElement);

} }

return true; }

private void parseSchemaElement(String element) throws ParseException { char elementId = element.charAt(0);

String elementTail = element.substring(1); validateSchemaElementId(elementId);

if (isBooleanSchemaElement(elementTail)) parseBooleanSchemaElement(elementId);

else if (isStringSchemaElement(elementTail)) parseStringSchemaElement(elementId);

else if (isIntegerSchemaElement(elementTail)) { parseIntegerSchemaElement(elementId);

} else {

throw new ParseException(

String.format("Argument: %c has invalid format: %s.", elementId, elementTail), 0);

} }

private void validateSchemaElementId(char elementId) throws ParseException { if (!Character.isLetter(elementId)) {

throw new ParseException(

"Bad character:" + elementId + "in Args format: " + schema, 0); }

}

private void parseBooleanSchemaElement(char elementId) { booleanArgs.put(elementId, false);

}

private void parseIntegerSchemaElement(char elementId) { intArgs.put(elementId, 0);

}

private void parseStringSchemaElement(char elementId) { stringArgs.put(elementId, "");

}

private boolean isStringSchemaElement(String elementTail) { return elementTail.equals("\*");

}

private boolean isBooleanSchemaElement(String elementTail) { return elementTail.length() == 0;

}

private boolean isIntegerSchemaElement(String elementTail) { return elementTail.equals("#");

}

<https://www.facebook.com/groups/phattrienlaptrinh>

**Args:The Rough Draft** 203  
Lập luận:Bản nháp thô 203

**Listing 14-8 (continued) Args.java (first draft)  
Liệt kê 14-8 (tiếp theo) Args.java (bản thảo đầu tiên)**

private boolean parseArguments() throws ArgsException {

for (currentArgument = 0; currentArgument < args.length; currentArgument++) {

String arg = args[currentArgument]; parseArgument(arg);

}

return true; }

private void parseArgument(String arg) throws ArgsException { if (arg.startsWith("-"))

parseElements(arg); }

private void parseElements(String arg) throws ArgsException { for (int i = 1; i < arg.length(); i++)

parseElement(arg.charAt(i)); }

private void parseElement(char argChar) throws ArgsException { if (setArgument(argChar))

argsFound.add(argChar); else {

unexpectedArguments.add(argChar); errorCode = ErrorCode.UNEXPECTED\_ARGUMENT; valid = false;

} }

private boolean setArgument(char argChar) throws ArgsException { if (isBooleanArg(argChar))

setBooleanArg(argChar, true); else if (isStringArg(argChar))

setStringArg(argChar); else if (isIntArg(argChar))

setIntArg(argChar); else

return false;

return true; }

private boolean isIntArg(char argChar) {return intArgs.containsKey(argChar);}

private void setIntArg(char argChar) throws ArgsException { currentArgument++;

String parameter = null; try {

parameter = args[currentArgument]; intArgs.put(argChar, new Integer(parameter));

} catch (ArrayIndexOutOfBoundsException e) { valid = false;

errorArgumentId = argChar;

errorCode = ErrorCode.MISSING\_INTEGER;

<https://www.facebook.com/groups/phattrienlaptrinh>

204 **Chapter 14: Successive Reﬁnement**  
204 Chương 14: Tinh luyện liên tiếp

**Listing 14-8 (continued) Args.java (first draft)**  
Liệt kê 14-8 (tiếp theo) Args.java (bản thảo đầu tiên)

throw new ArgsException();

} catch (NumberFormatException e) { valid = false;

errorArgumentId = argChar; errorParameter = parameter;

errorCode = ErrorCode.INVALID\_INTEGER; throw new ArgsException();

} }

private void setStringArg(char argChar) throws ArgsException { currentArgument++;

try {

stringArgs.put(argChar, args[currentArgument]); } catch (ArrayIndexOutOfBoundsException e) {

valid = false; errorArgumentId = argChar;

errorCode = ErrorCode.MISSING\_STRING; throw new ArgsException();

} }

private boolean isStringArg(char argChar) { return stringArgs.containsKey(argChar);

}

private void setBooleanArg(char argChar, boolean value) { booleanArgs.put(argChar, value);

}

private boolean isBooleanArg(char argChar) { return booleanArgs.containsKey(argChar);

}

public int cardinality() { return argsFound.size();

}

public String usage() { if (schema.length() > 0)

return "-[" + schema + "]"; else

return ""; }

public String errorMessage() throws Exception { switch (errorCode) {

case OK:

throw new Exception("TILT: Should not get here."); case UNEXPECTED\_ARGUMENT:

return unexpectedArgumentMessage(); case MISSING\_STRING:

return String.format("Could not find string parameter for -%c.", errorArgumentId);

<https://www.facebook.com/groups/phattrienlaptrinh>

**Args:The Rough Draft** 205  
Lập luận:Bản nháp thô 205

**Listing 14-8 (continued) Args.java (first draft)  
Liệt kê 14-8 (tiếp theo) Args.java (bản thảo đầu tiên)**

case INVALID\_INTEGER:

return String.format("Argument -%c expects an integer but was '%s'.", errorArgumentId, errorParameter);

case MISSING\_INTEGER:

return String.format("Could not find integer parameter for -%c.", errorArgumentId);

}

return ""; }

private String unexpectedArgumentMessage() {

StringBuffer message = new StringBuffer("Argument(s) -"); for (char c : unexpectedArguments) {

message.append(c); }

message.append(" unexpected.");

return message.toString(); }

private boolean falseIfNull(Boolean b) { return b != null && b;

}

private int zeroIfNull(Integer i) { return i == null ? 0 : i;

}

private String blankIfNull(String s) { return s == null ? "" : s;

}

public String getString(char arg) { return blankIfNull(stringArgs.get(arg));

}

public int getInt(char arg) {

return zeroIfNull(intArgs.get(arg)); }

public boolean getBoolean(char arg) { return falseIfNull(booleanArgs.get(arg));

}

public boolean has(char arg) { return argsFound.contains(arg);

}

public boolean isValid() { return valid;

}

private class ArgsException extends Exception { }

}

<https://www.facebook.com/groups/phattrienlaptrinh>

206 **Chapter 14: Successive Reﬁnement**  
206 Chương 14: Tinh luyện liên tiếp

I hope your initial reaction to this mass of code is “I’m certainly glad he didn’t leave it like that!” If you feel like this, then remember that’s how other people are going to feel about code that you leave in rough-draft form.  
Tôi hy vọng phản ứng ban đầu của bạn đối với khối mã này là "Tôi rất vui vì anh ấy đã không để nó như vậy!" Nếu bạn cảm thấy như vậy, thì hãy nhớ rằng đó là cách người khác sẽ cảm nhận về mã mà bạn để lại ở dạng bản nháp.

Actually “rough draft” is probably the kindest thing you can say about this code. It’s clearly a work in progress. The sheer number of instance variables is daunting. The odd strings like “TILT,”the HashSetsand TreeSets, and the try-catch-catchblocks all add up to a festering pile.  
Trên thực tế, "bản thảo thô" có lẽ là điều tử tế nhất mà bạn có thể nói về đoạn mã này. Đó rõ ràng là một công việc đang được tiến hành. Số lượng tuyệt đối của các biến thể hiện là khó khăn. Các chuỗi lẻ như “TILT,” HashSetsand TreeSets và các khối try-catch-catch tất cả tạo thành một đống đang mưng mủ.

I had not wanted to write a festering pile. Indeed, I was trying to keep things reason-ably well organized. You can probably tell that from my choice of function and variable names and the fact that there is a crude structure to the program. But, clearly, I had let the problem get away from me.  
Tôi đã không muốn viết một đống mưng mủ. Thật vậy, tôi đã cố gắng sắp xếp mọi thứ hợp lý. Bạn có thể có thể nói điều đó từ sự lựa chọn của tôi về tên hàm và tên biến và thực tế là có một cấu trúc thô cho chương trình. Nhưng, rõ ràng, tôi đã để cho vấn đề thoát khỏi tôi.

The mess built gradually. Earlier versions had not been nearly so nasty. For example, Listing 14-9 shows an earlier version in which only Boolean arguments were working.  
Sự lộn xộn được xây dựng dần dần. Các phiên bản trước đó gần như không quá khó chịu. Ví dụ, Liệt kê 14-9 hiển thị một phiên bản cũ hơn trong đó chỉ có các đối số Boolean hoạt động.

**Listing 14-9**

**Args.java (Boolean only)  
Args.java (chỉ Boolean)**

package com.objectmentor.utilities.getopts;

import java.util.\*;

public class Args { private String schema; private String[] args; private boolean valid;

private Set<Character> unexpectedArguments = new TreeSet<Character>(); private Map<Character, Boolean> booleanArgs =

new HashMap<Character, Boolean>(); private int numberOfArguments = 0;

public Args(String schema, String[] args) { this.schema = schema;

this.args = args; valid = parse();

}

public boolean isValid() { return valid;

}

private boolean parse() {

if (schema.length() == 0 && args.length == 0) return true;

parseSchema(); parseArguments();

return unexpectedArguments.size() == 0; }

private boolean parseSchema() {

for (String element : schema.split(",")) { parseSchemaElement(element);

}

<https://www.facebook.com/groups/phattrienlaptrinh>

**Args:The Rough Draft** 207  
Lập luận:Bản nháp thô 207

**Listing 14-9 (continued) Args.java (Boolean only)  
Liệt kê 14-9 (tiếp theo) Args.java (chỉ Boolean)**

return true; }

private void parseSchemaElement(String element) { if (element.length() == 1) {

parseBooleanSchemaElement(element); }

}

private void parseBooleanSchemaElement(String element) { char c = element.charAt(0);

if (Character.isLetter(c)) { booleanArgs.put(c, false);

} }

private boolean parseArguments() { for (String arg : args)

parseArgument(arg); return true;

}

private void parseArgument(String arg) { if (arg.startsWith("-"))

parseElements(arg); }

private void parseElements(String arg) { for (int i = 1; i < arg.length(); i++)

parseElement(arg.charAt(i)); }

private void parseElement(char argChar) { if (isBoolean(argChar)) {

numberOfArguments++; setBooleanArg(argChar, true);

} else unexpectedArguments.add(argChar);

}

private void setBooleanArg(char argChar, boolean value) { booleanArgs.put(argChar, value);

}

private boolean isBoolean(char argChar) { return booleanArgs.containsKey(argChar);

}

public int cardinality() { return numberOfArguments;

}

public String usage() { if (schema.length() > 0)

return "-["+schema+"]";

<https://www.facebook.com/groups/phattrienlaptrinh>

208 **Chapter 14: Successive Reﬁnement**  
208 Chương 14: Tinh Luyện Liên Tiếp

**Listing 14-9 (continued) Args.java (Boolean only)**  
Liệt kê 14-9 (tiếp theo) Args.java (chỉ Boolean)

else

return ""; }

public String errorMessage() {

if (unexpectedArguments.size() > 0) { return unexpectedArgumentMessage();

} else return "";

}

private String unexpectedArgumentMessage() {

StringBuffer message = new StringBuffer("Argument(s) -"); for (char c : unexpectedArguments) {

message.append(c); }

message.append(" unexpected.");

return message.toString(); }

public boolean getBoolean(char arg) { return booleanArgs.get(arg);

} }

Although you can ﬁnd plenty to complain about in this code, it’s really not that bad. It’s compact and simple and easy to understand. However, within this code it is easy to see the seeds of the later festering pile. It’s quite clear how this grew into the latter mess.  
Mặc dù bạn có thể tìm thấy nhiều điều để phàn nàn về mã này, nhưng nó thực sự không tệ đến thế. Nó nhỏ gọn, đơn giản và dễ hiểu. Tuy nhiên, trong mã này, có thể dễ dàng nhìn thấy hạt giống của đống mưng mủ sau này. Rõ ràng là điều này đã phát triển thành mớ hỗn độn sau này như thế nào.

Notice that the latter mess has only two more argument types than this: Stringand integer. The addition of just two more argument types had a massively negative impact on the code. It converted it from something that would have been reasonably maintainable into something that I would expect to become riddled with bugs and warts.  
Lưu ý rằng mớ hỗn độn sau chỉ có hai loại đối số hơn loại này: Chuỗi và số nguyên. Việc bổ sung thêm hai loại đối số có tác động tiêu cực lớn đến mã. Nó đã biến nó từ một thứ có thể bảo trì được một cách hợp lý thành một thứ mà tôi mong đợi sẽ trở nên đầy bọ và mụn cóc.

I added the two argument types incrementally. First, I added the Stringargument, which yielded this:  
Tôi đã thêm dần hai loại đối số. Đầu tiên, tôi đã thêm Stringargument, mang lại điều này:

**Listing 14-10  
Liệt kê 14-10**

**Args.java (Boolean and String)** package com.objectmentor.utilities.getopts;  
Gói Args.java (Boolean và String) com.objectmentor.utilities.getopts;

import java.text.ParseException; import java.util.\*;

public class Args { private String schema; private String[] args;

private boolean valid = true;

private Set<Character> unexpectedArguments = new TreeSet<Character>(); private Map<Character, Boolean> booleanArgs =

new HashMap<Character, Boolean>();

<https://www.facebook.com/groups/phattrienlaptrinh>

**Args:The Rough Draft** 209  
Lập luận:Bản nháp thô 209

**Listing 14-10 (continued)**  
Liệt kê 14-10 (tiếp theo)

**Args.java (Boolean and String)  
Args.java (Boolean và String)**

private Map<Character, String> stringArgs = new HashMap<Character, String>();

private Set<Character> argsFound = new HashSet<Character>(); private int currentArgument;

private char errorArgument = '\0';

enum ErrorCode {

OK, MISSING\_STRING}

private ErrorCode errorCode = ErrorCode.OK;

public Args(String schema, String[] args) throws ParseException { this.schema = schema;

this.args = args; valid = parse();

}

private boolean parse() throws ParseException { if (schema.length() == 0 && args.length == 0)

return true; parseSchema(); parseArguments(); return valid;

}

private boolean parseSchema() throws ParseException { for (String element : schema.split(",")) {

if (element.length() > 0) {

String trimmedElement = element.trim(); parseSchemaElement(trimmedElement);

} }

return true; }

private void parseSchemaElement(String element) throws ParseException { char elementId = element.charAt(0);

String elementTail = element.substring(1); validateSchemaElementId(elementId);

if (isBooleanSchemaElement(elementTail)) parseBooleanSchemaElement(elementId);

else if (isStringSchemaElement(elementTail)) parseStringSchemaElement(elementId);

}

private void validateSchemaElementId(char elementId) throws ParseException { if (!Character.isLetter(elementId)) {

throw new ParseException(

"Bad character:" + elementId + "in Args format: " + schema, 0); }

}

private void parseStringSchemaElement(char elementId) { stringArgs.put(elementId, "");

}

<https://www.facebook.com/groups/phattrienlaptrinh>

210 **Chapter 14: Successive Reﬁnement**  
210 Chương 14: Tinh Luyện Liên Tiếp

**Listing 14-10 (continued)**

**Args.java (Boolean and String)  
Args.java (Boolean và String)**

private boolean isStringSchemaElement(String elementTail) { return elementTail.equals("\*");

}

private boolean isBooleanSchemaElement(String elementTail) { return elementTail.length() == 0;

}

private void parseBooleanSchemaElement(char elementId) { booleanArgs.put(elementId, false);

}

private boolean parseArguments() {

for (currentArgument = 0; currentArgument < args.length; currentArgument++) {

String arg = args[currentArgument]; parseArgument(arg);

}

return true; }

private void parseArgument(String arg) { if (arg.startsWith("-"))

parseElements(arg); }

private void parseElements(String arg) { for (int i = 1; i < arg.length(); i++)

parseElement(arg.charAt(i)); }

private void parseElement(char argChar) { if (setArgument(argChar))

argsFound.add(argChar); else {

unexpectedArguments.add(argChar); valid = false;

} }

private boolean setArgument(char argChar) { boolean set = true;

if (isBoolean(argChar)) setBooleanArg(argChar, true);

else if (isString(argChar)) setStringArg(argChar, "");

else

set = false;

return set; }

private void setStringArg(char argChar, String s) { currentArgument++;

try {

<https://www.facebook.com/groups/phattrienlaptrinh>

**Args:The Rough Draft** 211  
Lập luận:Bản nháp thô 211

**Listing 14-10 (continued)**  
Liệt kê 14-10 (tiếp theo)

**Args.java (Boolean and String)  
Args.java (Boolean và String)**

stringArgs.put(argChar, args[currentArgument]); } catch (ArrayIndexOutOfBoundsException e) {

valid = false; errorArgument = argChar;

errorCode = ErrorCode.MISSING\_STRING; }

}

private boolean isString(char argChar) { return stringArgs.containsKey(argChar);

}

private void setBooleanArg(char argChar, boolean value) { booleanArgs.put(argChar, value);

}

private boolean isBoolean(char argChar) { return booleanArgs.containsKey(argChar);

}

public int cardinality() { return argsFound.size();

}

public String usage() { if (schema.length() > 0)

return "-[" + schema + "]"; else

return ""; }

public String errorMessage() throws Exception { if (unexpectedArguments.size() > 0) {

return unexpectedArgumentMessage(); } else

switch (errorCode) { case MISSING\_STRING:

return String.format("Could not find string parameter for -%c.", errorArgument);

case OK:

throw new Exception("TILT: Should not get here."); }

return ""; }

private String unexpectedArgumentMessage() {

StringBuffer message = new StringBuffer("Argument(s) -"); for (char c : unexpectedArguments) {

message.append(c); }

message.append(" unexpected.");

return message.toString(); }

<https://www.facebook.com/groups/phattrienlaptrinh>

212 **Chapter 14: Successive Reﬁnement**  
212 Chương 14: Tinh Luyện Liên Tiếp

**Listing 14-10 (continued)**

**Args.java (Boolean and String)  
Args.java (Boolean và String)**

public boolean getBoolean(char arg) { return falseIfNull(booleanArgs.get(arg));

}

private boolean falseIfNull(Boolean b) { return b == null ? false : b;

}

public String getString(char arg) { return blankIfNull(stringArgs.get(arg));

}

private String blankIfNull(String s) { return s == null ? "" : s;

}

public boolean has(char arg) { return argsFound.contains(arg);

}

public boolean isValid() { return valid;

} }

You can see that this is starting to get out of hand. It’s still not horrible, but the mess is certainly starting to grow. It’s a pile, but it’s not festering quite yet. It took the addition of the integer argument type to get this pile really fermenting and festering.  
Bạn có thể thấy rằng điều này đang bắt đầu vượt khỏi tầm kiểm soát. Nó vẫn chưa đến mức khủng khiếp, nhưng chắc chắn là mớ hỗn độn đang bắt đầu lớn dần lên. Đó là một đống, nhưng nó vẫn chưa mưng mủ. Cần phải bổ sung loại đối số số nguyên để làm cho đống này thực sự lên men và mưng mủ.

[**So I Stopped**](#_page_459_0)  
Vì vậy, tôi dừng lại

I had at least two more argument types to add, and I could tell that they would make things much worse. If I bulldozed my way forward, I could probably get them to work, but I’d leave behind a mess that was too large to ﬁx. If the structure of this code was ever going to be maintainable, now was the time to ﬁx it.  
Tôi có thêm ít nhất hai loại đối số nữa và tôi có thể nói rằng chúng sẽ khiến mọi thứ trở nên tồi tệ hơn nhiều. Nếu tôi tiến lên phía trước, tôi có thể khiến chúng hoạt động, nhưng tôi sẽ để lại một đống hỗn độn quá lớn để sửa chữa. Nếu cấu trúc của đoạn mã này có thể duy trì được, thì bây giờ là lúc để sửa nó.

So I stopped adding features and started refactoring. Having just added the Stringand integerarguments, I knew that each argument type required new code in three major places. First, each argument type required some way to parse its schema element in order to select the HashMapfor that type. Next, each argument type needed to be parsed in the command-line strings and converted to its true type. Finally, each argument type needed a getXXX method so that it could be returned to the caller as its true type.  
Vì vậy, tôi đã ngừng thêm các tính năng và bắt đầu tái cấu trúc. Vừa mới thêm các đối số số nguyên Stringand, tôi biết rằng mỗi loại đối số yêu cầu mã mới ở ba vị trí chính. Đầu tiên, mỗi loại đối số yêu cầu một số cách để phân tích thành phần lược đồ của nó để chọn HashMap cho loại đó. Tiếp theo, mỗi loại đối số cần được phân tích cú pháp trong chuỗi dòng lệnh và được chuyển đổi thành loại thực của nó. Cuối cùng, mỗi loại đối số cần một phương thức getXXX để nó có thể được trả lại cho người gọi dưới dạng đúng loại của nó.

Many different types, all with similar methods—that sounds like a class to me. And so the ArgumentMarshaler concept was born.  
Nhiều loại khác nhau, tất cả đều có các phương thức tương tự—đối với tôi, điều đó giống như một lớp học. Và thế là khái niệm ArgumentMarshaler ra đời.

[**On Incrementalism**](#_page_459_0)  
về chủ nghĩa gia tăng

One of the best ways to ruin a program is to make massive changes to its structure in the name of improvement. Some programs never recover from such “improvements.” The problem is that it’s very hard to get the program working the same way it worked before the “improvement.”  
Một trong những cách tốt nhất để làm hỏng một chương trình là thực hiện những thay đổi lớn đối với cấu trúc của nó dưới danh nghĩa cải tiến. Một số chương trình không bao giờ phục hồi sau những “cải tiến” như vậy. Vấn đề là rất khó để chương trình hoạt động giống như trước khi “cải tiến”.

<https://www.facebook.com/groups/phattrienlaptrinh>

**Args:The Rough Draft** 213  
Lập luận:Bản nháp thô 213

To avoid this, I use the discipline of Test-Driven Development (TDD). One of the cen-tral doctrines of this approach is to keep the system running at all times. In other words, using TDD, I am not allowed to make a change to the system that breaks that system. Every change I make must keep the system working as it worked before.  
Để tránh điều này, tôi sử dụng nguyên tắc Phát triển dựa trên thử nghiệm (TDD). Một trong những học thuyết trung tâm của phương pháp này là giữ cho hệ thống luôn hoạt động. Nói cách khác, sử dụng TDD, tôi không được phép thực hiện thay đổi đối với hệ thống làm hỏng hệ thống đó. Mọi thay đổi tôi thực hiện phải giữ cho hệ thống hoạt động như trước đây.

To achieve this, I need a suite of automated tests that I can run on a whim and that ver-iﬁes that the behavior of the system is unchanged. For the Argsclass I had created a suite of unit and acceptance tests while I was building the festering pile. The unit tests were written in Javaand administered by JUnit. The acceptance tests were written as wiki pages in FitNesse. I could run these tests any time I wanted, and if they passed, I was conﬁdent that the system was working as I speciﬁed.  
Để đạt được điều này, tôi cần một bộ kiểm tra tự động mà tôi có thể chạy tùy ý và xác minh rằng hoạt động của hệ thống không thay đổi. Đối với Argsclass, tôi đã tạo một bộ các bài kiểm tra nghiệm thu và đơn vị trong khi tôi đang xây dựng đống mưng mủ. Các bài kiểm tra đơn vị được viết bằng Java và do JUnit quản lý. Các bài kiểm tra chấp nhận được viết dưới dạng các trang wiki trong FitNesse. Tôi có thể chạy các bài kiểm tra này bất cứ lúc nào tôi muốn và nếu chúng vượt qua, tôi tin chắc rằng hệ thống đang hoạt động như tôi đã chỉ định.

So I proceeded to make a large number of very tiny changes. Each change moved the structure of the system toward the ArgumentMarshalerconcept. And yet each change kept the system working. The ﬁrst change I made was to add the skeleton of the ArgumentMarshaller to the end of the festering pile (Listing 14-11).  
Vì vậy, tôi đã tiến hành thực hiện một số lượng lớn các thay đổi rất nhỏ. Mỗi thay đổi đã chuyển cấu trúc của hệ thống sang khái niệm ArgumentMarshaler. Tuy nhiên, mỗi thay đổi vẫn giữ cho hệ thống hoạt động. Thay đổi đầu tiên mà tôi thực hiện là thêm khung của Argument Marshaller vào cuối đống mưng mủ (Liệt kê 14-11).

**Listing 14-11**

**ArgumentMarshaller appended to Args.java  
ArgumentMarshaller nối vào Args.java**

private class ArgumentMarshaler { private boolean booleanValue = false;

public void setBoolean(boolean value) { booleanValue = value;

}

public boolean getBoolean() {return booleanValue;} }

private class BooleanArgumentMarshaler extends ArgumentMarshaler { }

private class StringArgumentMarshaler extends ArgumentMarshaler { }

private class IntegerArgumentMarshaler extends ArgumentMarshaler { }

}

Clearly, this wasn’t going to break anything. So then I made the simplest modiﬁcation I could, one that would break as little as possible. I changed the HashMapfor the Boolean arguments to take an ArgumentMarshaler.  
Rõ ràng, điều này sẽ không phá vỡ bất cứ điều gì. Vì vậy, sau đó tôi thực hiện sửa đổi đơn giản nhất có thể, một sửa đổi ít bị hỏng nhất có thể. Tôi đã thay đổi HashMap cho các đối số Boolean để lấy một ArgumentMarshaler.

private Map<Character, **ArgumentMarshaler**> booleanArgs = new HashMap<Character, **ArgumentMarshaler**>();

This broke a few statements, which I quickly ﬁxed.  
Điều này đã phá vỡ một vài tuyên bố, mà tôi đã nhanh chóng sửa chữa.

...

private void parseBooleanSchemaElement(char elementId) { booleanArgs.put(elementId, **new BooleanArgumentMarshaler()**);

} ..

<https://www.facebook.com/groups/phattrienlaptrinh>

214 **Chapter 14: Successive Reﬁnement**  
214 Chương 14: Tinh luyện liên tiếp

private void setBooleanArg(char argChar, boolean value) { booleanArgs.**get**(argChar)**.setBoolean**(value);

} ...

public boolean getBoolean(char arg) {

return falseIfNull(booleanArgs.get(arg).**getBoolean()**); }

Notice how these changes are in exactly the areas that I mentioned before: the parse, set, and getfor the argument type. Unfortunately, small as this change was, some of the tests started failing. If you look carefully at getBoolean, you’ll see that if you call it with 'y,'but there is no yargument, then booleanArgs.get('y')will return null, and the func-tion will throw a NullPointerException. The falseIfNullfunction had been used to protect against this, but the change I made caused that function to become irrelevant.  
Lưu ý cách những thay đổi này diễn ra chính xác trong các lĩnh vực mà tôi đã đề cập trước đây: phân tích cú pháp, đặt và nhận đối với loại đối số. Thật không may, dù thay đổi này rất nhỏ, một số thử nghiệm đã bắt đầu thất bại. Nếu bạn xem kỹ getBoolean, bạn sẽ thấy rằng nếu bạn gọi nó bằng 'y', nhưng không có yargument, thì booleanArgs.get('y') sẽ trả về giá trị rỗng và hàm này sẽ đưa ra Ngoại lệ NullPulum. Hàm falseIfNull đã được sử dụng để bảo vệ chống lại điều này, nhưng thay đổi tôi thực hiện đã khiến hàm đó trở nên không liên quan.

Incrementalism demanded that I get this working quickly before making any other changes. Indeed, the ﬁx was not too difﬁcult. I just had to move the check for null. It was no longer the boolean being null that I needed to check; it was the ArgumentMarshaller.  
Chủ nghĩa gia tăng yêu cầu tôi phải nhanh chóng làm việc này trước khi thực hiện bất kỳ thay đổi nào khác. Thật vậy, việc sửa chữa không quá khó khăn. Tôi chỉ phải di chuyển kiểm tra null. Tôi không cần phải kiểm tra boolean null nữa; đó là ArgumentMarshaller.

First, I removed the falseIfNullcall in the getBooleanfunction. It was useless now, so I also eliminated the function itself. The tests still failed in the same way, so I was conﬁ-dent that I hadn’t introduced any new errors.  
Trước tiên, tôi đã xóa falseIfNullcall trong hàm getBoolean. Bây giờ nó vô dụng, vì vậy tôi cũng loại bỏ chức năng đó. Các bài kiểm tra vẫn thất bại theo cùng một cách, vì vậy tôi tự tin rằng mình đã không đưa ra bất kỳ lỗi mới nào.

public boolean getBoolean(char arg) { return booleanArgs.get(arg).getBoolean();

}

Next, I split the function into two lines and put the ArgumentMarshallerinto its own vari-able named argumentMarshaller. I didn’t care for the long variable name; it was badly redundant and cluttered up the function. So I shortened it to am [N5].  
Tiếp theo, tôi chia hàm thành hai dòng và đặt ArgumentMarshaller vào đối số có thể thay đổi có tên là Marshaller của chính nó. Tôi không quan tâm đến tên biến dài; nó rất dư thừa và làm lộn xộn chức năng. Vì vậy, tôi đã rút ngắn nó thành am [N5].

public boolean getBoolean(char arg) { Args.ArgumentMarshaler **am** = booleanArgs.get(arg); return **am**.getBoolean();

}

And then I put in the null detection logic.  
Và sau đó tôi đưa vào logic phát hiện null.

public boolean getBoolean(char arg) { Args.ArgumentMarshaler am = booleanArgs.get(arg); return **am != null &&** am.getBoolean();

}

[**String Arguments**](#_page_459_0)  
Đối số chuỗi

Adding Stringarguments was very similar to adding booleanarguments. I had to change the HashMapand get the parse, set, and getfunctions working. There shouldn’t be any sur-prises in what follows except, perhaps, that I seem to be putting all the marshalling imple-mentation in the ArgumentMarshaller base class instead of distributing it to the derivatives.  
Việc thêm các đối số chuỗi rất giống với việc thêm các đối số boolean. Tôi đã phải thay đổi HashMap và làm cho các hàm phân tích, thiết lập và get hoạt động. Không nên có bất kỳ sự ngạc nhiên nào trong những điều tiếp theo ngoại trừ, có lẽ, rằng tôi dường như đang đặt tất cả việc triển khai sắp xếp thứ tự trong lớp cơ sở ArgumentMarshaller thay vì phân phối nó cho các công cụ phái sinh.

private Map<Character**, ArgumentMarshaler**> stringArgs = new HashMap<Character**, ArgumentMarshaler**>();

...

<https://www.facebook.com/groups/phattrienlaptrinh>

**String Arguments** 215  
Đối số chuỗi 215

private void parseStringSchemaElement(char elementId) { stringArgs.put(elementId**, new StringArgumentMarshaler()**);

} ...

private void setStringArg(char argChar) throws ArgsException { currentArgument++;

try { stringArgs**.get**(argChar)**.setString**(args[currentArgument]);

} catch (ArrayIndexOutOfBoundsException e) { valid = false;

errorArgumentId = argChar;

errorCode = ErrorCode.MISSING\_STRING; throw new ArgsException();

} }

...

public String getString(char arg) { **Args.ArgumentMarshaler am =** stringArgs.get(arg); return **am == null ? ""** : am.getString();

} ...

private class ArgumentMarshaler { private boolean booleanValue = false; **private String stringValue;**

public void setBoolean(boolean value) { booleanValue = value;

}

public boolean getBoolean() { return booleanValue;

}

**public void setString(String s) { stringValue = s;**

**}**

**public String getString() {**

**return stringValue == null ? "" : stringValue; }**

}

Again, these changes were made one at a time and in such a way that the tests kept running, if not passing. When a test broke, I made sure to get it passing again before con-tinuing with the next change.  
Một lần nữa, những thay đổi này được thực hiện lần lượt và theo cách sao cho các bài kiểm tra tiếp tục chạy, nếu không vượt qua. Khi một bài kiểm tra bị lỗi, tôi đảm bảo rằng nó sẽ vượt qua một lần nữa trước khi tiếp tục với thay đổi tiếp theo.

By now you should be able to see my intent. Once I get all the current marshalling behavior into the ArgumentMarshalerbase class, I’m going to start pushing that behavior down into the derivatives. This will allow me to keep everything running while I gradually change the shape of this program.  
Bây giờ bạn sẽ có thể nhìn thấy ý định của tôi. Sau khi tôi nhận được tất cả các hành vi sắp xếp hiện tại vào lớp ArgumentMarshalerbase, tôi sẽ bắt đầu đẩy hành vi đó xuống các công cụ phái sinh. Điều này sẽ cho phép tôi giữ mọi thứ chạy trong khi tôi dần dần thay đổi hình dạng của chương trình này.

The obvious next step was to move the int argument functionality into the ArgumentMarshaler. Again, there weren’t any surprises.  
Bước tiếp theo rõ ràng là chuyển chức năng đối số int vào ArgumentMarshaler. Một lần nữa, không có bất kỳ sự ngạc nhiên nào.

private Map<Character**, ArgumentMarshaler**> intArgs = new HashMap<Character**, ArgumentMarshaler**>();

...

<https://www.facebook.com/groups/phattrienlaptrinh>

216 **Chapter 14: Successive Reﬁnement**  
216 Chương 14: Tinh Luyện Liên Tiếp

private void parseIntegerSchemaElement(char elementId) { intArgs.put(elementId, **new IntegerArgumentMarshaler()**);

} ...

private void setIntArg(char argChar) throws ArgsException { currentArgument++;

String parameter = null; try {

parameter = args[currentArgument]; intArgs**.get**(argChar)**.setInteger**(Integer.parseInt(parameter));

} catch (ArrayIndexOutOfBoundsException e) { valid = false;

errorArgumentId = argChar;

errorCode = ErrorCode.MISSING\_INTEGER; throw new ArgsException();

} catch (NumberFormatException e) { valid = false;

errorArgumentId = argChar; errorParameter = parameter;

errorCode = ErrorCode.INVALID\_INTEGER; throw new ArgsException();

} }

...

public int getInt(char arg) { **Args.ArgumentMarshaler am =** intArgs.get(arg); return **am == null ? 0** : am.getInteger();

} ...

private class ArgumentMarshaler { private boolean booleanValue = false; private String stringValue;

**private int integerValue;**

public void setBoolean(boolean value) { booleanValue = value;

}

public boolean getBoolean() { return booleanValue;

}

public void setString(String s) { stringValue = s;

}

public String getString() {

return stringValue == null ? "" : stringValue; }

**public void setInteger(int i) { integerValue = i;**

**}**

**public int getInteger() { return integerValue;**

**}** }

<https://www.facebook.com/groups/phattrienlaptrinh>

**String Arguments** 217  
Đối số chuỗi 217

With all the marshalling moved to the ArgumentMarshaler, I started pushing functional-ity into the derivatives. The ﬁrst step was to move the setBoolean function into the BooleanArgumentMarshallerand make sure it got called correctly. So I created an abstract set method.  
Với tất cả các sắp xếp được chuyển đến ArgumentMarshaler, tôi bắt đầu đẩy chức năng vào các công cụ phái sinh. Bước đầu tiên là di chuyển hàm setBoolean vào BooleanArgumentMarshaller và đảm bảo rằng nó được gọi chính xác. Vì vậy, tôi đã tạo một phương thức tập hợp trừu tượng.

private **abstract** class ArgumentMarshaler { **protected** boolean booleanValue = false; private String stringValue;

private int integerValue;

public void setBoolean(boolean value) { booleanValue = value;

}

public boolean getBoolean() { return booleanValue;

}

public void setString(String s) { stringValue = s;

}

public String getString() {

return stringValue == null ? "" : stringValue; }

public void setInteger(int i) { integerValue = i;

}

public int getInteger() { return integerValue;

}

**public abstract void set(String s); }**

Then I implemented the set method in BooleanArgumentMarshaller.  
Sau đó, tôi đã triển khai phương thức thiết lập trong BooleanArgumentMarshaller.

private class BooleanArgumentMarshaler extends ArgumentMarshaler { **public void set(String s) {**

**booleanValue = true; }**

}

And ﬁnally I replaced the call to setBoolean with a call to set.  
Và cuối cùng tôi đã thay thế lệnh gọi setBoolean bằng lệnh gọi set.

private void setBooleanArg(char argChar, boolean value) { booleanArgs.get(argChar)**.set("true");**

}

The tests all still passed. Because this change caused setto be deployed to the Boolean-ArgumentMarshaler, I removed the setBooleanmethod from the ArgumentMarshalerbase class.  
Các bài kiểm tra tất cả vẫn vượt qua. Vì thay đổi này khiến setto được triển khai cho Boolean-ArgumentMarshaler, nên tôi đã xóa phương thức setBoolean khỏi lớp ArgumentMarshalerbase.

Notice that the abstract setfunction takes a Stringargument, but the implementation in the BooleanArgumentMarshallerdoes not use it. I put that argument in there because I knew that the StringArgumentMarshaller and IntegerArgumentMarshaller*would* use it.  
Lưu ý rằng hàm tập hợp trừu tượng nhận một Đối số chuỗi, nhưng việc triển khai trong BooleanArgumentMarshaller không sử dụng nó. Tôi đặt đối số đó ở đó vì tôi biết rằng StringArgumentMarshaller và IntegerArgumentMarshaller sẽ sử dụng nó.

<https://www.facebook.com/groups/phattrienlaptrinh>

218 **Chapter 14: Successive Reﬁnement**  
218 Chương 14: Tinh Luyện Liên Tiếp

Next, I wanted to deploy the getmethod into BooleanArgumentMarshaler. Deploying getfunctions is always ugly because the return type has to be Object, and in this case needs to be cast to a Boolean.  
Tiếp theo, tôi muốn triển khai getmethod vào BooleanArgumentMarshaler. Việc triển khai các hàm get luôn xấu vì kiểu trả về phải là Đối tượng và trong trường hợp này cần phải chuyển thành Boolean.

public boolean getBoolean(char arg) { Args.ArgumentMarshaler am = booleanArgs.get(arg); return am != null && **(Boolean)**am.**get**();

}

Just to get this to compile, I added the get function to the ArgumentMarshaler.  
Để biên dịch cái này, tôi đã thêm hàm get vào ArgumentMarshaler.

private abstract class ArgumentMarshaler { ...

**public Object get() { return null;**

**}** }

This compiled and obviously failed the tests. Getting the tests working again was simply a matter of making get abstract and implementing it in BooleanAgumentMarshaler.  
Điều này được tổng hợp và rõ ràng là đã thất bại trong các bài kiểm tra. Làm cho các bài kiểm tra hoạt động trở lại đơn giản chỉ là vấn đề tạo bản tóm tắt và triển khai nó trong BooleanAgumentMarshaler.

private abstract class ArgumentMarshaler { protected boolean booleanValue = false; ...

public **abstract** Object get(); }

private class BooleanArgumentMarshaler extends ArgumentMarshaler { public void set(String s) {

booleanValue = true; }

**public Object get() { return booleanValue;**

**}** }

Once again the tests passed. So both getand setdeploy to the BooleanArgumentMarshaler! This allowed me to remove the old getBooleanfunction from ArgumentMarshaler, move the protected booleanValue variable down to BooleanArgumentMarshaler, and make it private.  
Một lần nữa các bài kiểm tra trôi qua. Vì vậy, cả get và setdeploy tới BooleanArgumentMarshaler! Điều này cho phép tôi xóa hàm getBoolean cũ khỏi ArgumentMarshaler, di chuyển biến booleanValue được bảo vệ xuống BooleanArgumentMarshaler và đặt nó ở chế độ riêng tư.

I did the same pattern of changes for Strings. I deployed both setand get, deleted the unused functions, and moved the variables.  
Tôi đã thực hiện mô hình thay đổi tương tự cho Chuỗi. Tôi đã triển khai cả setand get, xóa các hàm không sử dụng và di chuyển các biến.

private void setStringArg(char argChar) throws ArgsException { currentArgument++;

try { stringArgs.get(argChar).**set**(args[currentArgument]);

} catch (ArrayIndexOutOfBoundsException e) { valid = false;

errorArgumentId = argChar;

errorCode = ErrorCode.MISSING\_STRING; throw new ArgsException();

} }

<https://www.facebook.com/groups/phattrienlaptrinh>

**String Arguments** 219  
Đối số chuỗi 219

...

public String getString(char arg) { Args.ArgumentMarshaler am = stringArgs.get(arg); return am == null ? "" : **(String)** am.**get**();

} ...

private abstract class ArgumentMarshaler { private int integerValue;

public void setInteger(int i) { integerValue = i;

}

public int getInteger() { return integerValue;

}

public abstract void set(String s);

public abstract Object get(); }

private class BooleanArgumentMarshaler extends ArgumentMarshaler { **private boolean booleanValue = false;**

public void set(String s) { booleanValue = true;

}

public Object get() { return booleanValue;

} }

private class StringArgumentMarshaler extends ArgumentMarshaler { **private String stringValue = "";**

public void set(String s) { **stringValue = s;**

}

public Object get() { **return stringValue;**

} }

private class IntegerArgumentMarshaler extends ArgumentMarshaler {

public void set(String s) {

}

public Object get() { return null;

} }

}

<https://www.facebook.com/groups/phattrienlaptrinh>

220 **Chapter 14: Successive Reﬁnement**  
220 Chương 14: Tinh luyện liên tiếp

Finally, I repeated the process for integers. This was just a little more complicated because integersneeded to be parsed, and the parseoperation can throw an exception. But the result is better because the whole concept of NumberFormatExceptiongot buried in the IntegerArgumentMarshaler.  
Cuối cùng, tôi lặp lại quy trình cho số nguyên. Điều này phức tạp hơn một chút vì các số nguyên cần được phân tích cú pháp và thao tác phân tích cú pháp có thể đưa ra một ngoại lệ. Nhưng kết quả tốt hơn vì toàn bộ khái niệm về NumberFormatException đã bị chôn vùi trong IntegerArgumentMarshaler.

private boolean isIntArg(char argChar) {return intArgs.containsKey(argChar);}

private void setIntArg(char argChar) throws ArgsException { currentArgument++;

String parameter = null; try {

parameter = args[currentArgument]; intArgs.get(argChar).**set**(parameter);

} catch (ArrayIndexOutOfBoundsException e) { valid = false;

errorArgumentId = argChar;

errorCode = ErrorCode.MISSING\_INTEGER; throw new ArgsException();

} catch (**ArgsException** e) { valid = false; errorArgumentId = argChar; errorParameter = parameter;

errorCode = ErrorCode.INVALID\_INTEGER; throw **e**;

} }

...

private void setBooleanArg(char argChar) { **try {**

booleanArgs.get(argChar).set("true"); **} catch (ArgsException e) {**

**}** }

...

public int getInt(char arg) { Args.ArgumentMarshaler am = intArgs.get(arg); return am == null ? 0 : **(Integer)** am.**get**();

} ...

private abstract class ArgumentMarshaler {

public abstract void set(String s) throws ArgsException; public abstract Object get();

} ...

private class IntegerArgumentMarshaler extends ArgumentMarshaler { **private int intValue = 0;**

public void set(String s) **throws ArgsException { try {**

**intValue = Integer.parseInt(s); } catch (NumberFormatException e) {**

**throw new ArgsException(); }**

}

public Object get() { **return intValue;**

} }

<https://www.facebook.com/groups/phattrienlaptrinh>

**String Arguments** 221  
Đối số chuỗi 221

Of course, the tests continued to pass. Next, I got rid of the three different maps up at the top of the algorithm. This made the whole system much more generic. However, I couldn’t get rid of them just by deleting them because that would break the system. Instead, I added a new Mapfor the ArgumentMarshalerand then one by one changed the methods to use it instead of the three original maps.  
Tất nhiên, các bài kiểm tra tiếp tục vượt qua. Tiếp theo, tôi loại bỏ ba bản đồ khác nhau ở đầu thuật toán. Điều này làm cho toàn bộ hệ thống chung chung hơn nhiều. Tuy nhiên, tôi không thể loại bỏ chúng chỉ bằng cách xóa chúng vì điều đó sẽ phá vỡ hệ thống. Thay vào đó, tôi đã thêm một Bản đồ mới cho Đối số Marshaler và sau đó lần lượt thay đổi các phương pháp để sử dụng nó thay vì ba bản đồ ban đầu.

public class Args { ...

private Map<Character, ArgumentMarshaler> booleanArgs = new HashMap<Character, ArgumentMarshaler>();

private Map<Character, ArgumentMarshaler> stringArgs = new HashMap<Character, ArgumentMarshaler>();

private Map<Character, ArgumentMarshaler> intArgs = new HashMap<Character, ArgumentMarshaler>();

**private Map<Character, ArgumentMarshaler> marshalers = new HashMap<Character, ArgumentMarshaler>();**

...

private void parseBooleanSchemaElement(char elementId) { **ArgumentMarshaler m = new BooleanArgumentMarshaler();** booleanArgs.put(elementId, m); **marshalers.put(elementId, m);**

}

private void parseIntegerSchemaElement(char elementId) { **ArgumentMarshaler m = new IntegerArgumentMarshaler();** intArgs.put(elementId, m);

**marshalers.put(elementId, m);** }

private void parseStringSchemaElement(char elementId) { **ArgumentMarshaler m = new StringArgumentMarshaler();** stringArgs.put(elementId, m); **marshalers.put(elementId, m);**

}

Of course the tests all still passed. Next, I changed isBooleanArg from this:  
Tất nhiên các bài kiểm tra tất cả vẫn vượt qua. Tiếp theo, tôi đã thay đổi isBooleanArg từ đây:

private boolean isBooleanArg(char argChar) { return booleanArgs.containsKey(argChar);

}

to this:  
để này:

private boolean isBooleanArg(char argChar) { **ArgumentMarshaler m = marshalers.get(argChar); return m instanceof BooleanArgumentMarshaler;**

}

The tests still passed. So I made the same change to isIntArg and isStringArg.  
Các bài kiểm tra vẫn trôi qua. Vì vậy, tôi đã thực hiện thay đổi tương tự đối với isIntArg và isStringArg.

private boolean isIntArg(char argChar) { **ArgumentMarshaler m = marshalers.get(argChar); return m instanceof IntegerArgumentMarshaler;**

}

private boolean isStringArg(char argChar) { **ArgumentMarshaler m = marshalers.get(argChar); return m instanceof StringArgumentMarshaler;**

}

<https://www.facebook.com/groups/phattrienlaptrinh>

222 **Chapter 14: Successive Reﬁnement**  
222 Chương 14: Tinh luyện liên tiếp

The tests still passed. So I eliminated all the duplicate calls to marshalers.get as follows:  
Các bài kiểm tra vẫn trôi qua. Vì vậy, tôi đã loại bỏ tất cả các cuộc gọi trùng lặp tới marshalers.get như sau:

private boolean setArgument(char argChar) throws ArgsException { **ArgumentMarshaler m = marshalers.get(argChar);**

if (isBooleanArg(**m**)) setBooleanArg(argChar);

else if (isStringArg(**m**)) setStringArg(argChar);

else if (isIntArg(**m**)) setIntArg(argChar);

else

return false;

return true; }

private boolean isIntArg(**ArgumentMarshaler m**) { return m instanceof IntegerArgumentMarshaler;

}

private boolean isStringArg(**ArgumentMarshaler m**) { return m instanceof StringArgumentMarshaler;

}

private boolean isBooleanArg(**ArgumentMarshaler m**) { return m instanceof BooleanArgumentMarshaler;

}

This left no good reason for the three isxxxArg methods. So I inlined them:

private boolean setArgument(char argChar) throws ArgsException { ArgumentMarshaler m = marshalers.get(argChar);

if (**m instanceof BooleanArgumentMarshaler**) setBooleanArg(argChar);

else if (**m instanceof StringArgumentMarshaler**) setStringArg(argChar);

else if (**m instanceof IntegerArgumentMarshaler**) setIntArg(argChar);

else

return false;

return true; }

Next, I started using the marshalers map in the set functions, breaking the use of the other three maps. I started with the booleans.  
Tiếp theo, tôi bắt đầu sử dụng bản đồ soái ca trong các chức năng đã thiết lập, bỏ qua việc sử dụng ba bản đồ còn lại. Tôi bắt đầu với các booleans.

private boolean setArgument(char argChar) throws ArgsException { ArgumentMarshaler m = marshalers.get(argChar);

if (m instanceof BooleanArgumentMarshaler) setBooleanArg(**m**);

else if (m instanceof StringArgumentMarshaler) setStringArg(argChar);

else if (m instanceof IntegerArgumentMarshaler) setIntArg(argChar);

else

return false;

<https://www.facebook.com/groups/phattrienlaptrinh>

**String Arguments** 223  
Đối số chuỗi 223

return true; }

...

private void setBooleanArg(**ArgumentMarshaler m**) { try {

**m**.set("true");**// was: booleanArgs.get(argChar).set("true");** } catch (ArgsException e) {

} }

The tests still passed, so I did the same with Stringsand Integers. This allowed me to inte-grate some of the ugly exception management code into the setArgument function.  
Các bài kiểm tra vẫn vượt qua, vì vậy tôi đã làm điều tương tự với Stringsand Integers. Điều này cho phép tôi tích hợp một số mã quản lý ngoại lệ xấu xí vào hàm setArgument.

private boolean setArgument(char argChar) throws ArgsException { ArgumentMarshaler m = marshalers.get(argChar);

**try {**

if (m instanceof BooleanArgumentMarshaler) setBooleanArg(m);

else if (m instanceof StringArgumentMarshaler) setStringArg(**m**);

else if (m instanceof IntegerArgumentMarshaler) setIntArg(**m**);

else

return false;

**} catch (ArgsException e) { valid = false; errorArgumentId = argChar; throw e;**

**}**

return true; }

private void setIntArg(**ArgumentMarshaler m**) throws ArgsException { currentArgument++;

String parameter = null; try {

parameter = args[currentArgument]; **m**.set(parameter);

} catch (ArrayIndexOutOfBoundsException e) { errorCode = ErrorCode.MISSING\_INTEGER; throw new ArgsException();

} catch (ArgsException e) { errorParameter = parameter;

errorCode = ErrorCode.INVALID\_INTEGER; throw e;

} }

private void setStringArg(**ArgumentMarshaler m**) throws ArgsException { currentArgument++;

try { **m**.set(args[currentArgument]);

} catch (ArrayIndexOutOfBoundsException e) { errorCode = ErrorCode.MISSING\_STRING; throw new ArgsException();

} }

<https://www.facebook.com/groups/phattrienlaptrinh>

224 **Chapter 14: Successive Reﬁnement**  
224 Chương 14: Tinh luyện liên tiếp

I was close to being able to remove the three old maps. First, I needed to change the getBoolean function from this:  
Tôi gần như có thể xóa ba bản đồ cũ. Đầu tiên, tôi cần thay đổi hàm getBoolean từ đây:

public boolean getBoolean(char arg) { Args.ArgumentMarshaler am = booleanArgs.get(arg); return am != null && (Boolean) am.get();

}

to this:  
để này:

public boolean getBoolean(char arg) { Args.ArgumentMarshaler am = marshalers.get(arg); **boolean b = false;**

**try {**

**b =** am != null && (Boolean) am.get(); **} catch (ClassCastException e) {**

**b = false; }**

**return b;** }

This last change might have been a surprise. Why did I suddenly decide to deal with the ClassCastException? The reason is that I have a set of unit tests and a separate set of acceptance tests written in FitNesse. It turns out that the FitNesse tests made sure that if you called getBooleanon a nonboolean argument, you got a false. The unit tests did not. Up to this point I had only been running the unit tests.2  
Thay đổi cuối cùng này có thể là một bất ngờ. Tại sao tôi đột nhiên quyết định xử lý ClassCastException? Lý do là tôi có một bộ bài kiểm tra đơn vị và một bộ bài kiểm tra chấp nhận riêng biệt được viết bằng FitNesse. Hóa ra là các bài kiểm tra của FitNesse đã đảm bảo rằng nếu bạn gọi getBooleanon là một đối số không phải là boolean, thì bạn đã sai. Các bài kiểm tra đơn vị đã không. Cho đến thời điểm này, tôi mới chỉ chạy các bài kiểm tra đơn vị.2

This last change allowed me to pull out another use of the boolean map:  
Thay đổi cuối cùng này cho phép tôi rút ra một cách sử dụng khác của bản đồ boolean:

private void parseBooleanSchemaElement(char elementId) { ArgumentMarshaler m = new BooleanArgumentMarshaler();

**~~booleanArgs.put(elementId, m);~~** marshalers.put(elementId, m);

}

And now we can delete the boolean map.  
Và bây giờ chúng ta có thể xóa bản đồ boolean.

public class Args { ...

**~~private Map<Character, ArgumentMarshaler> booleanArgs =~~ ~~new HashMap<Character, ArgumentMarshaler>();~~**

private Map<Character, ArgumentMarshaler> stringArgs = new HashMap<Character, ArgumentMarshaler>();

private Map<Character, ArgumentMarshaler> intArgs = new HashMap<Character, ArgumentMarshaler>();

private Map<Character, ArgumentMarshaler> marshalers = new HashMap<Character, ArgumentMarshaler>();

...

Next, I migrated the Stringand Integerarguments in the same manner and did a little cleanup with the booleans.  
Tiếp theo, tôi di chuyển Đối số chuỗi và số nguyên theo cách tương tự và dọn dẹp một chút với các phép toán luận.

private void parseBooleanSchemaElement(char elementId) { marshalers.put(elementId, **new BooleanArgumentMarshaler()**);

}

2. To prevent further surprises of this kind, I added a new unit test that invoked all the FitNesse tests.  
2. Để tránh những bất ngờ tiếp theo thuộc loại này, tôi đã thêm một bài kiểm tra đơn vị mới gọi tất cả các bài kiểm tra FitNesse.

<https://www.facebook.com/groups/phattrienlaptrinh>

**String Arguments** 225  
Đối số chuỗi 225

private void parseIntegerSchemaElement(char elementId) { marshalers.put(elementId, **new IntegerArgumentMarshaler()**);

}

private void parseStringSchemaElement(char elementId) { marshalers.put(elementId, **new StringArgumentMarshaler()**);

} ...

public String getString(char arg) { Args.ArgumentMarshaler am = **marshalers**.get(arg); **try {**

return am == null ? "" : (String) am.get(); **} catch (ClassCastException e) {**

**return ""; }**

}

public int getInt(char arg) { Args.ArgumentMarshaler am = **marshalers**.get(arg); **try {**

return am == null ? 0 : (Integer) am.get(); **} catch (Exception e) {**

**return 0; }**

} ...

public class Args { ...

**~~private Map<Character, ArgumentMarshaler> stringArgs =~~ ~~new HashMap<Character, ArgumentMarshaler>();~~**

**~~private Map<Character, ArgumentMarshaler> intArgs =~~ ~~new HashMap<Character, ArgumentMarshaler>();~~**

private Map<Character, ArgumentMarshaler> marshalers = new HashMap<Character, ArgumentMarshaler>();

...

Next, I inlined the three parse methods because they didn’t do much anymore:  
Tiếp theo, tôi đã thêm vào ba phương thức phân tích cú pháp vì chúng không còn tác dụng nhiều nữa:

private void parseSchemaElement(String element) throws ParseException { char elementId = element.charAt(0);

String elementTail = element.substring(1); validateSchemaElementId(elementId);

if (isBooleanSchemaElement(elementTail)) **marshalers.put(elementId, new BooleanArgumentMarshaler());**

else if (isStringSchemaElement(elementTail)) **marshalers.put(elementId, new StringArgumentMarshaler());**

else if (isIntegerSchemaElement(elementTail)) { **marshalers.put(elementId, new IntegerArgumentMarshaler());**

} else {

throw new ParseException(String.format(

"Argument: %c has invalid format: %s.", elementId, elementTail), 0); }

}

Okay, so now let’s look at the whole picture again. Listing 14-12 shows the current form of the Args class.  
Được rồi, vậy bây giờ chúng ta hãy nhìn vào toàn bộ bức tranh một lần nữa. Liệt kê 14-12 cho thấy hình thức hiện tại của lớp Args.

<https://www.facebook.com/groups/phattrienlaptrinh>

226 **Chapter 14: Successive Reﬁnement**  
226 Chương 14: Tinh luyện liên tiếp

**Listing 14-12**

**Args.java (After first refactoring)** package com.objectmentor.utilities.getopts;

import java.text.ParseException; import java.util.\*;

public class Args { private String schema; private String[] args;

private boolean valid = true;

private Set<Character> unexpectedArguments = new TreeSet<Character>(); private Map<Character, ArgumentMarshaler> marshalers =

new HashMap<Character, ArgumentMarshaler>();

private Set<Character> argsFound = new HashSet<Character>(); private int currentArgument;

private char errorArgumentId = '\0'; private String errorParameter = "TILT"; private ErrorCode errorCode = ErrorCode.OK;

private enum ErrorCode {

OK, MISSING\_STRING, MISSING\_INTEGER, INVALID\_INTEGER, UNEXPECTED\_ARGUMENT}

public Args(String schema, String[] args) throws ParseException { this.schema = schema;

this.args = args; valid = parse();

}

private boolean parse() throws ParseException { if (schema.length() == 0 && args.length == 0)

return true; parseSchema(); try {

parseArguments();

} catch (ArgsException e) { }

return valid; }

private boolean parseSchema() throws ParseException { for (String element : schema.split(",")) {

if (element.length() > 0) {

String trimmedElement = element.trim(); parseSchemaElement(trimmedElement);

} }

return true; }

private void parseSchemaElement(String element) throws ParseException { char elementId = element.charAt(0);

String elementTail = element.substring(1); validateSchemaElementId(elementId);

if (isBooleanSchemaElement(elementTail)) marshalers.put(elementId, new BooleanArgumentMarshaler());

else if (isStringSchemaElement(elementTail)) marshalers.put(elementId, new StringArgumentMarshaler());

<https://www.facebook.com/groups/phattrienlaptrinh>

**String Arguments** 227  
Đối số chuỗi 227

**Listing 14-12 (continued)**

**Args.java (After first refactoring)  
Args.java (Sau lần tái cấu trúc đầu tiên)**

else if (isIntegerSchemaElement(elementTail)) { marshalers.put(elementId, new IntegerArgumentMarshaler());

} else {

throw new ParseException(String.format(

"Argument: %c has invalid format: %s.", elementId, elementTail), 0); }

}

private void validateSchemaElementId(char elementId) throws ParseException { if (!Character.isLetter(elementId)) {

throw new ParseException(

"Bad character:" + elementId + "in Args format: " + schema, 0); }

}

private boolean isStringSchemaElement(String elementTail) { return elementTail.equals("\*");

}

private boolean isBooleanSchemaElement(String elementTail) { return elementTail.length() == 0;

}

private boolean isIntegerSchemaElement(String elementTail) { return elementTail.equals("#");

}

private boolean parseArguments() throws ArgsException {

for (currentArgument=0; currentArgument<args.length; currentArgument++) { String arg = args[currentArgument];

parseArgument(arg); }

return true; }

private void parseArgument(String arg) throws ArgsException { if (arg.startsWith("-"))

parseElements(arg); }

private void parseElements(String arg) throws ArgsException { for (int i = 1; i < arg.length(); i++)

parseElement(arg.charAt(i)); }

private void parseElement(char argChar) throws ArgsException { if (setArgument(argChar))

argsFound.add(argChar); else {

unexpectedArguments.add(argChar); errorCode = ErrorCode.UNEXPECTED\_ARGUMENT; valid = false;

} }

<https://www.facebook.com/groups/phattrienlaptrinh>

228 **Chapter 14: Successive Reﬁnement**  
228 Chương 14: Tinh Luyện Liên Tiếp

**Listing 14-12 (continued)**  
Liệt kê 14-12 (tiếp theo)

**Args.java (After first refactoring)  
Args.java (Sau lần tái cấu trúc đầu tiên)**

private boolean setArgument(char argChar) throws ArgsException { ArgumentMarshaler m = marshalers.get(argChar);

try {

if (m instanceof BooleanArgumentMarshaler) setBooleanArg(m);

else if (m instanceof StringArgumentMarshaler) setStringArg(m);

else if (m instanceof IntegerArgumentMarshaler) setIntArg(m);

else

return false;

} catch (ArgsException e) { valid = false; errorArgumentId = argChar; throw e;

}

return true; }

private void setIntArg(ArgumentMarshaler m) throws ArgsException { currentArgument++;

String parameter = null; try {

parameter = args[currentArgument]; m.set(parameter);

} catch (ArrayIndexOutOfBoundsException e) { errorCode = ErrorCode.MISSING\_INTEGER; throw new ArgsException();

} catch (ArgsException e) { errorParameter = parameter;

errorCode = ErrorCode.INVALID\_INTEGER; throw e;

} }

private void setStringArg(ArgumentMarshaler m) throws ArgsException { currentArgument++;

try { m.set(args[currentArgument]);

} catch (ArrayIndexOutOfBoundsException e) { errorCode = ErrorCode.MISSING\_STRING; throw new ArgsException();

} }

private void setBooleanArg(ArgumentMarshaler m) { try {

m.set("true");

} catch (ArgsException e) { }

}

public int cardinality() { return argsFound.size();

}

<https://www.facebook.com/groups/phattrienlaptrinh>

**String Arguments** 229  
Đối số chuỗi 229

**Listing 14-12 (continued)**

**Args.java (After first refactoring)  
Args.java (Sau lần tái cấu trúc đầu tiên)**

public String usage() { if (schema.length() > 0)

return "-[" + schema + "]"; else

return ""; }

public String errorMessage() throws Exception { switch (errorCode) {

case OK:

throw new Exception("TILT: Should not get here."); case UNEXPECTED\_ARGUMENT:

return unexpectedArgumentMessage(); case MISSING\_STRING:

return String.format("Could not find string parameter for -%c.", errorArgumentId);

case INVALID\_INTEGER:

return String.format("Argument -%c expects an integer but was '%s'.", errorArgumentId, errorParameter);

case MISSING\_INTEGER:

return String.format("Could not find integer parameter for -%c.", errorArgumentId);

}

return ""; }

private String unexpectedArgumentMessage() {

StringBuffer message = new StringBuffer("Argument(s) -"); for (char c : unexpectedArguments) {

message.append(c); }

message.append(" unexpected.");

return message.toString(); }

public boolean getBoolean(char arg) { Args.ArgumentMarshaler am = marshalers.get(arg); boolean b = false;

try {

b = am != null && (Boolean) am.get(); } catch (ClassCastException e) {

b = false; }

return b; }

public String getString(char arg) { Args.ArgumentMarshaler am = marshalers.get(arg); try {

return am == null ? "" : (String) am.get(); } catch (ClassCastException e) {

return ""; }

}

<https://www.facebook.com/groups/phattrienlaptrinh>

230 **Chapter 14: Successive Reﬁnement**  
230 Chương 14: Tinh luyện liên tiếp

**Listing 14-12 (continued)**  
Liệt kê 14-12 (tiếp theo)

**Args.java (After first refactoring)  
Args.java (Sau lần tái cấu trúc đầu tiên)**

public int getInt(char arg) { Args.ArgumentMarshaler am = marshalers.get(arg); try {

return am == null ? 0 : (Integer) am.get(); } catch (Exception e) {

return 0; }

}

public boolean has(char arg) { return argsFound.contains(arg);

}

public boolean isValid() { return valid;

}

private class ArgsException extends Exception { }

private abstract class ArgumentMarshaler {

public abstract void set(String s) throws ArgsException; public abstract Object get();

}

private class BooleanArgumentMarshaler extends ArgumentMarshaler { private boolean booleanValue = false;

public void set(String s) { booleanValue = true;

}

public Object get() { return booleanValue;

} }

private class StringArgumentMarshaler extends ArgumentMarshaler { private String stringValue = "";

public void set(String s) { stringValue = s;

}

public Object get() { return stringValue;

} }

private class IntegerArgumentMarshaler extends ArgumentMarshaler { private int intValue = 0;

public void set(String s) throws ArgsException { try {

intValue = Integer.parseInt(s);

<https://www.facebook.com/groups/phattrienlaptrinh>

**String Arguments** 231  
Đối số chuỗi 231

**Listing 14-12 (continued)**

**Args.java (After first refactoring)**

} catch (NumberFormatException e) { throw new ArgsException();

} }

public Object get() { return intValue;

} }

}

After all that work, this is a bit disappointing. The structure is a bit better, but we still have all those variables up at the top; there’s still a horrible type-case in setArgument; and all those setfunctions are really ugly. Not to mention all the error processing. We still have a lot of work ahead of us.  
Sau tất cả những công việc đó, điều này là một chút thất vọng. Cấu trúc tốt hơn một chút, nhưng chúng tôi vẫn có tất cả các biến đó ở trên cùng; vẫn còn một trường hợp kiểu khủng khiếp trong setArgument; và tất cả các hàm thiết lập đó thực sự xấu xí. Chưa kể tất cả các xử lý lỗi. Chúng tôi vẫn còn rất nhiều công việc phía trước.

I’d really like to get rid of that type-case up in setArgument [G23]. What I’d like in setArgument is a single call to ArgumentMarshaler.set. This means I need to push setIntArg, setStringArg, and setBooleanArgdown into the appropriate ArgumentMarshaler derivatives. But there is a problem.  
Tôi thực sự muốn loại bỏ trường hợp kiểu đó trong setArgument [G23]. Điều tôi muốn trong setArgument là một cuộc gọi duy nhất tới ArgumentMarshaler.set. Điều này có nghĩa là tôi cần đẩy setIntArg, setStringArg và setBooleanArgdown vào các dẫn xuất ArgumentMarshaler thích hợp. Nhưng có một vấn đề.

If you look closely at setIntArg, you’ll notice that it uses two instance variables: args and currentArg. To move setIntArgdown into BooleanArgumentMarshaler, I’ll have to pass both argsand currentArgsas function arguments. That’s dirty [F1]. I’d rather pass one argument instead of two. Fortunately, there is a simple solution. We can convert the args array into a listand pass an Iteratordown to the setfunctions. The following took me ten steps, passing all the tests after each. But I’ll just show you the result. You should be able to ﬁgure out what most of the tiny little steps were.  
Nếu quan sát kỹ setIntArg, bạn sẽ nhận thấy rằng nó sử dụng hai biến thể hiện: args và currentArg. Để di chuyển setIntArgdown thành BooleanArgumentMarshaler, tôi sẽ phải chuyển cả hai đối số hàm args và currentArgsas. Thật là bẩn [F1]. Tôi muốn vượt qua một đối số thay vì hai đối số. May mắn thay, có một giải pháp đơn giản. Chúng ta có thể chuyển đổi mảng args thành một danh sách và chuyển một Iteratordown cho các hàm set. Sau đây tôi đã đi được mười bước, vượt qua tất cả các bài kiểm tra sau mỗi bước. Nhưng tôi sẽ chỉ cho bạn thấy kết quả. Bạn sẽ có thể tìm ra hầu hết các bước nhỏ là gì.

public class Args { private String schema;

**~~private String[] args;~~** private boolean valid = true;

private Set<Character> unexpectedArguments = new TreeSet<Character>(); private Map<Character, ArgumentMarshaler> marshalers =

new HashMap<Character, ArgumentMarshaler>();

private Set<Character> argsFound = new HashSet<Character>(); private **Iterator<String>** currentArgument;

private char errorArgumentId = '\0'; private String errorParameter = "TILT"; private ErrorCode errorCode = ErrorCode.OK; **private List<String> argsList;**

private enum ErrorCode {

OK, MISSING\_STRING, MISSING\_INTEGER, INVALID\_INTEGER, UNEXPECTED\_ARGUMENT}

public Args(String schema, String[] args) throws ParseException { this.schema = schema;

**argsList = Arrays.asList(args);** valid = parse();

}

<https://www.facebook.com/groups/phattrienlaptrinh>

232 **Chapter 14: Successive Reﬁnement**  
232 Chương 14: Tinh luyện liên tiếp

private boolean parse() throws ParseException {

if (schema.length() == 0 && **argsList.size()** == 0) return true;

parseSchema(); try {

parseArguments();

} catch (ArgsException e) { }

return valid; }

---

private boolean parseArguments() throws ArgsException {

for (currentArgument = **argsList.iterator()**; currentArgument.**hasNext()**;) { String arg = currentArgument.**next()**;

parseArgument(arg); }

return true; }

---

private void setIntArg(ArgumentMarshaler m) throws ArgsException { String parameter = null;

try {

parameter = currentArgument.**next()**; m.set(parameter);

} catch (**NoSuchElementException** e) { errorCode = ErrorCode.MISSING\_INTEGER; throw new ArgsException();

} catch (ArgsException e) { errorParameter = parameter;

errorCode = ErrorCode.INVALID\_INTEGER; throw e;

} }

private void setStringArg(ArgumentMarshaler m) throws ArgsException { try {

m.set(currentArgument**.next()**);

} catch (**NoSuchElementException** e) { errorCode = ErrorCode.MISSING\_STRING; throw new ArgsException();

} }

These were simple changes that kept all the tests passing. Now we can start moving the set functions down into the appropriate derivatives. First, I need to make the following change in setArgument:  
Đây là những thay đổi đơn giản giúp tất cả các bài kiểm tra đều vượt qua. Bây giờ chúng ta có thể bắt đầu chuyển các hàm tập hợp xuống các đạo hàm thích hợp. Trước tiên, tôi cần thực hiện thay đổi sau trong setArgument:

private boolean setArgument(char argChar) throws ArgsException { ArgumentMarshaler m = marshalers.get(argChar);

**if (m == null) return false;**

try {

if (m instanceof BooleanArgumentMarshaler) setBooleanArg(m);

else if (m instanceof StringArgumentMarshaler) setStringArg(m);

else if (m instanceof IntegerArgumentMarshaler) setIntArg(m);

<https://www.facebook.com/groups/phattrienlaptrinh>

**String Arguments** 233  
Đối số chuỗi 233

**~~else~~**

**~~return false;~~**

} catch (ArgsException e) { valid = false; errorArgumentId = argChar; throw e;

}

return true; }

This change is important because we want to completely eliminate the if-elsechain. Therefore, we needed to get the error condition out of it.  
Thay đổi này rất quan trọng vì chúng tôi muốn loại bỏ hoàn toàn if-elsechain. Do đó, chúng tôi cần loại bỏ tình trạng lỗi ra khỏi nó.

Now we can start to move the setfunctions. The setBooleanArgfunction is trivial, so we’ll prepare that one ﬁrst. Our goal is to change the setBooleanArgfunction to simply for-ward to the BooleanArgumentMarshaler.  
Bây giờ chúng ta có thể bắt đầu di chuyển các setfunctions. Hàm setBooleanArg không quan trọng, vì vậy chúng ta sẽ chuẩn bị hàm đó trước. Mục tiêu của chúng ta là thay đổi hàm setBooleanArg thành chuyển tiếp đơn giản tới BooleanArgumentMarshaler.

private boolean setArgument(char argChar) throws ArgsException { ArgumentMarshaler m = marshalers.get(argChar);

if (m == null) return false;

try {

if (m instanceof BooleanArgumentMarshaler) setBooleanArg(m, **currentArgument**);

else if (m instanceof StringArgumentMarshaler) setStringArg(m);

else if (m instanceof IntegerArgumentMarshaler) setIntArg(m);

} catch (ArgsException e) { valid = false; errorArgumentId = argChar; throw e;

}

return true; }

---

private void setBooleanArg(ArgumentMarshaler m, **Iterator<String> currentArgument)** throws ArgsException {

**~~try {~~** m.set("true");

**~~catch (ArgsException e) {~~**

**~~}~~** }

Didn’t we just put that exception processing in? Putting things in so you can take them out again is pretty common in refactoring. The smallness of the steps and the need to keep the tests running means that you move things around a lot. Refactoring is a lot like solving a Rubik’s cube. There are lots of little steps required to achieve a large goal. Each step enables the next.  
Không phải chúng ta vừa đưa quá trình xử lý ngoại lệ đó vào sao? Đưa mọi thứ vào để bạn có thể lấy chúng ra lần nữa là điều khá phổ biến trong quá trình tái cấu trúc. Độ nhỏ của các bước và nhu cầu duy trì các bài kiểm tra chạy đồng nghĩa với việc bạn phải di chuyển mọi thứ xung quanh rất nhiều. Tái cấu trúc rất giống với việc giải khối Rubik. Có rất nhiều bước nhỏ cần thiết để đạt được một mục tiêu lớn. Mỗi bước cho phép tiếp theo.

Why did we pass that iteratorwhen setBooleanArgcertainly doesn’t need it? Because setIntArgand setStringArgwill! And because I want to deploy all three of these functions through an abstract method in ArgumentMarshaller, I need to pass it to setBooleanArg.  
Tại sao chúng ta chuyển iterator đó khi setBooleanArg chắc chắn không cần nó? Bởi vì setIntArgand setStringArgwill! Và vì tôi muốn triển khai cả ba hàm này thông qua một phương thức trừu tượng trong ArgumentMarshaller, nên tôi cần chuyển nó tới setBooleanArg.

<https://www.facebook.com/groups/phattrienlaptrinh>

234 **Chapter 14: Successive Reﬁnement**  
234 Chương 14: Tinh luyện liên tiếp

So now setBooleanArgis useless. If there were a setfunction in ArgumentMarshaler, we could call it directly. So it’s time to make that function! The ﬁrst step is to add the new abstract method to ArgumentMarshaler.  
Vì vậy, bây giờ setBooleanArgis vô dụng. Nếu có một setfunction trong ArgumentMarshaler, chúng ta có thể gọi nó trực tiếp. Vì vậy, đã đến lúc thực hiện chức năng đó! Bước đầu tiên là thêm phương thức trừu tượng mới vào ArgumentMarshaler.

private abstract class ArgumentMarshaler {

**public abstract void set(Iterator<String> currentArgument) throws ArgsException;**

public abstract void set(String s) throws ArgsException; public abstract Object get();

}

Of course this breaks all the derivatives. So let’s implement the new method in each.  
Tất nhiên điều này phá vỡ tất cả các công cụ phái sinh. Vì vậy, hãy triển khai phương thức mới trong mỗi phương thức.

private class BooleanArgumentMarshaler extends ArgumentMarshaler { private boolean booleanValue = false;

**public void set(Iterator<String> currentArgument) throws ArgsException { booleanValue = true;**

}

public void set(String s) {

**~~booleanValue = true;~~** }

public Object get() { return booleanValue;

} }

private class StringArgumentMarshaler extends ArgumentMarshaler { private String stringValue = "";

**public void set(Iterator<String> currentArgument) throws ArgsException { }**

public void set(String s) { stringValue = s;

}

public Object get() { return stringValue;

} }

private class IntegerArgumentMarshaler extends ArgumentMarshaler { private int intValue = 0;

**public void set(Iterator<String> currentArgument) throws ArgsException { }**

public void set(String s) throws ArgsException { try {

intValue = Integer.parseInt(s); } catch (NumberFormatException e) {

throw new ArgsException(); }

}

<https://www.facebook.com/groups/phattrienlaptrinh>

**String Arguments** 235  
Đối số chuỗi 235

public Object get() { return intValue;

} }

And now we can eliminate setBooleanArg!  
Và bây giờ chúng ta có thể loại bỏ setBooleanArg!

private boolean setArgument(char argChar) throws ArgsException { ArgumentMarshaler m = marshalers.get(argChar);

if (m == null) return false;

try {

if (m instanceof BooleanArgumentMarshaler) **m.set**(currentArgument);

else if (m instanceof StringArgumentMarshaler) setStringArg(m);

else if (m instanceof IntegerArgumentMarshaler) setIntArg(m);

} catch (ArgsException e) { valid = false; errorArgumentId = argChar; throw e;

}

return true; }

The tests all pass, and the set function is deploying to BooleanArgumentMarshaler! Now we can do the same for Strings and Integers.  
Tất cả các bài kiểm tra đều vượt qua và chức năng thiết lập đang triển khai cho BooleanArgumentMarshaler! Bây giờ chúng ta có thể làm tương tự cho Chuỗi và Số nguyên.

private boolean setArgument(char argChar) throws ArgsException { ArgumentMarshaler m = marshalers.get(argChar);

if (m == null) return false;

try {

if (m instanceof BooleanArgumentMarshaler) m.set(currentArgument);

else if (m instanceof StringArgumentMarshaler) **m.set(currentArgument);**

else if (m instanceof IntegerArgumentMarshaler) **m.set(currentArgument);**

} catch (ArgsException e) { valid = false; errorArgumentId = argChar; throw e;

}

return true; }

---

private class StringArgumentMarshaler extends ArgumentMarshaler { private String stringValue = "";

public void set(Iterator<String> currentArgument) throws ArgsException { **try {**

**stringValue = currentArgument.next(); } catch (NoSuchElementException e) {**

**errorCode = ErrorCode.MISSING\_STRING;**

<https://www.facebook.com/groups/phattrienlaptrinh>

236 **Chapter 14: Successive Reﬁnement**  
236 Chương 14: Tinh luyện liên tiếp

**throw new ArgsException(); }**

}

public void set(String s) { }

public Object get() { return stringValue;

} }

private class IntegerArgumentMarshaler extends ArgumentMarshaler { private int intValue = 0;

public void set(Iterator<String> currentArgument) throws ArgsException { **String parameter = null;**

**try {**

**parameter = currentArgument.next(); set(parameter);**

**} catch (NoSuchElementException e) { errorCode = ErrorCode.MISSING\_INTEGER; throw new ArgsException();**

**} catch (ArgsException e) { errorParameter = parameter;**

**errorCode = ErrorCode.INVALID\_INTEGER; throw e;**

**}** }

public void set(String s) throws ArgsException { try {

intValue = Integer.parseInt(s); } catch (NumberFormatException e) {

throw new ArgsException(); }

}

public Object get() { return intValue;

} }

And so the *coup de grace*: The type-case can be removed! Touche!  
Và thế là cuộc đảo chính: Hộp chữ có thể được gỡ bỏ! Chạm vào!

private boolean setArgument(char argChar) throws ArgsException { ArgumentMarshaler m = marshalers.get(argChar);

if (m == null) return false;

try { m.set(currentArgument); return true;

} catch (ArgsException e) { valid = false; errorArgumentId = argChar; throw e;

} }

<https://www.facebook.com/groups/phattrienlaptrinh>

**String Arguments** 237  
Đối số chuỗi 237

Now we can get rid of some crufty functions in IntegerArgumentMarshalerand clean it up a bit.  
Bây giờ chúng ta có thể loại bỏ một số chức năng phức tạp trong IntegerArgumentMarshaler và làm sạch nó một chút.

private class IntegerArgumentMarshaler extends ArgumentMarshaler { private int intValue = 0

public void set(Iterator<String> currentArgument) throws ArgsException { String parameter = null;

try {

parameter = currentArgument.next(); **intValue = Integer.parseInt**(parameter);

} catch (NoSuchElementException e) { errorCode = ErrorCode.MISSING\_INTEGER; throw new ArgsException();

} catch (**NumberFormatException** e) { errorParameter = parameter;

errorCode = ErrorCode.INVALID\_INTEGER; throw new ArgsException();

} }

public Object get() { return intValue;

} }

We can also turn **ArgumentMarshaler** into an interface.  
Chúng ta cũng có thể biến ArgumentMarshaler thành một giao diện.

private **interface** ArgumentMarshaler {

void set(Iterator<String> currentArgument) throws ArgsException; Object get();

}

So now let’s see how easy it is to add a new argument type to our structure. It should require very few changes, and those changes should be isolated. First, we begin by adding a new test case to check that the double argument works correctly.  
Vì vậy, bây giờ hãy xem việc thêm một loại đối số mới vào cấu trúc của chúng ta dễ dàng như thế nào. Nó sẽ yêu cầu rất ít thay đổi và những thay đổi đó nên được tách biệt. Đầu tiên, chúng tôi bắt đầu bằng cách thêm một trường hợp thử nghiệm mới để kiểm tra xem đối số kép có hoạt động chính xác không.

public void testSimpleDoublePresent() throws Exception { Args args = new Args("x##", new String[] {"-x","42.3"}); assertTrue(args.isValid());

assertEquals(1, args.cardinality()); assertTrue(args.has('x'));

assertEquals(42.3, args.getDouble('x'), .001); }

Now we clean up the schema parsing code and add the ## detection for the double argument type.  
Bây giờ, chúng tôi dọn sạch mã phân tích lược đồ và thêm phát hiện ## cho loại đối số kép.

private void parseSchemaElement(String element) throws ParseException { char elementId = element.charAt(0);

String elementTail = element.substring(1); validateSchemaElementId(elementId);

if (elementTail.**length() == 0**)

marshalers.put(elementId, new BooleanArgumentMarshaler()); else if (elementTail.**equals("\*")**)

marshalers.put(elementId, new StringArgumentMarshaler()); else if (elementTail.**equals("#")**)

marshalers.put(elementId, new IntegerArgumentMarshaler());

<https://www.facebook.com/groups/phattrienlaptrinh>

238 **Chapter 14: Successive Reﬁnement**  
238 Chương 14: Tinh luyện liên tiếp

**else if (elementTail.equals("##")) marshalers.put(elementId, new DoubleArgumentMarshaler());**

else

throw new ParseException(String.format(

"Argument: %c has invalid format: %s.", elementId, elementTail), 0); }

Next, we write the DoubleArgumentMarshaler class.  
Tiếp theo, chúng ta viết lớp DoubleArgumentMarshaler.

**private class DoubleArgumentMarshaler implements ArgumentMarshaler { private double doubleValue = 0;**

**public void set(Iterator<String> currentArgument) throws ArgsException { String parameter = null;**

**try {**

**parameter = currentArgument.next(); doubleValue = Double.parseDouble(parameter);**

**} catch (NoSuchElementException e) { errorCode = ErrorCode.MISSING\_DOUBLE; throw new ArgsException();**

**} catch (NumberFormatException e) { errorParameter = parameter; errorCode = ErrorCode.INVALID\_DOUBLE; throw new ArgsException();**

**} }**

**public Object get() { return doubleValue;**

**} }**

This forces us to add a new ErrorCode.  
Điều này buộc chúng tôi phải thêm Mã lỗi mới.

private enum ErrorCode {

OK, MISSING\_STRING, MISSING\_INTEGER, INVALID\_INTEGER, UNEXPECTED\_ARGUMENT, **MISSING\_DOUBLE, INVALID\_DOUBLE**}

And we need a getDouble function.  
Và chúng ta cần một chức năng getDouble.

**public double getDouble(char arg) { Args.ArgumentMarshaler am = marshalers.get(arg); try {**

**return am == null ? 0 : (Double) am.get(); } catch (Exception e) {**

**return 0,0; }**

**}**

And all the tests pass! That was pretty painless. So now let’s make sure all the error processing works correctly. The next test case checks that an error is declared if an unparseable string is fed to a ## argument.  
Và tất cả các bài kiểm tra đều vượt qua! Đó là khá đau đớn. Vì vậy, bây giờ hãy đảm bảo rằng tất cả quá trình xử lý lỗi hoạt động chính xác. Trường hợp kiểm tra tiếp theo kiểm tra xem có lỗi nào được khai báo nếu một chuỗi không thể phân tích cú pháp được đưa vào đối số ## hay không.

public void testInvalidDouble() throws Exception {

Args args = new Args("x##", new String[] {"-x","Forty two"}); assertFalse(args.isValid());

assertEquals(0, args.cardinality()); assertFalse(args.has('x')); assertEquals(0, args.getInt('x'));

<https://www.facebook.com/groups/phattrienlaptrinh>

**String Arguments** 239  
Đối số chuỗi 239

assertEquals("Argument -x expects a double but was 'Forty two'.", args.errorMessage());

} ---

public String errorMessage() throws Exception { switch (errorCode) {

case OK:

throw new Exception("TILT: Should not get here."); case UNEXPECTED\_ARGUMENT:

return unexpectedArgumentMessage(); case MISSING\_STRING:

return String.format("Could not find string parameter for -%c.", errorArgumentId);

case INVALID\_INTEGER:

return String.format("Argument -%c expects an integer but was '%s'.", errorArgumentId, errorParameter);

case MISSING\_INTEGER:

return String.format("Could not find integer parameter for -%c.", errorArgumentId);

**case INVALID\_DOUBLE:**

**return String.format("Argument -%c expects a double but was '%s'.", errorArgumentId, errorParameter);**

**case MISSING\_DOUBLE:**

**return String.format("Could not find double parameter for -%c.", errorArgumentId);**

}

return ""; }

And the tests pass. The next test makes sure we detect a missing doubleargument properly.  
Và các bài kiểm tra vượt qua. Thử nghiệm tiếp theo đảm bảo rằng chúng tôi phát hiện chính xác một đối số kép bị thiếu.

**public void testMissingDouble() throws Exception { Args args = new Args("x##", new String[]{"-x"}); assertFalse(args.isValid());**

**assertEquals(0, args.cardinality()); assertFalse(args.has('x')); assertEquals(0.0, args.getDouble('x'), 0.01);**

**assertEquals("Could not find double parameter for -x.", args.errorMessage());**

**}**

This passes as expected. We wrote it simply for completeness.  
Điều này vượt qua như mong đợi. Chúng tôi đã viết nó chỉ đơn giản là cho đầy đủ.

The exception code is pretty ugly and doesn’t really belong in the Argsclass. We are also throwing out ParseException, which doesn’t really belong to us. So let’s merge all the exceptions into a single ArgsException class and move it into its own module.  
Mã ngoại lệ khá xấu và không thực sự thuộc về Argsclass. Chúng tôi cũng đang loại bỏ ParseException, thứ không thực sự thuộc về chúng tôi. Vì vậy, hãy hợp nhất tất cả các ngoại lệ vào một lớp ArgsException duy nhất và chuyển nó vào mô-đun của chính nó.

**public class ArgsException extends Exception { private char errorArgumentId = '\0'; private String errorParameter = "TILT"; private ErrorCode errorCode = ErrorCode.OK;**

**public ArgsException() {}**

**public ArgsException(String message) {super(message);}**

**public enum ErrorCode {**

**OK, MISSING\_STRING, MISSING\_INTEGER, INVALID\_INTEGER, UNEXPECTED\_ARGUMENT, MISSING\_DOUBLE, INVALID\_DOUBLE}**

**}** ---

<https://www.facebook.com/groups/phattrienlaptrinh>

240 **Chapter 14: Successive Reﬁnement**  
240 Chương 14: Tinh luyện liên tiếp

public class Args { ...

private char errorArgumentId = '\0'; private String errorParameter = "TILT";

private **ArgsException**.ErrorCode errorCode = **ArgsException**.ErrorCode.OK; private List<String> argsList;

public Args(String schema, String[] args) throws **ArgsException** { this.schema = schema;

argsList = Arrays.asList(args); valid = parse();

}

private boolean parse() throws ArgsException {

if (schema.length() == 0 && argsList.size() == 0) return true;

parseSchema(); try {

parseArguments();

} catch (**ArgsException** e) { }

return valid; }

private boolean parseSchema() throws **ArgsException** { ...

}

private void parseSchemaElement(String element) throws **ArgsException** { ...

else

throw new **ArgsException**(

String.format("Argument: %c has invalid format: %s.", elementId,elementTail));

}

private void validateSchemaElementId(char elementId) throws **ArgsException** { if (!Character.isLetter(elementId)) {

throw new **ArgsException**(

"Bad character:" + elementId + "in Args format: " + schema); }

}

...

private void parseElement(char argChar) throws **ArgsException** { if (setArgument(argChar))

argsFound.add(argChar); else {

unexpectedArguments.add(argChar);

errorCode = **ArgsException**.ErrorCode.UNEXPECTED\_ARGUMENT; valid = false;

} }

...

<https://www.facebook.com/groups/phattrienlaptrinh>

**String Arguments** 241  
Đối số chuỗi 241

private class StringArgumentMarshaler implements ArgumentMarshaler { private String stringValue = "";

public void set(Iterator<String> currentArgument) throws ArgsException { try {

stringValue = currentArgument.next(); } catch (NoSuchElementException e) {

errorCode = **ArgsException**.ErrorCode.MISSING\_STRING; throw new ArgsException();

} }

public Object get() { return stringValue;

} }

private class IntegerArgumentMarshaler implements ArgumentMarshaler { private int intValue = 0;

public void set(Iterator<String> currentArgument) throws **ArgsException** { String parameter = null;

try {

parameter = currentArgument.next(); intValue = Integer.parseInt(parameter);

} catch (NoSuchElementException e) {

errorCode = ArgsException.ErrorCode.MISSING\_INTEGER; throw new **ArgsException**();

} catch (NumberFormatException e) { errorParameter = parameter;

errorCode = **ArgsException**.ErrorCode.INVALID\_INTEGER; throw new **ArgsException**();

} }

public Object get() { return intValue;

} }

private class DoubleArgumentMarshaler implements ArgumentMarshaler { private double doubleValue = 0;

public void set(Iterator<String> currentArgument) throws ArgsException { String parameter = null;

try {

parameter = currentArgument.next(); doubleValue = Double.parseDouble(parameter);

} catch (NoSuchElementException e) {

errorCode = **ArgsException**.ErrorCode.MISSING\_DOUBLE; throw new ArgsException();

} catch (NumberFormatException e) { errorParameter = parameter;

errorCode = **ArgsException**.ErrorCode.INVALID\_DOUBLE; throw new ArgsException();

} }

<https://www.facebook.com/groups/phattrienlaptrinh>

242 **Chapter 14: Successive Reﬁnement**  
242 Chương 14: Tinh luyện liên tiếp

public Object get() { return doubleValue;

} }

}

This is nice. Now the only exception thrown by Args is ArgsException. Moving ArgsExceptioninto its own module means that we can move a lot of the miscellaneous error support code into that module and out of the Argsmodule. It provides a natural and obvious place to put all that code and will really help us clean up the Argsmodule going forward.  
Cái này đẹp đấy. Bây giờ, ngoại lệ duy nhất do Args ném ra là ArgsException. Di chuyển ArgsException vào mô-đun riêng của nó có nghĩa là chúng ta có thể di chuyển rất nhiều mã hỗ trợ lỗi linh tinh vào mô-đun đó và ra khỏi mô-đun Args. Nó cung cấp một nơi tự nhiên và rõ ràng để đặt tất cả mã đó và sẽ thực sự giúp chúng tôi dọn dẹp Argsmodule trong tương lai.

So now we have completely separated the exception and error code from the Args module. (See Listing 14-13 through Listing 14-16.) This was achieved through a series of about 30 tiny steps, keeping the tests passing between each step.  
Vì vậy, bây giờ chúng tôi đã tách hoàn toàn mã lỗi và ngoại lệ khỏi mô-đun Args. (Xem Liệt kê 14-13 đến Liệt kê 14-16.) Điều này đạt được thông qua một loạt khoảng 30 bước nhỏ, giữ cho các bài kiểm tra được chuyển qua lại giữa mỗi bước.

**Listing 14-13 ArgsTest.java**  
Liệt kê 14-13 ArgsTest.java

package com.objectmentor.utilities.args;

import junit.framework.TestCase;

public class ArgsTest extends TestCase {

public void testCreateWithNoSchemaOrArguments() throws Exception { Args args = new Args("", new String[0]);

assertEquals(0, args.cardinality()); }

public void testWithNoSchemaButWithOneArgument() throws Exception { try {

new Args("", new String[]{"-x"}); fail();

} catch (ArgsException e) { assertEquals(ArgsException.ErrorCode.UNEXPECTED\_ARGUMENT,

e.getErrorCode()); assertEquals('x', e.getErrorArgumentId());

} }

public void testWithNoSchemaButWithMultipleArguments() throws Exception { try {

new Args("", new String[]{"-x", "-y"}); fail();

} catch (ArgsException e) { assertEquals(ArgsException.ErrorCode.UNEXPECTED\_ARGUMENT,

e.getErrorCode()); assertEquals('x', e.getErrorArgumentId());

}

}

public void testNonLetterSchema() throws Exception { try {

new Args("\*", new String[]{});

fail("Args constructor should have thrown exception"); } catch (ArgsException e) {

<https://www.facebook.com/groups/phattrienlaptrinh>

**String Arguments** 243  
Đối số chuỗi 243

**Listing 14-13 (continued) ArgsTest.java  
Liệt kê 14-13 (tiếp theo) ArgsTest.java**

assertEquals(ArgsException.ErrorCode.INVALID\_ARGUMENT\_NAME, e.getErrorCode());

assertEquals('\*', e.getErrorArgumentId()); }

}

public void testInvalidArgumentFormat() throws Exception { try {

new Args("f~", new String[]{});

fail("Args constructor should have throws exception"); } catch (ArgsException e) {

assertEquals(ArgsException.ErrorCode.INVALID\_FORMAT, e.getErrorCode()); assertEquals('f', e.getErrorArgumentId());

} }

public void testSimpleBooleanPresent() throws Exception { Args args = new Args("x", new String[]{"-x"}); assertEquals(1, args.cardinality()); assertEquals(true, args.getBoolean('x'));

}

public void testSimpleStringPresent() throws Exception { Args args = new Args("x\*", new String[]{"-x", "param"}); assertEquals(1, args.cardinality()); assertTrue(args.has('x'));

assertEquals("param", args.getString('x')); }

public void testMissingStringArgument() throws Exception { try {

new Args("x\*", new String[]{"-x"}); fail();

} catch (ArgsException e) { assertEquals(ArgsException.ErrorCode.MISSING\_STRING, e.getErrorCode()); assertEquals('x', e.getErrorArgumentId());

} }

public void testSpacesInFormat() throws Exception { Args args = new Args("x, y", new String[]{"-xy"}); assertEquals(2, args.cardinality()); assertTrue(args.has('x')); assertTrue(args.has('y'));

}

public void testSimpleIntPresent() throws Exception { Args args = new Args("x#", new String[]{"-x", "42"}); assertEquals(1, args.cardinality()); assertTrue(args.has('x'));

assertEquals(42, args.getInt('x')); }

public void testInvalidInteger() throws Exception { try {

new Args("x#", new String[]{"-x", "Forty two"});

<https://www.facebook.com/groups/phattrienlaptrinh>

244 **Chapter 14: Successive Reﬁnement**  
244 Chương 14: Tinh luyện liên tiếp

**Listing 14-13 (continued) ArgsTest.java**  
Liệt kê 14-13 (tiếp theo) ArgsTest.java

fail();

} catch (ArgsException e) { assertEquals(ArgsException.ErrorCode.INVALID\_INTEGER, e.getErrorCode()); assertEquals('x', e.getErrorArgumentId());

assertEquals("Forty two", e.getErrorParameter()); }

}

public void testMissingInteger() throws Exception { try {

new Args("x#", new String[]{"-x"}); fail();

} catch (ArgsException e) { assertEquals(ArgsException.ErrorCode.MISSING\_INTEGER, e.getErrorCode()); assertEquals('x', e.getErrorArgumentId());

} }

public void testSimpleDoublePresent() throws Exception { Args args = new Args("x##", new String[]{"-x", "42.3"}); assertEquals(1, args.cardinality()); assertTrue(args.has('x'));

assertEquals(42.3, args.getDouble('x'), .001); }

public void testInvalidDouble() throws Exception { try {

new Args("x##", new String[]{"-x", "Forty two"}); fail();

} catch (ArgsException e) { assertEquals(ArgsException.ErrorCode.INVALID\_DOUBLE, e.getErrorCode()); assertEquals('x', e.getErrorArgumentId());

assertEquals("Forty two", e.getErrorParameter()); }

}

public void testMissingDouble() throws Exception { try {

new Args("x##", new String[]{"-x"}); fail();

} catch (ArgsException e) { assertEquals(ArgsException.ErrorCode.MISSING\_DOUBLE, e.getErrorCode()); assertEquals('x', e.getErrorArgumentId());

} }

}

**Listing 14-14 ArgsExceptionTest.java  
Liệt kê 14-14 ArgsExceptionTest.java**

public class ArgsExceptionTest extends TestCase { public void testUnexpectedMessage() throws Exception {

ArgsException e =

<https://www.facebook.com/groups/phattrienlaptrinh>

**String Arguments** 245  
Đối số chuỗi 245

**Listing 14-14 (continued) ArgsExceptionTest.java**  
Liệt kê 14-14 (tiếp theo) ArgsExceptionTest.java

new ArgsException(ArgsException.ErrorCode.UNEXPECTED\_ARGUMENT, 'x', null);

assertEquals("Argument -x unexpected.", e.errorMessage()); }

public void testMissingStringMessage() throws Exception {

ArgsException e = new ArgsException(ArgsException.ErrorCode.MISSING\_STRING, 'x', null);

assertEquals("Could not find string parameter for -x.", e.errorMessage()); }

public void testInvalidIntegerMessage() throws Exception { ArgsException e =

new ArgsException(ArgsException.ErrorCode.INVALID\_INTEGER, 'x', "Forty two");

assertEquals("Argument -x expects an integer but was 'Forty two'.", e.errorMessage());

}

public void testMissingIntegerMessage() throws Exception { ArgsException e =

new ArgsException(ArgsException.ErrorCode.MISSING\_INTEGER, 'x', null); assertEquals("Could not find integer parameter for -x.", e.errorMessage());

}

public void testInvalidDoubleMessage() throws Exception {

ArgsException e = new ArgsException(ArgsException.ErrorCode.INVALID\_DOUBLE, 'x', "Forty two");

assertEquals("Argument -x expects a double but was 'Forty two'.", e.errorMessage());

}

public void testMissingDoubleMessage() throws Exception {

ArgsException e = new ArgsException(ArgsException.ErrorCode.MISSING\_DOUBLE, 'x', null);

assertEquals("Could not find double parameter for -x.", e.errorMessage()); }

}

**Listing 14-15 ArgsException.java  
Liệt kê 14-15 ArgsException.java**

public class ArgsException extends Exception { private char errorArgumentId = '\0'; private String errorParameter = "TILT"; private ErrorCode errorCode = ErrorCode.OK;

public ArgsException() {}

public ArgsException(String message) {super(message);}

public ArgsException(ErrorCode errorCode) { this.errorCode = errorCode;

}

<https://www.facebook.com/groups/phattrienlaptrinh>

246 **Chapter 14: Successive Reﬁnement**  
246 Chương 14: Tinh luyện liên tiếp

**Listing 14-15 (continued) ArgsException.java**  
Liệt kê 14-15 (tiếp theo) ArgsException.java

public ArgsException(ErrorCode errorCode, String errorParameter) { this.errorCode = errorCode;

this.errorParameter = errorParameter; }

public ArgsException(ErrorCode errorCode, char errorArgumentId, String errorParameter) {

this.errorCode = errorCode; this.errorParameter = errorParameter; this.errorArgumentId = errorArgumentId;

}

public char getErrorArgumentId() { return errorArgumentId;

}

public void setErrorArgumentId(char errorArgumentId) { this.errorArgumentId = errorArgumentId;

}

public String getErrorParameter() { return errorParameter;

}

public void setErrorParameter(String errorParameter) { this.errorParameter = errorParameter;

}

public ErrorCode getErrorCode() { return errorCode;

}

public void setErrorCode(ErrorCode errorCode) { this.errorCode = errorCode;

}

public String errorMessage() throws Exception { switch (errorCode) {

case OK:

throw new Exception("TILT: Should not get here."); case UNEXPECTED\_ARGUMENT:

return String.format("Argument -%c unexpected.", errorArgumentId); case MISSING\_STRING:

return String.format("Could not find string parameter for -%c.", errorArgumentId);

case INVALID\_INTEGER:

return String.format("Argument -%c expects an integer but was '%s'.", errorArgumentId, errorParameter);

case MISSING\_INTEGER:

return String.format("Could not find integer parameter for -%c.", errorArgumentId);

case INVALID\_DOUBLE:

return String.format("Argument -%c expects a double but was '%s'.", errorArgumentId, errorParameter);

<https://www.facebook.com/groups/phattrienlaptrinh>

**String Arguments** 247  
Đối số chuỗi 247

**Listing 14-15 (continued) ArgsException.java  
Liệt kê 14-15 (tiếp theo) ArgsException.java**

case MISSING\_DOUBLE:

return String.format("Could not find double parameter for -%c.", errorArgumentId);

}

return ""; }

public enum ErrorCode {

OK, INVALID\_FORMAT, UNEXPECTED\_ARGUMENT, INVALID\_ARGUMENT\_NAME, MISSING\_STRING,

MISSING\_INTEGER, INVALID\_INTEGER, MISSING\_DOUBLE, INVALID\_DOUBLE}

}

**Listing 14-16 Args.java**  
Liệt kê 14-16 Args.java

public class Args { private String schema;

private Map<Character, ArgumentMarshaler> marshalers = new HashMap<Character, ArgumentMarshaler>();

private Set<Character> argsFound = new HashSet<Character>(); private Iterator<String> currentArgument;

private List<String> argsList;

public Args(String schema, String[] args) throws ArgsException { this.schema = schema;

argsList = Arrays.asList(args); parse();

}

private void parse() throws ArgsException { parseSchema();

parseArguments(); }

private boolean parseSchema() throws ArgsException {

for (String element : schema.split(",")) { if (element.length() > 0) {

parseSchemaElement(element.trim()); }

}

return true; }

private void parseSchemaElement(String element) throws ArgsException { char elementId = element.charAt(0);

String elementTail = element.substring(1); validateSchemaElementId(elementId);

if (elementTail.length() == 0)

marshalers.put(elementId, new BooleanArgumentMarshaler()); else if (elementTail.equals("\*"))

marshalers.put(elementId, new StringArgumentMarshaler());

<https://www.facebook.com/groups/phattrienlaptrinh>

248 **Chapter 14: Successive Reﬁnement**  
248 Chương 14: Tinh luyện liên tiếp

**Listing 14-16 (continued) Args.java  
Liệt kê 14-16 (tiếp theo) Args.java**

else if (elementTail.equals("#"))

marshalers.put(elementId, new IntegerArgumentMarshaler()); else if (elementTail.equals("##"))

marshalers.put(elementId, new DoubleArgumentMarshaler()); else

throw new ArgsException(ArgsException.ErrorCode.INVALID\_FORMAT, elementId, elementTail);

}

private void validateSchemaElementId(char elementId) throws ArgsException { if (!Character.isLetter(elementId)) {

throw new ArgsException(ArgsException.ErrorCode.INVALID\_ARGUMENT\_NAME, elementId, null);

} }

private void parseArguments() throws ArgsException {

for (currentArgument = argsList.iterator(); currentArgument.hasNext();) { String arg = currentArgument.next();

parseArgument(arg); }

}

private void parseArgument(String arg) throws ArgsException { if (arg.startsWith("-"))

parseElements(arg); }

private void parseElements(String arg) throws ArgsException { for (int i = 1; i < arg.length(); i++)

parseElement(arg.charAt(i)); }

private void parseElement(char argChar) throws ArgsException { if (setArgument(argChar))

argsFound.add(argChar); else {

throw new ArgsException(ArgsException.ErrorCode.UNEXPECTED\_ARGUMENT, argChar, null);

} }

private boolean setArgument(char argChar) throws ArgsException { ArgumentMarshaler m = marshalers.get(argChar);

if (m == null) return false;

try { m.set(currentArgument); return true;

} catch (ArgsException e) { e.setErrorArgumentId(argChar); throw e;

} }

<https://www.facebook.com/groups/phattrienlaptrinh>

**String Arguments** 249  
Đối số chuỗi 249

**Listing 14-16 (continued) Args.java**  
Liệt kê 14-16 (tiếp theo) Args.java

public int cardinality() { return argsFound.size();

}

public String usage() { if (schema.length() > 0)

return "-[" + schema + "]"; else

return ""; }

public boolean getBoolean(char arg) { ArgumentMarshaler am = marshalers.get(arg); boolean b = false;

try {

b = am != null && (Boolean) am.get(); } catch (ClassCastException e) {

b = false; }

return b; }

public String getString(char arg) { ArgumentMarshaler am = marshalers.get(arg); try {

return am == null ? "" : (String) am.get(); } catch (ClassCastException e) {

return ""; }

}

public int getInt(char arg) { ArgumentMarshaler am = marshalers.get(arg); try {

return am == null ? 0 : (Integer) am.get(); } catch (Exception e) {

return 0; }

}

public double getDouble(char arg) { ArgumentMarshaler am = marshalers.get(arg); try {

return am == null ? 0 : (Double) am.get(); } catch (Exception e) {

return 0,0; }

}

public boolean has(char arg) { return argsFound.contains(arg);

} }

<https://www.facebook.com/groups/phattrienlaptrinh>

250 **Chapter 14: Successive Reﬁnement**  
250 Chương 14: Tinh Luyện Liên Tiếp

The majority of the changes to the Argsclass were deletions. A lot of code just got moved out of Args and put into ArgsException. Nice. We also moved all the ArgumentMarshallers into their own ﬁles. Nicer!  
Phần lớn các thay đổi đối với Argsclass là xóa. Rất nhiều mã vừa được chuyển ra khỏi Args và đưa vào ArgsException. Đẹp. Chúng tôi cũng di chuyển tất cả các Argument Marshallers vào các tệp riêng của chúng. Tốt hơn!

Much of good software design is simply about partitioning—creating appropriate places to put different kinds of code. This separation of concerns makes the code much simpler to understand and maintain.  
Phần lớn thiết kế phần mềm tốt chỉ đơn giản là về phân vùng—tạo ra những vị trí thích hợp để đặt các loại mã khác nhau. Sự tách biệt các mối quan tâm này làm cho mã dễ hiểu và dễ bảo trì hơn nhiều.

Of special interest is the errorMessagemethod of ArgsException. Clearly it was a vio-lation of the SRP to put the error message formatting into Args. Argsshould be about the processing of arguments, not about the format of the error messages. However, does it really make sense to put the error message formatting code into ArgsException?  
Quan tâm đặc biệt là errorMessagemethod của ArgsException. Rõ ràng là đã vi phạm SRP khi đặt định dạng thông báo lỗi vào Args. Đối số nên là về việc xử lý đối số, không phải về định dạng của thông báo lỗi. Tuy nhiên, việc đưa mã định dạng thông báo lỗi vào ArgsException có thực sự hợp lý không?

Frankly, it’s a compromise. Users who don’t like the error messages supplied by ArgsExceptionwill have to write their own. But the convenience of having canned error messages already prepared for you is not insigniﬁcant.  
Thành thật mà nói, đó là một sự thỏa hiệp. Người dùng không thích thông báo lỗi do ArgsException cung cấp sẽ phải tự viết. Nhưng sự tiện lợi của việc chuẩn bị sẵn các thông báo lỗi soạn sẵn cho bạn không phải là không đáng kể.

By now it should be clear that we are within striking distance of the ﬁnal solution that appeared at the start of this chapter. I’ll leave the ﬁnal transformations to you as an exercise.  
Đến đây, rõ ràng là chúng ta đang ở rất gần với giải pháp cuối cùng xuất hiện ở đầu chương này. Tôi sẽ để lại những phép biến đổi cuối cùng cho bạn như một bài tập.

[**Conclusion**](#_page_459_0)  
Phần kết luận

It is not enough for code to work. Code that works is often badly broken. Programmers who satisfy themselves with merely working code are behaving unprofessionally. They may fear that they don’t have time to improve the structure and design of their code, but I disagree. Nothing has a more profound and long-term degrading effect upon a develop-ment project than bad code. Bad schedules can be redone, bad requirements can be rede-ﬁned. Bad team dynamics can be repaired. But bad code rots and ferments, becoming an inexorable weight that drags the team down. Time and time again I have seen teams grind to a crawl because, in their haste, they created a malignant morass of code that forever thereafter dominated their destiny.  
Nó không đủ để mã hoạt động. Mã hoạt động thường bị hỏng nặng. Các lập trình viên tự thỏa mãn với mã làm việc đơn thuần đang cư xử không chuyên nghiệp. Họ có thể sợ rằng họ không có thời gian để cải thiện cấu trúc và thiết kế mã của họ, nhưng tôi không đồng ý. Không có gì có tác động làm suy giảm sâu sắc và lâu dài đối với một dự án phát triển hơn là mã xấu. Lịch trình không phù hợp có thể được làm lại, các yêu cầu không phù hợp có thể được xác định lại. Động lực nhóm xấu có thể được sửa chữa. Nhưng mã xấu sẽ thối rữa và lên men, trở thành một sức nặng không thể lay chuyển kéo cả nhóm đi xuống. Hết lần này đến lần khác, tôi đã chứng kiến các đội phải vật lộn vì trong lúc vội vàng, họ đã tạo ra một mớ mã độc ác mà sau đó sẽ mãi mãi chi phối vận mệnh của họ.

Of course bad code can be cleaned up. But it’s very expensive. As code rots, the mod-ules insinuate themselves into each other, creating lots of hidden and tangled dependen-cies. Finding and breaking old dependencies is a long and arduous task. On the other hand, keeping code clean is relatively easy. If you made a mess in a module in the morning, it is easy to clean it up in the afternoon. Better yet, if you made a mess ﬁve minutes ago, it’s very easy to clean it up right now.  
Tất nhiên mã xấu có thể được dọn sạch. Nhưng nó rất đắt. Khi mã bị thối rữa, các mô-đun tự ám chỉ lẫn nhau, tạo ra rất nhiều phụ thuộc ẩn và rối rắm. Tìm và phá vỡ các quan hệ phụ thuộc cũ là một nhiệm vụ lâu dài và gian khổ. Mặt khác, giữ mã sạch sẽ tương đối dễ dàng. Nếu bạn làm lộn xộn một mô-đun vào buổi sáng, bạn sẽ dễ dàng dọn dẹp nó vào buổi chiều. Tốt hơn nữa, nếu bạn đã làm bừa bộn năm phút trước, thì bạn sẽ rất dễ dàng dọn dẹp nó ngay bây giờ.

So the solution is to continuously keep your code as clean and simple as it can be. Never let the rot get started.  
Vì vậy, giải pháp là liên tục giữ cho mã của bạn sạch sẽ và đơn giản nhất có thể. Không bao giờ để thối bắt đầu.

<https://www.facebook.com/groups/phattrienlaptrinh>

[**15**](#_page_533_0)

[**JUnit Internals**](#_page_533_0)  
Nội bộ JUnit

JUnit is one of the most famous of all Java frameworks. As frameworks go, it is simple in conception, precise in deﬁnition, and elegant in implementation. But what does the code look like? In this chapter we’ll critique an example drawn from the JUnit framework.  
JUnit là một trong những framework Java nổi tiếng nhất. Khi các framework hoạt động, nó đơn giản trong khái niệm, chính xác trong định nghĩa và tao nhã trong triển khai. Nhưng mã trông như thế nào? Trong chương này, chúng ta sẽ phê bình một ví dụ được rút ra từ khung JUnit.

251

<https://www.facebook.com/groups/phattrienlaptrinh>

252 **Chapter 15: JUnit Internals**  
252 Chương 15: Nội bộ JUnit

[**The JUnit Framework**](#_page_533_0)  
Khung JUnit

JUnit has had many authors, but it began with Kent Beck and Eric Gamma together on a plane to Atlanta. Kent wanted to learn Java, and Eric wanted to learn about Kent’s Small-talk testing framework. “What could be more natural to a couple of geeks in cramped quarters than to pull out our laptops and start coding?”1 After three hours of high-altitude work, they had written the basics of JUnit.  
JUnit đã có nhiều tác giả, nhưng nó bắt đầu với Kent Beck và Eric Gamma cùng nhau trên một chuyến bay đến Atlanta. Kent muốn học Java và Eric muốn tìm hiểu về khuôn khổ thử nghiệm Small-talk của Kent. “Điều gì có thể tự nhiên hơn đối với một vài chuyên viên máy tính trong những căn phòng chật hẹp hơn là lấy máy tính xách tay của chúng tôi ra và bắt đầu viết mã?”1 Sau ba giờ làm việc căng thẳng, họ đã viết được những kiến thức cơ bản về JUnit.

The module we’ll look at is the clever bit of code that helps identify string compari-son errors. This module is called ComparisonCompactor. Given two strings that differ, such as ABCDEand ABXDE,it will expose the difference by generating a string such as <...B[X]D...>.  
Mô-đun mà chúng ta sẽ xem xét là một đoạn mã thông minh giúp xác định các lỗi so sánh chuỗi. Mô-đun này được gọi là Máy so sánh. Cho hai chuỗi khác nhau, chẳng hạn như ABCDE và ABXDE, nó sẽ cho thấy sự khác biệt bằng cách tạo ra một chuỗi chẳng hạn như <...B[X]D...>.

I could explain it further, but the test cases do a better job. So take a look at Listing 15-1 and you will understand the requirements of this module in depth. While you are at it, critique the structure of the tests. Could they be simpler or more obvious?  
Tôi có thể giải thích thêm, nhưng các trường hợp thử nghiệm hoạt động tốt hơn. Vì vậy, hãy xem Liệt kê 15-1 và bạn sẽ hiểu sâu các yêu cầu của mô-đun này. Trong khi bạn đang ở đó, hãy phê bình cấu trúc của các bài kiểm tra. Chúng có thể đơn giản hơn hoặc rõ ràng hơn không?

**Listing 15-1 ComparisonCompactorTest.java** package junit.tests.framework;  
Liệt kê 15-1 So sánhCompactorTest.java gói junit.tests.framework;

import junit.framework.ComparisonCompactor; import junit.framework.TestCase;

public class ComparisonCompactorTest extends TestCase {

public void testMessage() {

String failure= new ComparisonCompactor(0, "b", "c").compact("a"); assertTrue("a expected:<[b]> but was:<[c]>".equals(failure));

}

public void testStartSame() {

String failure= new ComparisonCompactor(1, "ba", "bc").compact(null); assertEquals("expected:<b[a]> but was:<b[c]>", failure);

}

public void testEndSame() {

String failure= new ComparisonCompactor(1, "ab", "cb").compact(null); assertEquals("expected:<[a]b> but was:<[c]b>", failure);

}

public void testSame() {

String failure= new ComparisonCompactor(1, "ab", "ab").compact(null); assertEquals("expected:<ab> but was:<ab>", failure);

}

public void testNoContextStartAndEndSame() {

String failure= new ComparisonCompactor(0, "abc", "adc").compact(null); assertEquals("expected:<...[b]...> but was:<...[d]...>", failure);

}

1. *JUnit Pocket Guide*, Kent Beck, O’Reilly, 2004, p. 43.

<https://www.facebook.com/groups/phattrienlaptrinh>

**The JUnit Framework** 253  
Khung JUnit 253

**Listing 15-1 (continued) ComparisonCompactorTest.java**  
Liệt kê 15-1 (tiếp theo) So sánhCompactorTest.java

public void testStartAndEndContext() {

String failure= new ComparisonCompactor(1, "abc", "adc").compact(null); assertEquals("expected:<a[b]c> but was:<a[d]c>", failure);

}

public void testStartAndEndContextWithEllipses() { String failure=

new ComparisonCompactor(1, "abcde", "abfde").compact(null); assertEquals("expected:<...b[c]d...> but was:<...b[f]d...>", failure);

}

public void testComparisonErrorStartSameComplete() {

String failure= new ComparisonCompactor(2, "ab", "abc").compact(null); assertEquals("expected:<ab[]> but was:<ab[c]>", failure);

}

public void testComparisonErrorEndSameComplete() {

String failure= new ComparisonCompactor(0, "bc", "abc").compact(null); assertEquals("expected:<[]...> but was:<[a]...>", failure);

}

public void testComparisonErrorEndSameCompleteContext() {

String failure= new ComparisonCompactor(2, "bc", "abc").compact(null); assertEquals("expected:<[]bc> but was:<[a]bc>", failure);

}

public void testComparisonErrorOverlapingMatches() {

String failure= new ComparisonCompactor(0, "abc", "abbc").compact(null); assertEquals("expected:<...[]...> but was:<...[b]...>", failure);

}

public void testComparisonErrorOverlapingMatchesContext() {

String failure= new ComparisonCompactor(2, "abc", "abbc").compact(null); assertEquals("expected:<ab[]c> but was:<ab[b]c>", failure);

}

public void testComparisonErrorOverlapingMatches2() { String failure= new ComparisonCompactor(0, "abcdde",

"abcde").compact(null);

assertEquals("expected:<...[d]...> but was:<...[]...>", failure); }

public void testComparisonErrorOverlapingMatches2Context() { String failure=

new ComparisonCompactor(2, "abcdde", "abcde").compact(null); assertEquals("expected:<...cd[d]e> but was:<...cd[]e>", failure);

}

public void testComparisonErrorWithActualNull() {

String failure= new ComparisonCompactor(0, "a", null).compact(null); assertEquals("expected:<a> but was:<null>", failure);

}

public void testComparisonErrorWithActualNullContext() {

String failure= new ComparisonCompactor(2, "a", null).compact(null);

<https://www.facebook.com/groups/phattrienlaptrinh>

254 **Chapter 15: JUnit Internals**  
254 Chương 15: Nội bộ JUnit

**Listing 15-1 (continued) ComparisonCompactorTest.java  
Liệt kê 15-1 (tiếp theo) So sánhCompactorTest.java**

assertEquals("expected:<a> but was:<null>", failure); }

public void testComparisonErrorWithExpectedNull() {

String failure= new ComparisonCompactor(0, null, "a").compact(null); assertEquals("expected:<null> but was:<a>", failure);

}

public void testComparisonErrorWithExpectedNullContext() {

String failure= new ComparisonCompactor(2, null, "a").compact(null); assertEquals("expected:<null> but was:<a>", failure);

}

public void testBug609972() {

String failure= new ComparisonCompactor(10, "S&P500", "0").compact(null); assertEquals("expected:<[S&P50]0> but was:<[]0>", failure);

} }

I ran a code coverage analysis on the ComparisonCompactorusing these tests. The code is 100 percent covered. Every line of code, every ifstatement and forloop, is executed by the tests. This gives me a high degree of conﬁdence that the code works and a high degree of respect for the craftsmanship of the authors.  
Tôi đã chạy phân tích phạm vi mã trên Máy so sánh bằng cách sử dụng các thử nghiệm này. Mã được bảo hiểm 100 phần trăm. Mọi dòng mã, mọi câu lệnh if và vòng lặp for đều được thực hiện bởi các bài kiểm tra. Điều này mang lại cho tôi sự tự tin cao độ rằng mã hoạt động và mức độ tôn trọng cao đối với tay nghề của các tác giả.

The code for ComparisonCompactoris in Listing 15-2. Take a moment to look over this code. I think you’ll ﬁnd it to be nicely partitioned, reasonably expressive, and simple in structure. Once you are done, then we’ll pick the nits together.  
Mã cho phép so sánhCompactoris trong Liệt kê 15-2. Hãy dành một chút thời gian để xem qua mã này. Tôi nghĩ bạn sẽ thấy nó được phân vùng độc đáo, diễn đạt hợp lý và cấu trúc đơn giản. Khi bạn đã hoàn tất, chúng ta sẽ cùng nhau nhặt trứng chấy.

**Listing 15-2**

**ComparisonCompactor.java (Original)** package junit.framework;  
Gói so sánhCompactor.java (Bản gốc) junit.framework;

public class ComparisonCompactor {

private static final String ELLIPSIS = "..."; private static final String DELTA\_END = "]"; private static final String DELTA\_START = "[";

private int fContextLength; private String fExpected; private String fActual; private int fPrefix; private int fSuffix;

public ComparisonCompactor(int contextLength, String expected,

String actual) { fContextLength = contextLength;

fExpected = expected; fActual = actual;

}

<https://www.facebook.com/groups/phattrienlaptrinh>

**The JUnit Framework** 255  
Khung JUnit 255

**Listing 15-2 (continued) ComparisonCompactor.java (Original)  
Liệt kê 15-2 (tiếp theo) So sánhCompactor.java (Bản gốc)**

public String compact(String message) {

if (fExpected == null || fActual == null || areStringsEqual()) return Assert.format(message, fExpected, fActual);

findCommonPrefix(); findCommonSuffix();

String expected = compactString(fExpected); String actual = compactString(fActual);

return Assert.format(message, expected, actual); }

private String compactString(String source) { String result = DELTA\_START +

source.substring(fPrefix, source.length() -fSuffix + 1) + DELTA\_END;

if (fPrefix > 0)

result = computeCommonPrefix() + result; if (fSuffix > 0)

result = result + computeCommonSuffix(); return result;

}

private void findCommonPrefix() { fPrefix = 0;

int end = Math.min(fExpected.length(), fActual.length()); for (; fPrefix < end; fPrefix++) {

if (fExpected.charAt(fPrefix) != fActual.charAt(fPrefix)) break;

} }

private void findCommonSuffix() {

int expectedSuffix = fExpected.length() - 1; int actualSuffix = fActual.length() - 1; for (;

actualSuffix >= fPrefix && expectedSuffix >= fPrefix; actualSuffix--, expectedSuffix--) {

if (fExpected.charAt(expectedSuffix) != fActual.charAt(actualSuffix)) break;

}

fSuffix = fExpected.length() - expectedSuffix; }

private String computeCommonPrefix() {

return (fPrefix > fContextLength ? ELLIPSIS : "") + fExpected.substring(Math.max(0, fPrefix - fContextLength),

fPrefix); }

private String computeCommonSuffix() {

int end = Math.min(fExpected.length() - fSuffix + 1 + fContextLength, fExpected.length());

return fExpected.substring(fExpected.length() - fSuffix + 1, end) + (fExpected.length() - fSuffix + 1 < fExpected.length() -

fContextLength ? ELLIPSIS : ""); }

<https://www.facebook.com/groups/phattrienlaptrinh>

256 **Chapter 15: JUnit Internals**  
256 Chương 15: Nội bộ JUnit

**Listing 15-2 (continued) ComparisonCompactor.java (Original)**  
Liệt kê 15-2 (tiếp theo) So sánhCompactor.java (Bản gốc)

private boolean areStringsEqual() { return fExpected.equals(fActual);

} }

You might have a few complaints about this module. There are some long expressions and some strange +1s and so forth. But overall this module is pretty good. After all, it might have looked like Listing 15-3.  
Bạn có thể có một vài phàn nàn về mô-đun này. Có một số biểu thức dài và một số +1 lạ, v.v. Nhưng nhìn chung mô-đun này là khá tốt. Xét cho cùng, nó có thể trông giống như Liệt kê 15-3.

**Listing 15-3  
Liệt kê 15-3**

**ComparisonCompator.java (defactored)** package junit.framework;  
Gói so sánhCompator.java (đã giải cấu trúc) junit.framework;

public class ComparisonCompactor { private int ctxt;

private String s1; private String s2; private int pfx; private int sfx;

public ComparisonCompactor(int ctxt, String s1, String s2) { this.ctxt = ctxt;

this.s1 = s1; this.s2 = s2;

}

public String compact(String msg) {

if (s1 == null || s2 == null || s1.equals(s2)) return Assert.format(msg, s1, s2);

pfx = 0;

for (; pfx < Math.min(s1.length(), s2.length()); pfx++) { if (s1.charAt(pfx) != s2.charAt(pfx))

break; }

int sfx1 = s1.length() - 1; int sfx2 = s2.length() - 1;

for (; sfx2 >= pfx && sfx1 >= pfx; sfx2--, sfx1--) { if (s1.charAt(sfx1) != s2.charAt(sfx2))

break; }

sfx = s1.length() - sfx1; String cmp1 = compactString(s1); String cmp2 = compactString(s2);

return Assert.format(msg, cmp1, cmp2); }

private String compactString(String s) { String result =

"[" + s.substring(pfx, s.length() - sfx + 1) + "]"; if (pfx > 0)

result = (pfx > ctxt ? "..." : "") + s1.substring(Math.max(0, pfx - ctxt), pfx) + result;

<https://www.facebook.com/groups/phattrienlaptrinh>

**The JUnit Framework** 257  
Khung JUnit 257

**Listing 15-3 (continued) ComparisonCompator.java (defactored)  
Liệt kê 15-3 (tiếp theo) So sánhCompator.java (đã giải cấu trúc)**

if (sfx > 0) {

int end = Math.min(s1.length() - sfx + 1 + ctxt, s1.length()); result = result + (s1.substring(s1.length() - sfx + 1, end) +

(s1.length() - sfx + 1 < s1.length() - ctxt ? "..." : "")); }

return result; }

}

Even though the authors left this module in very good shape, the *Boy Scout Rule*2 tells us we should leave it cleaner than we found it. So, how can we improve on the original code in Listing 15-2?  
Mặc dù các tác giả đã để lại mô-đun này ở tình trạng rất tốt, nhưng Quy tắc Hướng đạo sinh2 cho chúng ta biết rằng chúng ta nên để nó sạch sẽ hơn những gì chúng ta thấy. Vì vậy, làm thế nào chúng ta có thể cải thiện mã ban đầu trong Liệt kê 15-2?

The ﬁrst thing I don’t care for is the fpreﬁx for the member variables [N6]. Today’s environments make this kind of scope encoding redundant. So let’s eliminate all the f’s.  
Điều đầu tiên tôi không quan tâm là fprefix cho các biến thành viên [N6]. Môi trường ngày nay làm cho loại mã hóa phạm vi này trở nên dư thừa. Vì vậy, hãy loại bỏ tất cả các lỗi.

private int contextLength; private String expected; private String actual; private int prefix; private int suffix;

Next, we have an unencapsulated conditional at the beginning of the compactfunction [G28].  
Tiếp theo, chúng ta có một điều kiện không đóng gói ở đầu hàm thu gọn [G28].

public String compact(String message) {

**if (expected == null || actual == null || areStringsEqual())** return Assert.format(message, expected, actual);

findCommonPrefix(); findCommonSuffix();

String expected = compactString(this.expected); String actual = compactString(this.actual); return Assert.format(message, expected, actual);

}

This conditional should be encapsulated to make our intent clear. So let’s extract a method that explains it.  
Điều kiện này nên được gói gọn để làm rõ ý định của chúng ta. Vì vậy, hãy trích xuất một phương thức giải thích nó.

public String compact(String message) { if (**shouldNotCompact()**)

return Assert.format(message, expected, actual);

findCommonPrefix(); findCommonSuffix();

String expected = compactString(this.expected); String actual = compactString(this.actual); return Assert.format(message, expected, actual);

}

2. See “The Boy Scout Rule” on page 14.  
2. Xem “Quy tắc Hướng đạo sinh” ở trang 14.

<https://www.facebook.com/groups/phattrienlaptrinh>

258 **Chapter 15: JUnit Internals**  
258 Chương 15: Nội bộ JUnit

**private boolean shouldNotCompact() {**

**return expected == null || actual == null || areStringsEqual(); }**

I don’t much care for the this.expectedand this.actualnotation in the compactfunc-tion. This happened when we changed the name of fExpectedto expected. Why are there variables in this function that have the same names as the member variables? Don’t they represent something else [N4]? We should make the names unambiguous.  
Tôi không quan tâm lắm đến ký hiệu this.expected và this.actual trong hàm thu gọn. Điều này xảy ra khi chúng tôi thay đổi tên của fExpected thành mong đợi. Tại sao có các biến trong hàm này có cùng tên với các biến thành viên? Họ không đại diện cho cái gì khác [N4]? Chúng ta nên làm cho những cái tên rõ ràng.

String **compactExpected** = compactString(**expected**); String **compactActual** = compactString(**actual**);

Negatives are slightly harder to understand than positives [G29]. So let’s turn that if statement on its head and invert the sense of the conditional.  
Điều tiêu cực khó hiểu hơn một chút so với điều tích cực [G29]. Vì vậy, hãy đảo ngược câu lệnh if đó và đảo ngược ý nghĩa của điều kiện.

public String compact(String message) { if (**canBeCompacted**()) {

findCommonPrefix(); findCommonSuffix();

String compactExpected = compactString(expected); String compactActual = compactString(actual);

return Assert.format(message, compactExpected, compactActual); } else {

return Assert.format(message, expected, actual); }

}

private boolean **canBeCompacted**() {

return expected **!=** null **&&** actual **!=** null **&& !**areStringsEqual(); }

The name of the function is strange [N7]. Although it does compact the strings, it actually might not compact the strings if canBeCompactedreturns false. So naming this function compacthides the side effect of the error check. Notice also that the function returns a formatted message, not just the compacted strings. So the name of the function should really be formatCompactedComparison. That makes it read a lot better when taken with the function argument:  
Tên hàm lạ quá [N7]. Mặc dù nó nén các chuỗi, nhưng nó thực sự có thể không nén các chuỗi nếu canBeCompacted trả về false. Vì vậy, việc đặt tên cho chức năng này sẽ thu gọn tác dụng phụ của việc kiểm tra lỗi. Cũng lưu ý rằng hàm trả về một thông báo được định dạng, không chỉ các chuỗi được nén. Vì vậy, tên của hàm thực sự phải là formatCompactedComparison. Điều đó làm cho nó đọc tốt hơn rất nhiều khi được thực hiện với đối số chức năng:

public String formatCompactedComparison(String message) {

The body of the ifstatement is where the true compacting of the expected and actual strings is done. We should extract that as a method named compactExpectedAndActual. How-ever, we want the formatCompactedComparison function to do all the formatting. The compact... function should do nothing but compacting [G30]. So let’s split it up as follows:  
Phần thân của câu lệnh if là nơi thực hiện quá trình nén thực sự của các chuỗi dự kiến và thực tế. Chúng ta nên trích xuất nó dưới dạng một phương thức có tên là compactExpectedAndActual. Tuy nhiên, chúng tôi muốn hàm formatCompactedComparison thực hiện tất cả các định dạng. Hàm compact... không được làm gì khác ngoài việc nén [G30]. Vì vậy, hãy chia nó ra như sau:

...

**private String compactExpected; private String compactActual;**

...

public String formatCompactedComparison(String message) { if (canBeCompacted()) {

**compactExpectedAndActual();**

return Assert.format(message, compactExpected, compactActual); } else {

<https://www.facebook.com/groups/phattrienlaptrinh>

**The JUnit Framework** 259  
Khung JUnit 259

return Assert.format(message, expected, actual); }

}

private void **compactExpectedAndActual**() { findCommonPrefix(); findCommonSuffix();

compactExpected = compactString(expected); compactActual = compactString(actual);

}

Notice that this required us to promote compactExpectedand compactActualto member variables. I don’t like the way that the last two lines of the new function return variables, but the ﬁrst two don’t. They aren’t using consistent conventions [G11]. So we should change findCommonPrefix and findCommonSuffix to return the preﬁx and sufﬁx values.  
Lưu ý rằng điều này yêu cầu chúng tôi quảng cáo các biến thành viên compactExpected và compactActual. Tôi không thích cách mà hai dòng cuối cùng của hàm mới trả về các biến, nhưng hai dòng đầu tiên thì không. Họ không sử dụng các quy ước nhất quán [G11]. Vì vậy, chúng ta nên thay đổi findCommonPrefix và findCommonSuffix để trả về các giá trị tiền tố và hậu tố.

private void compactExpectedAndActual() { **prefixIndex =** findCommonPrefix(); **suffixIndex =** findCommonSuffix(); compactExpected = compactString(expected); compactActual = compactString(actual);

}

private **int** findCommonPrefix() { **int** prefix**Index** = 0;

int end = Math.min(expected.length(), actual.length()); for (; prefix**Index** < end; prefix**Index**++) {

if (expected.charAt(prefix**Index**) != actual.charAt(prefix**Index**)) break;

}

**return prefixIndex;** }

private **int** findCommonSuffix() {

int expectedSuffix = expected.length() - 1; int actualSuffix = actual.length() - 1;

for (; actualSuffix >= prefix**Index** && expectedSuffix >= prefixIndex; actualSuffix--, expectedSuffix--) {

if (expected.charAt(expectedSuffix) != actual.charAt(actualSuffix)) break;

}

**return** expected.length() - expectedSuffix; }

We should also change the names of the member variables to be a little more accurate [N1]; after all, they are both indices.  
Chúng ta cũng nên thay đổi tên của các biến thành viên để chính xác hơn một chút [N1]; Rốt cuộc, cả hai đều là chỉ số.

Careful inspection of findCommonSuffixexposes a *hidden temporal coupling* [G31]; it depends on the fact that prefixIndexis calculated by findCommonPrefix. If these two func-tions were called out of order, there would be a difﬁcult debugging session ahead. So, to expose this temporal coupling, let’s have findCommonSuffixtake the prefixIndex as an argument.  
Kiểm tra cẩn thận findCommonSuffix cho thấy khớp nối thời gian ẩn [G31]; nó phụ thuộc vào thực tế là prefixIndex được tính bởi findCommonPrefix. Nếu hai chức năng này được gọi không đúng thứ tự, sẽ có một phiên sửa lỗi khó khăn phía trước. Vì vậy, để hiển thị khớp nối thời gian này, hãy để findCommonSuffix lấy tiền tốIndex làm đối số.

private void compactExpectedAndActual() { prefixIndex = findCommonPrefix(); suffixIndex = findCommonSuffix(**prefixIndex**);

<https://www.facebook.com/groups/phattrienlaptrinh>

260 **Chapter 15: JUnit Internals**  
260 Chương 15: Nội bộ JUnit

compactExpected = compactString(expected); compactActual = compactString(actual);

}

private int findCommonSuffix(**int prefixIndex**) { int expectedSuffix = expected.length() - 1; int actualSuffix = actual.length() - 1;

for (; actualSuffix >= prefixIndex && expectedSuffix >= prefixIndex; actualSuffix--, expectedSuffix--) {

if (expected.charAt(expectedSuffix) != actual.charAt(actualSuffix)) break;

}

return expected.length() - expectedSuffix; }

I’m not really happy with this. The passing of the prefixIndexargument is a bit arbitrary [G32]. It works to establish the ordering but does nothing to explain the need for that ordering. Another programmer might undo what we have done because there’s no indica-tion that the parameter is really needed. So let’s take a different tack.  
Tôi không thực sự hài lòng với điều này. Việc chuyển tiền tố Indexargument hơi tùy ý [G32]. Nó hoạt động để thiết lập thứ tự nhưng không làm gì để giải thích sự cần thiết của thứ tự đó. Một lập trình viên khác có thể hoàn tác những gì chúng tôi đã làm vì không có dấu hiệu nào cho thấy tham số đó thực sự cần thiết. Vì vậy, hãy thực hiện một chiến thuật khác.

private void compactExpectedAndActual() { **findCommonPrefixAndSuffix**(); compactExpected = compactString(expected); compactActual = compactString(actual);

}

private **void findCommonPrefixAndSuffix**() { **findCommonPrefix();**

int expectedSuffix = expected.length() - 1; int actualSuffix = actual.length() - 1; for (;

actualSuffix >= prefixIndex && expectedSuffix >= prefixIndex; actualSuffix--, expectedSuffix--

) {

if (expected.charAt(expectedSuffix) != actual.charAt(actualSuffix)) break;

}

suffixIndex = expected.length() - expectedSuffix; }

private **void** findCommonPrefix() { prefixIndex = 0;

int end = Math.min(expected.length(), actual.length()); for (; prefixIndex < end; prefixIndex++)

if (expected.charAt(prefixIndex) != actual.charAt(prefixIndex)) break;

}

We put findCommonPrefixand findCommonSuffixback the way they were, changing the name of findCommonSuffixto findCommon**PrefixAnd**Suffixand having it call findCommon-Prefixbefore doing anything else. That establishes the temporal nature of the two func-tions in a much more dramatic way than the previous solution. It also points out how ugly findCommonPrefixAndSuffix is. Let’s clean it up now.  
Chúng tôi đặt findCommonPrefixand findCommonSuffix trở lại như cũ, thay đổi tên của findCommonSuffix thành findCommonPrefixAndSuffix và để nó gọi findCommon-Prefix trước khi thực hiện bất kỳ điều gì khác. Điều đó thiết lập bản chất tạm thời của hai chức năng theo cách ấn tượng hơn nhiều so với giải pháp trước đó. Nó cũng chỉ ra findCommonPrefixAndSuffix xấu như thế nào. Hãy làm sạch nó ngay bây giờ.

private void findCommonPrefixAndSuffix() { findCommonPrefix();

int suffixLength = 1;

<https://www.facebook.com/groups/phattrienlaptrinh>

**The JUnit Framework** 261  
Khung JUnit 261

for (; !suffixOverlapsPrefix(suffixLength); suffixLength++) { if (charFromEnd(expected, suffixLength) !=

charFromEnd(actual, suffixLength)) break;

}

suffixIndex = suffixLength; }

private char charFromEnd(String s, int i) { return s.charAt(s.length()-i);}

private boolean suffixOverlapsPrefix(int suffixLength) { return actual.length() - suffixLength < prefixLength ||

expected.length() - suffixLength < prefixLength; }

This is much better. It exposes that the suffixIndexis really the length of the sufﬁx and is not well named. The same is true of the prefixIndex, though in that case “index” and “length” are synonymous. Even so, it is more consistent to use “length.” The problem is that the suffixIndexvariable is not zero based; it is 1 based and so is not a true length. This is also the reason that there are all those +1s in computeCommonSuffix[G33]. So let’s ﬁx that. The result is in Listing 15-4.  
Thế này tốt hơn. Nó cho thấy rằng hậu tố Index thực sự là độ dài của hậu tố và không được đặt tên hay. Điều này cũng đúng với tiền tố Index, mặc dù trong trường hợp đó, “chỉ mục” và “độ dài” đồng nghĩa với nhau. Mặc dù vậy, việc sử dụng “độ dài” sẽ nhất quán hơn. Vấn đề là hậu tốIndexvariable không dựa trên số 0; nó dựa trên 1 và do đó không phải là độ dài thực. Đây cũng là lý do có tất cả các +1 đó trong computeCommonSuffix[G33]. Vì vậy, hãy khắc phục điều đó. Kết quả là trong Liệt kê 15-4.

**Listing 15-4**

**ComparisonCompactor.java (interim)**

public class ComparisonCompactor { ...

private int **suffixLength**; ...

private void findCommonPrefixAndSuffix() { findCommonPrefix();

**suffixLength = 0;**

for (; !suffixOverlapsPrefix(suffixLength); suffixLength++) { if (charFromEnd(expected, suffixLength) !=

charFromEnd(actual, suffixLength)) break;

} }

private char charFromEnd(String s, int i) { return s.charAt(s.length() - i **- 1**);

}

private boolean suffixOverlapsPrefix(int suffixLength) { return actual.length() - suffixLength **<=** prefixLength ||

expected.length() - suffixLength **<=** prefixLength; }

...

private String compactString(String source) { String result =

DELTA\_START +

source.substring(prefixLength, source.length() - **suffixLength**) + DELTA\_END;

if (prefixLength > 0)

result = computeCommonPrefix() + result;

<https://www.facebook.com/groups/phattrienlaptrinh>

262 **Chapter 15: JUnit Internals**  
262 Chương 15: Nội bộ JUnit

**Listing 15-4 (continued) ComparisonCompactor.java (interim)  
Liệt kê 15-4 (tiếp theo) So sánhCompactor.java (tạm thời)**

if (**suffixLength** > 0)

result = result + computeCommonSuffix(); return result;

}

...

private String computeCommonSuffix() {

int end = Math.min(expected.length() - **suffixLength** + contextLength, expected.length()

); return

expected.substring(expected.length() - **suffixLength**, end) + (expected.length() - **suffixLength** <

expected.length() - contextLength ? ELLIPSIS : "");

}

We replaced the +1s in computeCommonSuffixwith a -1in charFromEnd, where it makes perfect sense, and two <=operators in suffixOverlapsPrefix, where they also make perfect sense. This allowed us to change the name of suffixIndexto suffixLength, greatly enhanc-ing the readability of the code.  
Chúng tôi đã thay thế các +1 trong computeCommonSuffix bằng -1in charFromEnd, nơi nó có ý nghĩa hoàn hảo và hai toán tử <= trong hậu tốOverlapsPrefix, nơi chúng cũng có ý nghĩa hoàn hảo. Điều này cho phép chúng tôi thay đổi tên của suffixIndex thành suffixLength, giúp cải thiện đáng kể khả năng đọc mã.

There is a problem however.As I was eliminating the +1s, I noticed the following line in compactString:  
Tuy nhiên, có một vấn đề. Khi tôi đang loại bỏ +1, tôi nhận thấy dòng sau trong compactString:

if (suffixLength > 0)

Take a look at it in Listing 15-4. By rights, because suffixLengthis now one less than it used to be, I should change the >operator to a >= operator. But that makes no sense. It makes sense *now!* This means that it didn’t use to make sense and was probably a bug. Well, not quite a bug. Upon further analysis we see that the ifstatement now prevents a zero length sufﬁx from being appended. Before we made the change, the if statement was nonfunctional because suffixIndex could never be less than one!  
Hãy xem nó trong Liệt kê 15-4. Theo quyền, bởi vì hậu tốLengthis bây giờ ít hơn trước đây một hậu tố, tôi nên thay đổi toán tử > thành toán tử >=. Nhưng điều đó chẳng có ý nghĩa gì. Nó có ý nghĩa bây giờ! Điều này có nghĩa là nó không được sử dụng để hiểu và có thể là một lỗi. Vâng, không hoàn toàn là một lỗi. Sau khi phân tích sâu hơn, chúng ta thấy rằng câu lệnh if hiện ngăn không cho hậu tố có độ dài bằng 0 được thêm vào. Trước khi chúng tôi thực hiện thay đổi, câu lệnh if không hoạt động vì suffixIndex không bao giờ được nhỏ hơn một!

This calls into question *both* ifstatements in compactString! It looks as though they could both be eliminated. So let’s comment them out and run the tests. They passed! So let’s restructure compactStringto eliminate the extraneous ifstatements and make the function much simpler [G9].  
Điều này đặt ra câu hỏi về cả hai câu lệnh if trong compactString! Có vẻ như cả hai đều có thể bị loại bỏ. Vì vậy, hãy nhận xét chúng và chạy thử nghiệm. Họ đã đỗ! Vì vậy, hãy tái cấu trúc compactString để loại bỏ các câu lệnh if không liên quan và làm cho hàm đơn giản hơn nhiều [G9].

private String compactString(String source) { return

computeCommonPrefix() + DELTA\_START +

source.substring(prefixLength, source.length() - suffixLength) + DELTA\_END +

computeCommonSuffix(); }

This is much better! Now we see that the compactStringfunction is simply composing the fragments together. We can probably make this even clearer. Indeed, there are lots of little  
Thế này tốt hơn! Bây giờ chúng ta thấy rằng hàm compactString chỉ đơn giản là kết hợp các đoạn lại với nhau. Có lẽ chúng ta có thể làm cho điều này rõ ràng hơn. Thật vậy, có rất nhiều ít

<https://www.facebook.com/groups/phattrienlaptrinh>

**The JUnit Framework** 263  
Khung JUnit 263

cleanups we could do. But rather than drag you through the rest of the changes, I’ll just show you the result in Listing 15-5.  
dọn dẹp chúng ta có thể làm. Nhưng thay vì kéo bạn qua những thay đổi còn lại, tôi sẽ chỉ cho bạn thấy kết quả trong Liệt kê 15-5.

**Listing 15-5 ComparisonCompactor.java (final)** package junit.framework;  
Liệt kê 15-5 Gói so sánhCompactor.java (cuối cùng) junit.framework;

public class ComparisonCompactor {

private static final String ELLIPSIS = "..."; private static final String DELTA\_END = "]"; private static final String DELTA\_START = "[";

private int contextLength; private String expected; private String actual; private int prefixLength; private int suffixLength;

public ComparisonCompactor(

int contextLength, String expected, String actual ) {

this.contextLength = contextLength; this.expected = expected; this.actual = actual;

}

public String formatCompactedComparison(String message) { String compactExpected = expected;

String compactActual = actual; if (shouldBeCompacted()) {

findCommonPrefixAndSuffix(); compactExpected = compact(expected); compactActual = compact(actual);

}

return Assert.format(message, compactExpected, compactActual); }

private boolean shouldBeCompacted() { return !shouldNotBeCompacted();

}

private boolean shouldNotBeCompacted() { return expected == null ||

actual == null || expected.equals(actual);

}

private void findCommonPrefixAndSuffix() { findCommonPrefix();

suffixLength = 0;

for (; !suffixOverlapsPrefix(); suffixLength++) { if (charFromEnd(expected, suffixLength) !=

charFromEnd(actual, suffixLength) )

<https://www.facebook.com/groups/phattrienlaptrinh>

264 **Chapter 15: JUnit Internals**  
264 Chương 15: Nội bộ JUnit

**Listing 15-5 (continued) ComparisonCompactor.java (final)  
Liệt kê 15-5 (tiếp theo) So sánhCompactor.java (cuối cùng)**

break; }

}

private char charFromEnd(String s, int i) { return s.charAt(s.length() - i - 1);

}

private boolean suffixOverlapsPrefix() {

return actual.length() - suffixLength <= prefixLength || expected.length() - suffixLength <= prefixLength;

}

private void findCommonPrefix() { prefixLength = 0;

int end = Math.min(expected.length(), actual.length()); for (; prefixLength < end; prefixLength++)

if (expected.charAt(prefixLength) != actual.charAt(prefixLength)) break;

}

private String compact(String s) { return new StringBuilder()

.append(startingEllipsis()) .append(startingContext()) .append(DELTA\_START) .append(delta(s)) .append(DELTA\_END) .append(endingContext()) .append(endingEllipsis()) .toString();

}

private String startingEllipsis() {

return prefixLength > contextLength ? ELLIPSIS : ""; }

private String startingContext() {

int contextStart = Math.max(0, prefixLength - contextLength); int contextEnd = prefixLength;

return expected.substring(contextStart, contextEnd); }

private String delta(String s) { int deltaStart = prefixLength;

int deltaEnd = s.length() - suffixLength; return s.substring(deltaStart, deltaEnd);

}

private String endingContext() {

int contextStart = expected.length() - suffixLength; int contextEnd =

Math.min(contextStart + contextLength, expected.length()); return expected.substring(contextStart, contextEnd);

}

<https://www.facebook.com/groups/phattrienlaptrinh>

**Conclusion** 265  
Kết luận 265

**Listing 15-5 (continued) ComparisonCompactor.java (final)**  
Liệt kê 15-5 (tiếp theo) So sánhCompactor.java (cuối cùng)

private String endingEllipsis() {

return (suffixLength > contextLength ? ELLIPSIS : ""); }

}

This is actually quite pretty. The module is separated into a group of analysis func-tions and another group of synthesis functions. They are topologically sorted so that the deﬁnition of each function appears just after it is used. All the analysis functions appear ﬁrst, and all the synthesis functions appear last.  
Điều này thực sự khá đẹp. Mô-đun này được tách thành một nhóm chức năng phân tích và một nhóm chức năng tổng hợp khác. Chúng được sắp xếp tô pô để định nghĩa của mỗi chức năng xuất hiện ngay sau khi nó được sử dụng. Tất cả các chức năng phân tích xuất hiện đầu tiên và tất cả các chức năng tổng hợp xuất hiện sau cùng.

If you look carefully, you will notice that I reversed several of the decisions I made earlier in this chapter. For example, I inlined some extracted methods back into formatCompactedComparison, and I changed the sense of the shouldNotBeCompactedexpres-sion. This is typical. Often one refactoring leads to another that leads to the undoing of the ﬁrst. Refactoring is an iterative process full of trial and error, inevitably converging on something that we feel is worthy of a professional.  
Nếu để ý kỹ, bạn sẽ nhận thấy rằng tôi đã đảo ngược một số quyết định mà tôi đã đưa ra trước đó trong chương này. Ví dụ: tôi đã nội tuyến một số phương thức đã trích xuất trở lại formatCompactedComparison và tôi đã thay đổi ý nghĩa của shouldNotBeCompactedexpres-sion. Đây là điển hình. Thông thường, một lần tái cấu trúc dẫn đến một lần tái cấu trúc khác dẫn đến việc hoàn tác lần đầu tiên. Tái cấu trúc là một quá trình lặp đi lặp lại đầy thử và sai, chắc chắn sẽ hội tụ về điều gì đó mà chúng tôi cảm thấy xứng đáng với một chuyên gia.

[**Conclusion**](#_page_533_0)  
Phần kết luận

And so we have satisﬁed the Boy Scout Rule. We have left this module a bit cleaner than we found it. Not that it wasn’t clean already. The authors had done an excellent job with it. But no module is immune from improvement, and each of us has the responsibility to leave the code a little better than we found it.  
Và như vậy là chúng ta đã thỏa mãn Quy tắc Hướng đạo sinh. Chúng tôi đã để mô-đun này gọn gàng hơn một chút so với những gì chúng tôi tìm thấy. Không phải là nó đã không sạch rồi. Các tác giả đã làm một công việc tuyệt vời với nó. Nhưng không có mô-đun nào miễn nhiễm với sự cải tiến và mỗi chúng ta có trách nhiệm để mã tốt hơn một chút so với những gì chúng ta tìm thấy.

<https://www.facebook.com/groups/phattrienlaptrinh>

*This page intentionally left blank*

<https://www.facebook.com/groups/phattrienlaptrinh>

[**16**](#_page_533_0)

[**Refactoring SerialDate**](#_page_533_0)  
Tái cấu trúc SerialDate

If you go to [http://www.jfree.org/jcommon/index.php,](http://www.jfree.org/jcommon/index.php) you will ﬁnd the JCommon library. Deep within that library there is a package named org.jfree.date. Within that package there is a class named SerialDate. We are going to explore that class.  
Nếu bạn truy cập http://www.jfree.org/jcommon/index.php, bạn sẽ tìm thấy thư viện JCommon. Nằm sâu bên trong thư viện đó có một gói tên là org.jfree.date. Trong gói đó có một lớp có tên là SerialDate. Chúng ta sẽ khám phá lớp học đó.

The author of SerialDateis David Gilbert. David is clearly an experienced and com-petent programmer. As we shall see, he shows a signiﬁcant degree of professionalism and discipline within his code. For all intents and purposes, this is “good code.” And I am going to rip it to pieces.  
Tác giả của SerialDateis David Gilbert. David rõ ràng là một lập trình viên có kinh nghiệm và năng lực. Như chúng ta sẽ thấy, anh ấy thể hiện mức độ chuyên nghiệp và kỷ luật đáng kể trong quy tắc của mình. Đối với tất cả ý định và mục đích, đây là “mã tốt”. Và tôi sẽ xé nó ra từng mảnh.

267

<https://www.facebook.com/groups/phattrienlaptrinh>

268 **Chapter 16: Refactoring SerialDate**  
268 Chương 16: Tái cấu trúc SerialDate

This is not an activity of malice. Nor do I think that I am so much better than David that I somehow have a right to pass judgment on his code. Indeed, if you were to ﬁnd some of my code, I’m sure you could ﬁnd plenty of things to complain about.  
Đây không phải là một hoạt động ác ý. Tôi cũng không nghĩ rằng mình giỏi hơn David đến mức bằng cách nào đó, tôi có quyền đưa ra phán xét về quy tắc của anh ấy. Thật vậy, nếu bạn tìm thấy một số mã của tôi, tôi chắc chắn rằng bạn có thể tìm thấy rất nhiều thứ để phàn nàn.

No, this is not an activity of nastiness or arrogance. What I am about to do is nothing more and nothing less than a professional review. It is something that we should all be comfortable doing. And it is something we should welcome when it is done for us. It is only through critiques like these that we will learn. Doctors do it. Pilots do it. Lawyers do it. And we programmers need to learn how to do it too.  
Không, đây không phải là một hành động xấu xa hay kiêu ngạo. Những gì tôi sắp làm không hơn không kém hơn là một đánh giá chuyên nghiệp. Đó là điều mà tất cả chúng ta nên cảm thấy thoải mái khi làm. Và đó là điều chúng ta nên hoan nghênh khi nó được thực hiện cho chúng ta. Chỉ thông qua những bài phê bình như thế này chúng ta mới học được. Các bác sĩ làm điều đó. Phi công làm điều đó. Luật sư làm việc đó. Và các lập trình viên chúng ta cũng cần học cách làm điều đó.

One more thing about David Gilbert: David is more than just a good programmer. David had the courage and good will to offer his code to the community at large for free. He placed it out in the open for all to see and invited public usage and public scrutiny.This was well done!  
Một điều nữa về David Gilbert: David không chỉ là một lập trình viên giỏi. David đã can đảm và có thiện chí để cung cấp miễn phí mã của mình cho cộng đồng nói chung. Anh ấy đã đặt nó ra ngoài trời để mọi người cùng xem và kêu gọi công chúng sử dụng và giám sát công khai. Điều này đã được thực hiện rất tốt!

SerialDate(Listing B-1, page 349) is a class that represents a date in Java. Why have a class that represents a date, when Java already has java.util.Date and java.util.Calendar, and others? The author wrote this class in response to a pain that I have often felt myself. The comment in his opening Javadoc (line 67) explains it well. We could quibble about his intention, but I have certainly had to deal with this issue, and I welcome a class that is about dates instead of times.  
SerialDate(Danh sách B-1, trang 349) là một lớp đại diện cho một ngày trong Java. Tại sao có một lớp đại diện cho một ngày, khi Java đã có java.util.Date và java.util.Calendar, và những thứ khác? Tác giả đã viết lớp này để đáp lại một nỗi đau mà bản thân tôi thường cảm thấy. Nhận xét trong Javadoc mở đầu của anh ấy (dòng 67) giải thích rõ điều đó. Chúng tôi có thể tranh luận về ý định của anh ấy, nhưng tôi chắc chắn phải giải quyết vấn đề này và tôi hoan nghênh một lớp học về ngày tháng thay vì thời gian.

[**First, Make It Work**](#_page_533_0)  
Đầu tiên, làm cho nó hoạt động

There are some unit tests in a class named SerialDateTests(Listing B-2, page 366). The tests all pass. Unfortunately a quick inspection of the tests shows that they don’t test every-thing [T1]. For example, doing a “Find Usages” search on the method MonthCodeToQuarter (line 334) indicates that it is not used [F4]. Therefore, the unit tests don’t test it.  
Có một số bài kiểm tra đơn vị trong một lớp có tên là SerialDateTests(Danh sách B-2, trang 366). Các bài kiểm tra đều vượt qua. Thật không may, việc kiểm tra nhanh các bài kiểm tra cho thấy rằng họ không kiểm tra mọi thứ [T1]. Ví dụ: thực hiện tìm kiếm “Tìm cách sử dụng” trên phương thức MonthCodeToQuarter (dòng 334) cho biết rằng nó không được sử dụng [F4]. Do đó, các bài kiểm tra đơn vị không kiểm tra nó.

So I ﬁred up Clover to see what the unit tests covered and what they didn’t. Clover reported that the unit tests executed only 91 of the 185 executable statements in SerialDate (~50 percent) [T2]. The coverage map looks like a patchwork quilt, with big gobs of unex-ecuted code littered all through the class.  
Vì vậy, tôi đã kích hoạt Clover để xem bài kiểm tra đơn vị bao gồm những gì và chúng không bao gồm những gì. Clover báo cáo rằng các bài kiểm tra đơn vị chỉ thực hiện 91 trong số 185 câu lệnh thực thi trong SerialDate (~50 phần trăm) [T2]. Bản đồ bảo hiểm trông giống như một tấm chăn chắp vá, với những đoạn mã lớn chưa được thực thi nằm rải rác khắp lớp.

It was my goal to completely understand and also refactor this class. I couldn’t do that without much greater test coverage. So I wrote my own suite of completely independent unit tests (Listing B-4, page 374).  
Mục tiêu của tôi là hiểu hoàn toàn và cũng cấu trúc lại lớp này. Tôi không thể làm điều đó nếu không có phạm vi thử nghiệm lớn hơn nhiều. Vì vậy, tôi đã viết bộ bài kiểm tra đơn vị hoàn toàn độc lập của riêng mình (Danh sách B-4, trang 374).

As you look through these tests, you will note that many of them are commented out. These tests didn’t pass. They represent behavior that I think SerialDate should have. So as I refactor SerialDate, I’ll be working to make these tests pass too.  
Khi bạn xem qua các bài kiểm tra này, bạn sẽ lưu ý rằng nhiều bài kiểm tra trong số đó đã bị loại bỏ. Những bài kiểm tra này đã không vượt qua. Chúng đại diện cho hành vi mà tôi nghĩ SerialDate nên có. Vì vậy, khi tôi cấu trúc lại SerialDate, tôi cũng sẽ làm việc để vượt qua các bài kiểm tra này.

Even with some of the tests commented out, Clover reports that the new unit tests are executing 170 (92 percent) out of the 185 executable statements. This is pretty good, and I think we’ll be able to get this number higher.  
Ngay cả với một số bài kiểm tra đã được nhận xét, Clover báo cáo rằng các bài kiểm tra đơn vị mới đang thực hiện 170 (92 phần trăm) trong số 185 câu lệnh thực thi. Điều này khá tốt và tôi nghĩ chúng ta sẽ có thể đạt được con số này cao hơn.

The ﬁrst few commented-out tests (lines 23-63) were a bit of conceit on my part. The program was not designed to pass these tests, but the behavior seemed obvious [G2] to me.  
Một vài bài kiểm tra nhận xét đầu tiên (dòng 23-63) là một chút tự phụ về phía tôi. Chương trình không được thiết kế để vượt qua các bài kiểm tra này, nhưng hành vi có vẻ rõ ràng [G2] đối với tôi.

<https://www.facebook.com/groups/phattrienlaptrinh>

**First, Make It Work** 269  
Đầu tiên, làm cho nó hoạt động 269

I’m not sure why the testWeekdayCodeToStringmethod was written in the ﬁrst place, but because it is there, it seems obvious that it should not be case sensitive. Writing these tests was trivial [T3]. Making them pass was even easier; I just changed lines 259 and 263 to use equalsIgnoreCase.  
Tôi không chắc tại sao phương thức testWeekdayCodeToString được viết ngay từ đầu, nhưng vì nó ở đó nên rõ ràng là nó không nên phân biệt chữ hoa chữ thường. Viết những bài kiểm tra này là tầm thường [T3]. Làm cho họ vượt qua thậm chí còn dễ dàng hơn; Tôi vừa thay đổi dòng 259 và 263 để sử dụng equalsIgnoreCase.

I left the tests at line 32 and line 45 commented out because it’s not clear to me that the “tues” and “thurs” abbreviations ought to be supported.  
Tôi đã để lại các bài kiểm tra ở dòng 32 và dòng 45 đã nhận xét vì tôi không rõ ràng rằng các chữ viết tắt "tues" và "thurs" nên được hỗ trợ.

The tests on line 153 and line 154 don’t pass. Clearly, they should [G2]. We can easily ﬁx this, and the tests on line 163 through line 213, by making the following changes to the stringToMonthCode function.  
Các bài kiểm tra ở dòng 153 và dòng 154 không vượt qua. Rõ ràng, họ nên [G2]. Chúng ta có thể dễ dàng sửa lỗi này và kiểm tra từ dòng 163 đến dòng 213 bằng cách thực hiện các thay đổi sau đối với hàm stringToMonthCode.

457 if ((result < 1) || (result > 12)) { result = -1;

458 for (int i = 0; i < monthNames.length; i++) { 459 if(s.equalsIgnoreCase(shortMonthNames[i])) { 460 result = i + 1;

461 break; 462 }

463 if (s.equalsIgnoreCase(monthNames[i])) { 464 result = i + 1;

465 break; 466 }

467 } 468 }

The commented test on line 318 exposes a bug in the getFollowingDayOfWeekmethod (line 672). December 25th, 2004, was a Saturday.The following Saturday was January 1st, 2005. However, when we run the test, we see that getFollowingDayOfWeekreturns Decem-ber 25th as the Saturday that follows December 25th. Clearly, this is wrong [G3],[T1]. We see the problem in line 685. It is a typical boundary condition error [T5]. It should read as follows:  
Thử nghiệm đã nhận xét trên dòng 318 cho thấy một lỗi trong phương thức getFollowingDayOfWeek (dòng 672). Ngày 25 tháng 12 năm 2004 là Thứ Bảy. Thứ Bảy tiếp theo là ngày 1 tháng 1 năm 2005. Tuy nhiên, khi chạy thử nghiệm, chúng tôi thấy rằng getFollowingDayOfWeek trả về ngày 25 tháng 12 là Thứ Bảy sau ngày 25 tháng 12. Rõ ràng là sai [G3],[T1]. Chúng tôi thấy vấn đề ở dòng 685. Đây là lỗi điều kiện biên điển hình [T5]. Nó nên đọc như sau:

685 if (baseDOW >= targetWeekday) {

It is interesting to note that this function was the target of an earlier repair. The change history (line 43) shows that “bugs” were ﬁxed in getPreviousDayOfWeek, getFollowing-DayOfWeek, and getNearestDayOfWeek [T6].  
Thật thú vị khi lưu ý rằng chức năng này là mục tiêu của một lần sửa chữa trước đó. Lịch sử thay đổi (dòng 43) cho thấy "lỗi" đã được sửa trong getPreviousDayOfWeek, getFollowing-DayOfWeek và getNearestDayOfWeek [T6].

The testGetNearestDayOfWeek unit test (line 329), which tests the getNearestDayOfWeek method (line 705), did not start out as long and exhaustive as it currently is. I added a lot of test cases to it because my initial test cases did not all pass [T6]. You can see the pattern of failure by looking at which test cases are commented out. That pattern is revealing [T7]. It shows that the algorithm fails if the nearest day is in the future. Clearly there is some kind of boundary condition error [T5].  
Thử nghiệm đơn vị testGetNearestDayOfWeek (dòng 329), thử nghiệm phương thức getNearestDayOfWeek (dòng 705), đã không bắt đầu lâu và đầy đủ như hiện tại. Tôi đã thêm rất nhiều trường hợp thử nghiệm vào đó vì không phải tất cả các trường hợp thử nghiệm ban đầu của tôi đều vượt qua [T6]. Bạn có thể thấy mô hình lỗi bằng cách xem trường hợp thử nghiệm nào được nhận xét. Mẫu đó đang lộ [T7]. Nó chỉ ra rằng thuật toán không thành công nếu ngày gần nhất là trong tương lai. Rõ ràng là có một số loại lỗi điều kiện biên [T5].

The pattern of test coverage reported by Clover is also interesting [T8]. Line 719 never gets executed! This means that the ifstatement in line 718 is always false. Sure enough, a look at the code shows that this must be true. The adjust variable is always neg-ative and so cannot be greater or equal to 4. So this algorithm is just wrong.  
Mô hình phạm vi thử nghiệm do Clover báo cáo cũng rất thú vị [T8]. Dòng 719 không bao giờ được thực thi! Điều này có nghĩa là câu lệnh if trong dòng 718 luôn sai. Chắc chắn rồi, nhìn vào mã cho thấy điều này phải đúng. Biến điều chỉnh luôn âm và do đó không thể lớn hơn hoặc bằng 4. Vì vậy, thuật toán này chỉ sai.

<https://www.facebook.com/groups/phattrienlaptrinh>

270 **Chapter 16: Refactoring SerialDate**  
270 Chương 16: Tái cấu trúc SerialDate

The right algorithm is shown below:  
Thuật toán phù hợp được hiển thị bên dưới:

int delta = targetDOW - base.getDayOfWeek(); int positiveDelta = delta + 7;

int adjust = positiveDelta % 7; if (adjust > 3)

adjust -= 7;

return SerialDate.addDays(adjust, base);

Finally, the tests at line 417 and line 429 can be made to pass simply by throwing an IllegalArgumentExceptioninstead of returning an error string from weekInMonthToString and relativeToString.  
Cuối cùng, các bài kiểm tra ở dòng 417 và dòng 429 có thể được thực hiện đơn giản bằng cách ném một IllegalArgumentException thay vì trả về một chuỗi lỗi từ weekInMonthToString và relativeToString.

With these changes all the unit tests pass, and I believe SerialDatenow works. So now it’s time to make it “right.”  
Với những thay đổi này, tất cả các bài kiểm tra đơn vị đều vượt qua và tôi tin rằng SerialDatenow hoạt động. Vì vậy, bây giờ là lúc để làm cho nó “đúng”.

[**Then Make It Right**](#_page_533_0)  
Sau đó làm cho nó đúng

We are going to walk from the top to the bottom of SerialDate, improving it as we go along. Although you won’t see this in the discussion, I will be running all of the JCommon unit tests, including my improved unit test for SerialDate, after every change I make. So rest assured that every change you see here works for all of JCommon.  
Chúng tôi sẽ đi từ trên xuống dưới của SerialDate, cải thiện nó khi chúng tôi tiếp tục. Mặc dù bạn sẽ không thấy điều này trong cuộc thảo luận, nhưng tôi sẽ chạy tất cả các bài kiểm tra đơn vị JCommon, bao gồm bài kiểm tra đơn vị cải tiến của tôi cho SerialDate, sau mỗi thay đổi tôi thực hiện. Vì vậy, hãy yên tâm rằng mọi thay đổi bạn thấy ở đây đều hoạt động cho tất cả JCommon.

Starting at line 1, we see a ream of comments with license information, copyrights, authors, and change history. I acknowledge that there are certain legalities that need to be addressed, and so the copyrights and licenses must stay. On the other hand, the change his-tory is a leftover from the 1960s. We have source code control tools that do this for us now. This history should be deleted [C1].  
Bắt đầu từ dòng 1, chúng tôi thấy một loạt nhận xét với thông tin giấy phép, bản quyền, tác giả và lịch sử thay đổi. Tôi thừa nhận rằng có một số vấn đề pháp lý nhất định cần được giải quyết, vì vậy bản quyền và giấy phép phải được giữ nguyên. Mặt khác, lịch sử thay đổi của ông là một phần còn sót lại từ những năm 1960. Hiện tại chúng tôi có các công cụ kiểm soát mã nguồn làm việc này cho chúng tôi. Lịch sử này nên bị xóa [C1].

The import list starting at line 61 could be shortened by using java.text.\* and java.util.\*. [J1]  
Danh sách nhập bắt đầu từ dòng 61 có thể được rút ngắn bằng cách sử dụng java.text.\* và java.util.\*. [J1]

I wince at the HTML formatting in the Javadoc (line 67). Having a source ﬁle with more than one language in it troubles me. This comment has *four* languages in it: Java, English, Javadoc, and html [G1]. With that many languages in use, it’s hard to keep things straight. For example, the nice positioning of line 71 and line 72 are lost when the Javadoc is generated, and yet who wants to see <ul>and <li>in the source code? A better strategy might be to just surround the whole comment with <pre>so that the formatting that is apparent in the source code is preserved within the Javadoc.1  
Tôi nhăn mặt với định dạng HTML trong Javadoc (dòng 67). Có một tập tin nguồn với nhiều hơn một ngôn ngữ trong đó làm phiền tôi. Nhận xét này có bốn ngôn ngữ trong đó: Java, tiếng Anh, Javadoc và html [G1]. Với nhiều ngôn ngữ đang được sử dụng, thật khó để giữ cho mọi thứ thẳng thắn. Ví dụ: vị trí đẹp của dòng 71 và dòng 72 bị mất khi Javadoc được tạo, nhưng ai muốn xem <ul>và <li>trong mã nguồn? Một chiến lược tốt hơn có thể là chỉ bao quanh toàn bộ nhận xét bằng <pre>để định dạng rõ ràng trong mã nguồn được giữ nguyên trong Javadoc.1

Line 86 is the class declaration. Why is this class named SerialDate? What is the sig-niﬁcance of the world “serial”? Is it because the class is derived from Serializable? That doesn’t seem likely.  
Dòng 86 là khai báo lớp. Tại sao lớp này có tên là SerialDate? Ý nghĩa của thế giới “nối tiếp” là gì? Có phải vì lớp có nguồn gốc từ Serializable? Điều đó dường như không có khả năng.

1. An even better solution would have been for Javadoc to present all comments as preformatted, so that comments appear the same in both code and document.  
1. Một giải pháp thậm chí còn tốt hơn là Javadoc sẽ trình bày tất cả các nhận xét dưới dạng được định dạng trước, để các nhận xét xuất hiện giống nhau trong cả mã và tài liệu.

<https://www.facebook.com/groups/phattrienlaptrinh>

**Then Make It Right** 271  
Sau Đó Làm Cho Đúng 271

I won’t keep you guessing. I know why (or at least I think I know why) the word “serial” was used. The clue is in the constants SERIAL\_LOWER\_BOUND and SERIAL\_UPPER\_BOUND on line 98 and line 101. An even better clue is in the comment that begins on line 830. This class is named SerialDatebecause it is implemented using a “serial number,” which happens to be the number of days since December 30th, 1899.  
Tôi sẽ không bắt bạn phải đoán. Tôi biết tại sao (hoặc ít nhất tôi nghĩ rằng tôi biết tại sao) từ “nối tiếp” được sử dụng. Manh mối nằm trong các hằng số SERIAL\_LOWER\_BOUND và SERIAL\_UPPER\_BOUND trên dòng 98 và dòng 101. Một manh mối thậm chí còn tốt hơn là trong nhận xét bắt đầu ở dòng 830. Lớp này được đặt tên là SerialDate vì nó được triển khai bằng cách sử dụng một “số sê-ri”, tình cờ là lớp số ngày kể từ ngày 30 tháng 12 năm 1899.

I have two problems with this. First, the term “serial number” is not really correct. This may be a quibble, but the representation is more of a relative offset than a serial num-ber. The term “serial number” has more to do with product identiﬁcation markers than dates. So I don’t ﬁnd this name particularly descriptive [N1]. A more descriptive term might be “ordinal.”  
Tôi có hai vấn đề với điều này. Đầu tiên, thuật ngữ “số sê-ri” không thực sự chính xác. Đây có thể là một sự ngụy biện, nhưng cách biểu diễn giống như một phần bù tương đối hơn là một số sê-ri. Thuật ngữ “số sê-ri” liên quan nhiều đến các dấu hiệu nhận dạng sản phẩm hơn là ngày tháng. Vì vậy, tôi không thấy cái tên này có tính mô tả đặc biệt [N1]. Một thuật ngữ mô tả nhiều hơn có thể là "thứ tự."

The second problem is more signiﬁcant. The name SerialDateimplies an implementa-tion. This class is an abstract class. There is no need to imply anything at all about the implementation. Indeed, there is good reason to hide the implementation! So I ﬁnd this name to be at the wrong level of abstraction [N2]. In my opinion, the name of this class should simply be Date.  
Vấn đề thứ hai quan trọng hơn. Tên SerialDate ngụ ý một triển khai. Lớp này là một lớp trừu tượng. Không cần phải ngụ ý bất cứ điều gì về việc thực hiện. Thật vậy, có lý do chính đáng để che giấu việc thực hiện! Vì vậy, tôi thấy tên này ở mức độ trừu tượng sai [N2]. Theo tôi, tên của lớp này chỉ nên là Date.

Unfortunately, there are already too many classes in the Java library named Date, so this is probably not the best name to choose. Because this class is all about days, instead of time, I considered naming it Day, but this name is also heavily used in other places. In the end, I chose DayDate as the best compromise.  
Thật không may, đã có quá nhiều lớp trong thư viện Java có tên là Ngày, vì vậy đây có lẽ không phải là tên tốt nhất để chọn. Bởi vì lớp này chỉ nói về ngày, thay vì thời gian, tôi đã cân nhắc đặt tên nó là Ngày, nhưng tên này cũng được sử dụng nhiều ở những nơi khác. Cuối cùng, tôi đã chọn DayDate là sự thỏa hiệp tốt nhất.

From now on in this discussion I will use the term DayDate. I leave it to you to remem-ber that the listings you are looking at still use SerialDate.  
Từ bây giờ trong cuộc thảo luận này, tôi sẽ sử dụng thuật ngữ DayDate. Tôi để bạn nhớ rằng danh sách bạn đang xem vẫn sử dụng SerialDate.

I understand why DayDateinherits from Comparableand Serializable. But why does it inherit from MonthConstants? The class MonthConstants(Listing B-3, page 372) is just a bunch of static ﬁnal constants that deﬁne the months. Inheriting from classes with con-stants is an old trick that Java programmers used so that they could avoid using expres-sions like MonthConstants.January, but it’s a bad idea [J2]. MonthConstants should really be an enum.  
Tôi hiểu tại sao DayDate kế thừa từ Có thể so sánh và Có thể nối tiếp. Nhưng tại sao nó kế thừa từ MonthConstants? Lớp MonthConstants(Liệt kê B-3, trang 372) chỉ là một loạt các hằng số tĩnh cuối cùng xác định các tháng. Kế thừa từ các lớp có hằng số là một thủ thuật cũ mà các lập trình viên Java đã sử dụng để họ có thể tránh sử dụng các biểu thức như MonthConstants.Jan, nhưng đó là một ý tưởng tồi [J2]. MonthConstants thực sự phải là một enum.

public abstract class DayDate implements Comparable, Serializable {

public static enum Month { JANUARY(1), FEBRUARY(2),

MARCH(3), APRIL(4), MAY(5), JUNE(6), JULY(7), AUGUST(8), SEPTEMBER(9), OCTOBER(10), NOVEMBER(11), DECEMBER(12);

Month(int index) { this.index = index;

}

<https://www.facebook.com/groups/phattrienlaptrinh>

272 **Chapter 16: Refactoring SerialDate**  
272 Chương 16: Tái cấu trúc SerialDate

public static Month make(int monthIndex) { for (Month m : Month.values()) {

if (m.index == monthIndex) return m;

}

throw new IllegalArgumentException("Invalid month index " + monthIndex); }

public final int index; }

Changing MonthConstantsto this enumforces quite a few changes to the DayDateclass and all it’s users. It took me an hour to make all the changes. However, any function that used to take an intfor a month, now takes a Monthenumerator. This means we can get rid of the isValidMonthCodemethod (line 326), and all the month code error checking such as that in monthCodeToQuarter (line 356) [G5].  
Thay đổi MonthConstants để điều này liệt kê khá nhiều thay đổi đối với DayDateclass và tất cả người dùng của nó. Tôi mất một giờ để thực hiện tất cả các thay đổi. Tuy nhiên, bất kỳ chức năng nào trước đây cần một số nguyên trong một tháng, giờ đây cần một Số tháng. Điều này có nghĩa là chúng ta có thể loại bỏ phương thức isValidMonthCodemethod (dòng 326) và tất cả việc kiểm tra lỗi mã tháng, chẳng hạn như trong monthCodeToQuarter (dòng 356) [G5].

Next, we have line 91, serialVersionUID. This variable is used to control the serializer. If we change it, then any DayDatewritten with an older version of the software won’t be readable anymore and will result in an InvalidClassException. If you don’t declare the serialVersionUIDvariable, then the compiler automatically generates one for you, and it will be different every time you make a change to the module. I know that all the docu-ments recommend manual control of this variable, but it seems to me that automatic con-trol of serialization is a lot safer [G4]. After all, I’d much rather debug an InvalidClassExceptionthan the odd behavior that would ensue if I forgot to change the serialVersionUID. So I’m going to delete the variable—at least for the time being.2  
Tiếp theo, chúng ta có dòng 91, serialVersionUID. Biến này được sử dụng để điều khiển bộ nối tiếp. Nếu chúng tôi thay đổi nó, thì bất kỳ DayDate nào được viết bằng phiên bản phần mềm cũ hơn sẽ không thể đọc được nữa và sẽ dẫn đến InvalidClassException. Nếu bạn không khai báo biến serialVersionUID, thì trình biên dịch sẽ tự động tạo một biến cho bạn và nó sẽ khác mỗi khi bạn thực hiện thay đổi đối với mô-đun. Tôi biết rằng tất cả các tài liệu đều khuyến nghị kiểm soát thủ công biến này, nhưng đối với tôi, có vẻ như kiểm soát tự động lập số sê-ri an toàn hơn rất nhiều [G4]. Rốt cuộc, tôi muốn gỡ lỗi UnlimitedClassException hơn là hành vi kỳ quặc sẽ xảy ra nếu tôi quên thay đổi serialVersionUID. Vì vậy, tôi sẽ xóa biến—ít nhất là trong thời điểm hiện tại.2

I ﬁnd the comment on line 93 redundant. Redundant comments are just places to col-lect lies and misinformation [C2]. So I’m going to get rid of it and its ilk.  
Tôi thấy bình luận trên dòng 93 dư thừa. Bình luận dư thừa chỉ là nơi thu thập thông tin dối trá và sai lệch [C2]. Vì vậy, tôi sẽ loại bỏ nó và ilk của nó.

The comments at line 97 and line 100 talk about serial numbers, which I discussed earlier [C1]. The variables they describe are the earliest and latest possible dates that DayDate can describe. This can be made a bit clearer [N1].  
Các chú thích ở dòng 97 và dòng 100 nói về số sê-ri mà tôi đã thảo luận trước đó [C1]. Các biến mà chúng mô tả là ngày sớm nhất và mới nhất có thể có mà DayDate có thể mô tả. Điều này có thể được làm rõ hơn một chút [N1].

public static final int EARLIEST\_DATE\_ORDINAL = 2; // 1/1/1900 public static final int LATEST\_DATE\_ORDINAL = 2958465; // 12/31/9999

It’s not clear to me why EARLIEST\_DATE\_ORDINALis 2 instead of 0. There is a hint in the comment on line 829 that suggests that this has something to do with the way dates are represented in Microsoft Excel. There is a much deeper insight provided in a derivative of DayDatecalled SpreadsheetDate(Listing B-5, page 382). The comment on line 71 describes the issue nicely.  
Tôi không rõ tại sao EARLIEST\_DATE\_ORDINALis 2 thay vì 0. Có một gợi ý trong nhận xét trên dòng 829 gợi ý rằng điều này có liên quan đến cách trình bày ngày tháng trong Microsoft Excel. Có một cái nhìn sâu sắc hơn nhiều được cung cấp trong một dẫn xuất của DayDategọi là SpreadsheetDate(Danh sách B-5, trang 382). Nhận xét trên dòng 71 mô tả vấn đề độc đáo.

The problem I have with this is that the issue seems to be related to the implementa-tion of SpreadsheetDateand has nothing to do with DayDate. I conclude from this that  
Vấn đề tôi gặp phải với vấn đề này là vấn đề dường như liên quan đến việc triển khai Ngày của Bảng tính và không liên quan gì đến Ngày Ngày. Tôi kết luận từ điều này rằng

2. Several of the reviewers of this text have taken exception to this decision. They contend that in an open source framework it is better to assert manual control over the serial ID so that minor changes to the software don’t cause old serialized dates to be invalid. This is a fair point. However, at least the failure, inconvenient though it might be, has a clear-cut cause. On the other hand, if the author of the class forgets to update the ID, then the failure mode is undeﬁned and might very well be silent. I think the real moral of this story is that you should not expect to deserialize across versions.  
2. Một số người đánh giá văn bản này đã phản đối quyết định này. Họ cho rằng trong một khung mã nguồn mở, tốt hơn là nên khẳng định quyền kiểm soát thủ công đối với ID sê-ri để những thay đổi nhỏ đối với phần mềm không làm cho ngày tháng năm cũ bị mất hiệu lực. Đây là một điểm công bằng. Tuy nhiên, ít nhất sự thất bại, mặc dù có thể bất tiện, có một nguyên nhân rõ ràng. Mặt khác, nếu tác giả của lớp quên cập nhật ID, thì chế độ lỗi không được xác định và rất có thể là im lặng. Tôi nghĩ đạo đức thực sự của câu chuyện này là bạn không nên mong đợi giải tuần tự hóa trên các phiên bản.

<https://www.facebook.com/groups/phattrienlaptrinh>

**Then Make It Right** 273  
Rồi Làm Đúng 273

EARLIEST\_DATE\_ORDINAL and LATEST\_DATE\_ORDINAL do not really belong in DayDate and should be moved to SpreadsheetDate[G6].  
EARLIEST\_DATE\_ORDINAL và LATEST\_DATE\_ORDINAL không thực sự thuộc về DayDate và nên được chuyển sang SpreadsheetDate[G6].

Indeed, a search of the code shows that these variables are used only within SpreadsheetDate. Nothing in DayDate, nor in any other class in the JCommonframework, uses them. Therefore, I’ll move them down into SpreadsheetDate.  
Thật vậy, tìm kiếm mã cho thấy rằng các biến này chỉ được sử dụng trong SpreadsheetDate. Không có gì trong DayDate, cũng như trong bất kỳ lớp nào khác trong JCommonframework, sử dụng chúng. Do đó, tôi sẽ chuyển chúng xuống SpreadsheetDate.

The next variables, MINIMUM\_YEAR\_SUPPORTED, and MAXIMUM\_YEAR\_SUPPORTED (line 104 and line 107), provide something of a dilemma. It seems clear that if DayDate is an abstract class that provides no foreshadowing of implementation, then it should not inform us about a minimum or maximum year. Again, I am tempted to move these variables down into SpreadsheetDate[G6]. However, a quick search of the users of these variables shows that one other class uses them: RelativeDayOfWeekRule(Listing B-6, page 390). We see that usage at line 177 and line 178 in the getDatefunction, where they are used to check that the argument to getDateis a valid year. The dilemma is that a user of an abstract class needs information about its implementation.  
Các biến tiếp theo, MINIMUM\_YEAR\_SUPPORTED và MAXIMUM\_YEAR\_SUPPORTED (dòng 104 và dòng 107), đưa ra một tình huống khó xử. Rõ ràng là nếu DayDate là một lớp trừu tượng không đưa ra điềm báo trước về việc triển khai, thì nó sẽ không thông báo cho chúng ta về năm tối thiểu hoặc tối đa. Một lần nữa, tôi muốn di chuyển các biến này xuống SpreadsheetDate[G6]. Tuy nhiên, một tìm kiếm nhanh những người sử dụng các biến này cho thấy rằng một lớp khác sử dụng chúng:RuleRativeDayOfWeekRule(Danh sách B-6, trang 390). Chúng tôi thấy cách sử dụng đó ở dòng 177 và dòng 178 trong hàm getDate, nơi chúng được sử dụng để kiểm tra xem đối số getDate có phải là một năm hợp lệ hay không. Vấn đề nan giải là người dùng của một lớp trừu tượng cần thông tin về việc triển khai nó.

What we need to do is provide this information without polluting DayDateitself. Usually, we would get implementation information from an instance of a derivative. However, the getDatefunction is not passed an instance of a DayDate. It does, however, return such an instance, which means that somewhere it must be creating it. Line 187 through line 205 provide the hint. The DayDateinstance is being created by one of the three functions, getPreviousDayOfWeek, getNearestDayOfWeek, or getFollowingDayOfWeek. Looking back at the DayDatelisting, we see that these functions (lines 638–724) all return a date created by addDays(line 571), which calls createInstance(line 808), which creates a SpreadsheetDate! [G7].  
Những gì chúng tôi cần làm là cung cấp thông tin này mà không làm ô nhiễm DayDateitself. Thông thường, chúng ta sẽ lấy thông tin triển khai từ một thể hiện của đạo hàm. Tuy nhiên, hàm getDate không được chuyển qua một phiên bản của DayDate. Tuy nhiên, nó trả về một thể hiện như vậy, có nghĩa là nó phải tạo ra nó ở đâu đó. Dòng 187 đến dòng 205 cung cấp gợi ý. Trường hợp DayDate đang được tạo bởi một trong ba hàm, getPreviousDayOfWeek, getNearestDayOfWeek hoặc getFollowingDayOfWeek. Nhìn lại DayDatelisting, chúng ta thấy rằng tất cả các hàm này (dòng 638–724) đều trả về một ngày được tạo bởi addDays(dòng 571), hàm này gọi hàm createInstance(dòng 808), hàm này tạo ra Ngày của bảng tính! [G7].

It’s generally a bad idea for base classes to know about their derivatives. To ﬁx this, we should use the ABSTRACT FACTORY3 pattern and create a DayDateFactory. This factory will create the instances of DayDatethat we need and can also answer questions about the implementation, such as the maximum and minimum dates.  
Nói chung, việc các lớp cơ sở biết về các dẫn xuất của chúng là một ý tưởng tồi. Để khắc phục điều này, chúng ta nên sử dụng mẫu ABSTRACT FACTORY3 và tạo DayDateFactory. Nhà máy này sẽ tạo các phiên bản DayDate mà chúng tôi cần và cũng có thể trả lời các câu hỏi về việc triển khai, chẳng hạn như ngày tối đa và tối thiểu.

public abstract class DayDateFactory {

private static DayDateFactory factory = new SpreadsheetDateFactory(); public static void setInstance(DayDateFactory factory) {

DayDateFactory.factory = factory; }

protected abstract DayDate \_makeDate(int ordinal);

protected abstract DayDate \_makeDate(int day, DayDate.Month month, int year); protected abstract DayDate \_makeDate(int day, int month, int year); protected abstract DayDate \_makeDate(java.util.Date date);

protected abstract int \_getMinimumYear(); protected abstract int \_getMaximumYear();

public static DayDate makeDate(int ordinal) { return factory.\_makeDate(ordinal);

}

3. [GOF].

<https://www.facebook.com/groups/phattrienlaptrinh>

274 **Chapter 16: Refactoring SerialDate**  
274 Chương 16: Tái cấu trúc SerialDate

public static DayDate makeDate(int day, DayDate.Month month, int year) { return factory.\_makeDate(day, month, year);

}

public static DayDate makeDate(int day, int month, int year) { return factory.\_makeDate(day, month, year);

}

public static DayDate makeDate(java.util.Date date) { return factory.\_makeDate(date);

}

public static int getMinimumYear() { return factory.\_getMinimumYear();

}

public static int getMaximumYear() { return factory.\_getMaximumYear();

} }

This factory class replaces the createInstancemethods with makeDatemethods, which improves the names quite a bit [N1]. It defaults to a SpreadsheetDateFactorybut can be changed at any time to use a different factory. The static methods that delegate to abstract methods use a combination of the SINGLETON,4 DECORATOR,5 and ABSTRACT FACTORY patterns that I have found to be useful.  
Lớp xuất xưởng này thay thế các phương thức createInstancemethods bằng makeDatemethods, giúp cải thiện tên khá nhiều [N1]. Nó mặc định là Ngày nhà máy của bảng tính nhưng có thể thay đổi bất kỳ lúc nào để sử dụng nhà máy khác. Các phương thức tĩnh ủy quyền cho các phương thức trừu tượng sử dụng kết hợp các mẫu SINGLETON,4 DECORATOR,5 và ABSTRACT FACTORY mà tôi thấy là hữu ích.

The SpreadsheetDateFactory looks like this.  
SpreadsheetDateFactory trông như thế này.

public class SpreadsheetDateFactory extends DayDateFactory { public DayDate \_makeDate(int ordinal) {

return new SpreadsheetDate(ordinal); }

public DayDate \_makeDate(int day, DayDate.Month month, int year) { return new SpreadsheetDate(day, month, year);

}

public DayDate \_makeDate(int day, int month, int year) { return new SpreadsheetDate(day, month, year);

}

public DayDate \_makeDate(Date date) {

final GregorianCalendar calendar = new GregorianCalendar(); calendar.setTime(date);

return new SpreadsheetDate( calendar.get(Calendar.DATE), DayDate.Month.make(calendar.get(Calendar.MONTH) + 1), calendar.get(Calendar.YEAR));

}

4. Ibid. 5. Ibid.  
4. Sđd. 5. Sđd.

<https://www.facebook.com/groups/phattrienlaptrinh>

**Then Make It Right** 275  
Sau Đó Làm Cho Đúng 275

protected int \_getMinimumYear() {

return SpreadsheetDate.MINIMUM\_YEAR\_SUPPORTED; }

protected int \_getMaximumYear() {

return SpreadsheetDate.MAXIMUM\_YEAR\_SUPPORTED; }

}

As you can see, I have already moved the MINIMUM\_YEAR\_SUPPORTED and MAXIMUM\_YEAR\_SUPPORTED variables into SpreadsheetDate, where they belong [G6].  
Như bạn có thể thấy, tôi đã di chuyển các biến MINIMUM\_YEAR\_SUPPORTED và MAXIMUM\_YEAR\_SUPPORTED vào SpreadsheetDate, nơi chúng thuộc về [G6].

The next issue in DayDateare the day constants beginning at line 109. These should really be another enum [J3]. We’ve seen this pattern before, so I won’t repeat it here. You’ll see it in the ﬁnal listings.  
Vấn đề tiếp theo trong DayDate là các hằng số ngày bắt đầu ở dòng 109. Đây thực sự phải là một enum [J3] khác. Chúng tôi đã thấy mô hình này trước đây, vì vậy tôi sẽ không lặp lại nó ở đây. Bạn sẽ thấy nó trong danh sách cuối cùng.

Next, we see a series of tables starting with LAST\_DAY\_OF\_MONTHat line 140. My ﬁrst issue with these tables is that the comments that describe them are redundant [C3]. Their names are sufﬁcient. So I’m going to delete the comments.  
Tiếp theo, chúng ta thấy một loạt bảng bắt đầu bằng LAST\_DAY\_OF\_MONTHat dòng 140. Vấn đề đầu tiên của tôi với các bảng này là các nhận xét mô tả chúng là dư thừa [C3]. Tên của họ là đủ. Vì vậy, tôi sẽ xóa các bình luận.

There seems to be no good reason that this table isn’t private [G8], because there is a static function lastDayOfMonth that provides the same data.  
Dường như không có lý do chính đáng nào khiến bảng này không ở chế độ riêng tư [G8], bởi vì có một hàm tĩnh lastDayOfMonth cung cấp cùng một dữ liệu.

The next table, AGGREGATE\_DAYS\_TO\_END\_OF\_MONTH, is a bit more mysterious because it is not used anywhere in the JCommon framework [G9]. So I deleted it.  
Bảng tiếp theo, AGGREGATE\_DAYS\_TO\_END\_OF\_MONTH, bí ẩn hơn một chút vì nó không được sử dụng ở bất kỳ đâu trong khung JCommon [G9]. Vì vậy, tôi đã xóa nó.

The same goes for LEAP\_YEAR\_AGGREGATE\_DAYS\_TO\_END\_OF\_MONTH.  
Điều tương tự cũng xảy ra với LEAP\_YEAR\_AGGREGATE\_DAYS\_TO\_END\_OF\_MONTH.

The next table, AGGREGATE\_DAYS\_TO\_END\_OF\_PRECEDING\_MONTH, is used only in Spread-sheetDate(line 434 and line 473). This begs the question of whether it should be moved to SpreadsheetDate. The argument for not moving it is that the table is not speciﬁc to any particular implementation [G6]. On the other hand, no implementation other than SpreadsheetDateactually exists, and so the table should be moved close to where it is used [G10].  
Bảng tiếp theo, AGGREGATE\_DAYS\_TO\_END\_OF\_PRECEDING\_MONTH, chỉ được sử dụng trong Spread-sheetDate(dòng 434 và dòng 473). Điều này đặt ra câu hỏi liệu nó có nên được chuyển sang SpreadsheetDate hay không. Lập luận để không di chuyển nó là bảng không dành riêng cho bất kỳ triển khai cụ thể nào [G6]. Mặt khác, không có triển khai nào khác ngoài SpreadsheetDateactally tồn tại và do đó, bảng phải được di chuyển đến gần nơi nó được sử dụng [G10].

What settles the argument for me is that to be consistent [G11], we should make the table private and expose it through a function like julianDateOfLastDayOfMonth. Nobody seems to need a function like that. Moreover, the table can be moved back to DayDateeasily if any new implementation of DayDate needs it. So I moved it.  
Đối với tôi, điều giải quyết tranh luận là để nhất quán [G11], chúng ta nên đặt bảng ở chế độ riêng tư và hiển thị nó thông qua một hàm như julianDateOfLastDayOfMonth. Dường như không ai cần một chức năng như vậy. Hơn nữa, bảng có thể được chuyển trở lại DayDate một cách dễ dàng nếu bất kỳ triển khai mới nào của DayDate cần nó. Vì vậy, tôi đã di chuyển nó.

The same goes for the table, LEAP\_YEAR\_AGGREGATE\_DAYS\_TO\_END\_OF\_MONTH.  
Điều tương tự cũng xảy ra với bảng LEAP\_YEAR\_AGGREGATE\_DAYS\_TO\_END\_OF\_MONTH.

Next, we see three sets of constants that can be turned into enums (lines 162–205). The ﬁrst of the three selects a week within a month. I changed it into an enum named WeekInMonth.  
Tiếp theo, chúng ta thấy ba bộ hằng số có thể chuyển thành enum (dòng 162–205). Người đầu tiên trong số ba lựa chọn một tuần trong vòng một tháng. Tôi đã đổi nó thành một enum tên là WeekInMonth.

public enum WeekInMonth {  
công khai enum WeekInMonth {

FIRST(1), SECOND(2), THIRD(3), FOURTH(4), LAST(0); public final int index;

WeekInMonth(int index) { this.index = index;

} }

<https://www.facebook.com/groups/phattrienlaptrinh>

276 **Chapter 16: Refactoring SerialDate**  
276 Chương 16: Tái cấu trúc SerialDate

The second set of constants (lines 177–187) is a bit more obscure. The INCLUDE\_NONE, INCLUDE\_FIRST, INCLUDE\_SECOND, and INCLUDE\_BOTHconstants are used to describe whether the deﬁning end-point dates of a range should be included in that range. Mathematically, this is described using the terms “open interval,” “half-open interval,” and “closed inter-val.” I think it is clearer using the mathematical nomenclature [N3], so I changed it to an enum named DateInterval with CLOSED, CLOSED\_LEFT, CLOSED\_RIGHT, and OPEN enumerators.  
Nhóm hằng số thứ hai (dòng 177–187) khó hiểu hơn một chút. Các hằng số INCLUDE\_NONE, INCLUDE\_FIRST, INCLUDE\_SECOND và INCLUDE\_BOTH được sử dụng để mô tả liệu ngày kết thúc xác định của một phạm vi có nên được đưa vào phạm vi đó hay không. Về mặt toán học, điều này được mô tả bằng các thuật ngữ “khoảng mở”, “khoảng nửa mở” và “khoảng đóng”. Tôi nghĩ rằng nó rõ ràng hơn khi sử dụng danh pháp toán học [N3], vì vậy tôi đã đổi nó thành một enum có tên là DateInterval với các điều tra viên CLOSED, CLOSED\_LEFT, CLOSED\_RIGHT và OPEN.

The third set of constants (lines 18–205) describe whether a search for a particular day of the week should result in the last, next, or nearest instance. Deciding what to call this is difﬁcult at best. In the end, I settled for WeekdayRangewith LAST, NEXT, and NEAREST enumerators.  
Nhóm hằng số thứ ba (dòng 18–205) mô tả việc tìm kiếm một ngày cụ thể trong tuần sẽ dẫn đến phiên bản cuối cùng, tiếp theo hay gần nhất. Quyết định gọi điều này là gì là khó khăn nhất. Cuối cùng, tôi giải quyết cho WeekdayRangeRange với các điều tra viên LAST, TIẾP THEO và GẦN NHẤT.

You might not agree with the names I’ve chosen. They make sense to me, but they may not make sense to you. The point is that they are now in a form that makes them easy to change [J3]. They aren’t passed as integers anymore; they are passed as symbols. I can use the “change name” function of my IDE to change the names, or the types, without worrying that I missed some -1or 2somewhere in the code or that some intargument dec-laration is left poorly described.  
Bạn có thể không đồng ý với những cái tên tôi đã chọn. Chúng có ý nghĩa với tôi, nhưng chúng có thể không có ý nghĩa với bạn. Vấn đề là bây giờ chúng ở dạng giúp chúng dễ dàng thay đổi [J3]. Chúng không được chuyển thành số nguyên nữa; chúng được thông qua dưới dạng biểu tượng. Tôi có thể sử dụng chức năng “đổi tên” trong IDE của mình để thay đổi tên hoặc loại mà không phải lo lắng rằng tôi đã bỏ lỡ một số -1 hoặc 2 ở đâu đó trong mã hoặc một số phần khai báo nội dung được mô tả kém.

The description ﬁeld at line 208 does not seem to be used by anyone. I deleted it along with its accessor and mutator [G9].  
Trường mô tả ở dòng 208 dường như không được ai sử dụng. Tôi đã xóa nó cùng với trình truy cập và trình biến đổi [G9].

I also deleted the degenerate default constructor at line 213 [G12]. The compiler will generate it for us.  
Tôi cũng đã xóa hàm tạo mặc định suy biến ở dòng 213 [G12]. Trình biên dịch sẽ tạo ra nó cho chúng ta.

We can skip over the isValidWeekdayCodemethod (lines 216–238) because we deleted it when we created the Day enumeration.  
Chúng ta có thể bỏ qua phương thức isValidWeekdayCode (dòng 216–238) vì chúng ta đã xóa nó khi tạo phép liệt kê Ngày.

This brings us to the stringToWeekdayCode method (lines 242–270). Javadocs that don’t add much to the method signature are just clutter [C3],[G12]. The only value this Javadoc adds is the description of the -1return value. However, because we changed to the Day enumeration, the comment is actually wrong [C2]. The method now throws an IllegalArgumentException. So I deleted the Javadoc.  
Điều này đưa chúng ta đến phương thức stringToWeekdayCode (dòng 242–270). Javadocs không bổ sung nhiều vào chữ ký phương thức chỉ là sự lộn xộn [C3],[G12]. Giá trị duy nhất mà Javadoc này thêm vào là mô tả về giá trị -1return. Tuy nhiên, do ta đổi sang kiểu liệt kê Ngày nên nhận xét thực ra là sai [C2]. Phương thức hiện ném một IllegalArgumentException. Vì vậy, tôi đã xóa Javadoc.

I also deleted all the final keywords in arguments and variable declarations. As far as I could tell, they added no real value but did add to the clutter [G12]. Eliminating final ﬂies in the face of some conventional wisdom. For example, Robert Simmons6 strongly recommends us to “. . . spread finalall over your code.” Clearly I disagree. I think that there are a few good uses for final, such as the occasional finalconstant, but otherwise the keyword adds little value and creates a lot of clutter. Perhaps I feel this way because the kinds of errors that final might catch are already caught by the unit tests I write.  
Tôi cũng đã xóa tất cả các từ khóa cuối cùng trong các đối số và khai báo biến. Theo những gì tôi có thể nói, chúng không mang lại giá trị thực nào nhưng lại làm tăng thêm sự lộn xộn [G12]. Loại bỏ những con ruồi cuối cùng khi đối mặt với một số sự khôn ngoan thông thường. Ví dụ: Robert Simmons6 thực sự khuyên chúng ta nên “. . . trải rộng cuối cùng trên mã của bạn. Rõ ràng là tôi không đồng ý. Tôi nghĩ rằng có một số cách sử dụng tốt cho từ cuối cùng, chẳng hạn như hằng số cuối cùng không thường xuyên, nhưng nếu không thì từ khóa này chỉ thêm ít giá trị và tạo ra nhiều lộn xộn. Có lẽ tôi cảm thấy như vậy bởi vì các loại lỗi mà final có thể bắt gặp đã bị bắt bởi các bài kiểm tra đơn vị mà tôi viết.

I didn’t care for the duplicate ifstatements [G5] inside the forloop (line 259 and line 263), so I connected them into a single ifstatement using the ||operator. I also used the Day enumeration to direct the for loop and made a few other cosmetic changes.  
Tôi không quan tâm đến các câu lệnh if trùng lặp [G5] bên trong forloop (dòng 259 và dòng 263), vì vậy tôi đã kết nối chúng thành một câu lệnh if duy nhất bằng cách sử dụng toán tử ||. Tôi cũng đã sử dụng phép liệt kê Ngày để điều khiển vòng lặp for và thực hiện một số thay đổi về mặt thẩm mỹ khác.

It occurred to me that this method does not really belong in DayDate. It’s really the parse function of Day. So I moved it into the Dayenumeration. However, that made the Day  
Tôi chợt nhận ra rằng phương pháp này không thực sự thuộc về DayDate. Nó thực sự là chức năng phân tích cú pháp của Ngày. Vì vậy, tôi đã chuyển nó vào Bảng đếm ngày. Tuy nhiên, điều đó đã làm cho Ngày

6. [Simmons04], p. 73.

<https://www.facebook.com/groups/phattrienlaptrinh>

**Then Make It Right** 277  
Rồi Làm Đúng 277

enumeration pretty large. Because the concept of Daydoes not depend on DayDate, I moved the Day enumeration outside of the DayDate class into its own source ﬁle [G13].  
liệt kê khá lớn. Bởi vì khái niệm về Ngày không phụ thuộc vào Ngày, nên tôi đã chuyển phép liệt kê Ngày ra ngoài lớp DayDate vào tệp nguồn riêng của nó [G13].

I also moved the next function, weekdayCodeToString(lines 272–286) into the Day enumeration and called it toString.  
Tôi cũng đã chuyển hàm tiếp theo, weekdayCodeToString(lines 272–286) vào kiểu liệt kê Day và gọi nó là toString.

public enum Day { MONDAY(Calendar.MONDAY), TUESDAY(Calendar.TUESDAY), WEDNESDAY(Calendar.WEDNESDAY),s THURSDAY(Calendar.THURSDAY), FRIDAY(Calendar.FRIDAY), SATURDAY(Calendar.SATURDAY), SUNDAY(Calendar.SUNDAY);

public final int index;

private static DateFormatSymbols dateSymbols = new DateFormatSymbols();

Day(int day) { index = day;

}

public static Day make(int index) throws IllegalArgumentException { for (Day d : Day.values())

if (d.index == index) return d;

throw new IllegalArgumentException( String.format("Illegal day index: %d.", index));

}

public static Day parse(String s) throws IllegalArgumentException { String[] shortWeekdayNames =

dateSymbols.getShortWeekdays(); String[] weekDayNames =

dateSymbols.getWeekdays();

s = s.trim();

for (Day day : Day.values()) {

if (s.equalsIgnoreCase(shortWeekdayNames[day.index]) || s.equalsIgnoreCase(weekDayNames[day.index])) {

return day; }

}

throw new IllegalArgumentException(

String.format("%s is not a valid weekday string", s)); }

public String toString() {

return dateSymbols.getWeekdays()[index]; }

}

There are two getMonthsfunctions (lines 288–316). The ﬁrst calls the second. The second is never called by anyone but the ﬁrst. Therefore, I collapsed the two into one and vastly simpliﬁed them [G9],[G12],[F4]. Finally, I changed the name to be a bit more self-descriptive [N1].  
Có hai hàm getMonth (dòng 288–316). Người đầu tiên gọi người thứ hai. Cái thứ hai không bao giờ được gọi bởi bất cứ ai ngoài cái đầu tiên. Do đó, tôi đã gộp cả hai thành một và đơn giản hóa chúng rất nhiều [G9],[G12],[F4]. Cuối cùng, tôi đã đổi tên cho tự mô tả hơn một chút [N1].

<https://www.facebook.com/groups/phattrienlaptrinh>

278 **Chapter 16: Refactoring SerialDate**  
278 Chương 16: Tái cấu trúc SerialDate

public static String[] getMonthNames() { return dateFormatSymbols.getMonths();

}

The isValidMonthCode function (lines 326–346) was made irrelevant by the Month enum, so I deleted it [G9].  
Hàm isValidMonthCode (dòng 326–346) không còn phù hợp theo Tháng enum, vì vậy tôi đã xóa nó [G9].

The monthCodeToQuarterfunction (lines 356–375) smells of FEATURE ENVY7 [G14] and probably belongs in the Month enum as a method named quarter. So I replaced it.  
Hàm monthCodeToQuarter (dòng 356–375) có mùi của FEATURE ENVY7 [G14] và có thể thuộc về enum Tháng dưới dạng phương thức có tên quarter. Vì vậy, tôi đã thay thế nó.

public int quarter() { return 1 + (index-1)/3;  
public int quarter() { return 1 + (index-1)/3;

}

This made the Monthenum big enough to be in its own class. So I moved it out of DayDate to be consistent with the Day enum [G11],[G13].  
Điều này làm cho Monthenum đủ lớn để được xếp vào hạng riêng của nó. Vì vậy, tôi đã chuyển nó ra khỏi DayDate để phù hợp với Day enum [G11],[G13].

The next two methods are named monthCodeToString(lines 377–426). Again, we see the pattern of one method calling its twin with a ﬂag. It is usually a bad idea to pass a ﬂag as an argument to a function, especially when that ﬂag simply selects the format of the out-put [G15]. I renamed, simpliﬁed, and restructured these functions and moved them into the Month enum [N1],[N3],[C3],[G14].  
Hai phương pháp tiếp theo được đặt tên là monthCodeToString(dòng 377–426). Một lần nữa, chúng ta thấy mô hình của một phương thức gọi người song sinh của nó bằng một cờ. Việc chuyển một cờ làm đối số cho một hàm thường là một ý tưởng tồi, đặc biệt khi cờ đó chỉ đơn giản là chọn định dạng của đầu ra [G15]. Tôi đã đổi tên, đơn giản hóa và tái cấu trúc các chức năng này và chuyển chúng vào Tháng enum [N1],[N3],[C3],[G14].

public String toString() {

return dateFormatSymbols.getMonths()[index - 1]; }

public String toShortString() {

return dateFormatSymbols.getShortMonths()[index - 1]; }

The next method is stringToMonthCode(lines 428–472). I renamed it, moved it into the Month enum, and simpliﬁed it [N1],[N3],[C3],[G14],[G12].  
Phương thức tiếp theo là stringToMonthCode(dòng 428–472). Tôi đã đổi tên nó, chuyển nó vào Tháng enum và đơn giản hóa nó [N1],[N3],[C3],[G14],[G12].

public static Month parse(String s) { s = s.trim();

for (Month m : Month.values()) if (m.matches(s))

return m;

try {

return make(Integer.parseInt(s)); }

catch (NumberFormatException e) {}

throw new IllegalArgumentException("Invalid month " + s); }

7. [Refactoring].  
7. [Tái cấu trúc].

<https://www.facebook.com/groups/phattrienlaptrinh>

**Then Make It Right** 279  
Rồi Làm Đúng 279

private boolean matches(String s) { return s.equalsIgnoreCase(toString()) ||

s.equalsIgnoreCase(toShortString()); }

The isLeapYear method (lines 495–517) can be made a bit more expressive [G16].  
Phương thức isLeapYear (dòng 495–517) có thể biểu cảm hơn một chút [G16].

public static boolean isLeapYear(int year) { boolean fourth = year % 4 == 0;

boolean hundredth = year % 100 == 0; boolean fourHundredth = year % 400 == 0;

return fourth && (!hundredth || fourHundredth); }

The next function, leapYearCount(lines 519–536) doesn’t really belong in DayDate. Nobody calls it except for two methods in SpreadsheetDate. So I pushed it down [G6].  
Hàm tiếp theo, LeapYearCount(dòng 519–536) không thực sự thuộc về DayDate. Không ai gọi nó ngoại trừ hai phương thức trong SpreadsheetDate. Nên tôi đẩy nó xuống [G6].

The lastDayOfMonth function (lines 538–560) makes use of the LAST\_DAY\_OF\_MONTH array.This array really belongs in the Month enum [G17], so I moved it there. I also simpli-ﬁed the function and made it a bit more expressive [G16].  
Hàm lastDayOfMonth (dòng 538–560) sử dụng mảng LAST\_DAY\_OF\_MONTH. Mảng này thực sự thuộc về Tháng enum [G17], vì vậy tôi đã chuyển nó đến đó. Tôi cũng đã đơn giản hóa hàm này và làm cho nó biểu cảm hơn một chút [G16].

public static int lastDayOfMonth(Month month, int year) { if (month == Month.FEBRUARY && isLeapYear(year))

return month.lastDay() + 1; else

return month.lastDay(); }

Now things start to get a bit more interesting. The next function is addDays(lines 562– 576). First of all, because this function operates on the variables of DayDate, it should not be static [G18]. So I changed it to an instance method. Second, it calls the function toSerial. This function should be renamed toOrdinal[N1]. Finally, the method can be simpliﬁed.  
Bây giờ mọi thứ bắt đầu trở nên thú vị hơn một chút. Hàm tiếp theo là addDays(dòng 562–576). Trước hết, vì chức năng này hoạt động trên các biến của DayDate, nên nó không tĩnh [G18]. Vì vậy, tôi đã thay đổi nó thành một phương thức thể hiện. Thứ hai, nó gọi hàm toSerial. Chức năng này nên được đổi tên thànhOrdinal[N1]. Cuối cùng, phương pháp này có thể được đơn giản hóa.

public DayDate addDays(int days) {

return DayDateFactory.makeDate(toOrdinal() + days); }

The same goes for addMonths(lines 578–602). It should be an instance method [G18]. The algorithm is a bit complicated, so I used EXPLAINING TEMPORARY VARIABLES8 [G19] to make it more transparent. I also renamed the method getYYY to getYear [N1].  
Điều tương tự cũng xảy ra với addMonths(dòng 578–602). Nó phải là một phương thức thể hiện [G18]. Thuật toán hơi phức tạp nên tôi đã sử dụng GIẢI THÍCH BIẾN TẠM THỜI8 [G19] để minh bạch hơn. Tôi cũng đã đổi tên phương thức getYYY thành getYear [N1].

public DayDate addMonths(int months) {

int thisMonthAsOrdinal = 12 \* getYear() + getMonth().index - 1; int resultMonthAsOrdinal = thisMonthAsOrdinal + months; int resultYear = resultMonthAsOrdinal / 12;

Month resultMonth = Month.make(resultMonthAsOrdinal % 12 + 1);

8. [Beck97].

<https://www.facebook.com/groups/phattrienlaptrinh>

280 **Chapter 16: Refactoring SerialDate**  
280 Chương 16: Tái cấu trúc SerialDate

int lastDayOfResultMonth = lastDayOfMonth(resultMonth, resultYear); int resultDay = Math.min(getDayOfMonth(), lastDayOfResultMonth); return DayDateFactory.makeDate(resultDay, resultMonth, resultYear);

}

The addYears function (lines 604–626) provides no surprises over the others.

public DayDate plusYears(int years) { int resultYear = getYear() + years;

int lastDayOfMonthInResultYear = lastDayOfMonth(getMonth(), resultYear); int resultDay = Math.min(getDayOfMonth(), lastDayOfMonthInResultYear); return DayDateFactory.makeDate(resultDay, getMonth(), resultYear);

}

There is a little itch at the back of my mind that is bothering me about changing these methods from static to instance. Does the expression date.addDays(5) make it clear that the dateobject does not change and that a new instance of DayDateis returned? Or does it erroneously imply that we are adding ﬁve days to the dateobject? You might not think that is a big problem, but a bit of code that looks like the following can be very deceiving [G20].  
Có một chút ngứa ngáy trong tâm trí tôi đang làm phiền tôi về việc thay đổi các phương thức này từ tĩnh sang thể hiện. Biểu thức date.addDays(5) có làm rõ rằng đối tượng ngày không thay đổi và phiên bản mới của DayDateis được trả về không? Hay nó ngụ ý sai rằng chúng ta đang thêm năm ngày vào đối tượng ngày tháng? Bạn có thể không nghĩ đó là một vấn đề lớn, nhưng một đoạn mã giống như sau có thể rất dễ đánh lừa [G20].

DayDate date = DateFactory.makeDate(5, Month.DECEMBER, 1952); date.addDays(7); // bump date by one week.

Someone reading this code would very likely just accept that addDaysis changing the dateobject. So we need a name that breaks this ambiguity [N4]. So I changed the names to plusDays and plusMonths. It seems to me that the intent of the method is captured nicely by  
Ai đó đọc mã này rất có thể sẽ chấp nhận rằng addDaysis thay đổi dateobject. Vì vậy, chúng tôi cần một cái tên phá vỡ sự mơ hồ này [N4]. Vì vậy, tôi đã đổi tên thành plusDays và plusMonths. Đối với tôi, dường như mục đích của phương pháp được nắm bắt một cách độc đáo bởi

DayDate date = oldDate.plusDays(5);

whereas the following doesn’t read ﬂuidly enough for a reader to simply accept that the date object is changed:  
trong khi những điều sau đây đọc không đủ trôi chảy để người đọc chỉ đơn giản chấp nhận rằng đối tượng ngày tháng đã bị thay đổi:

date.plusDays(5);

The algorithms continue to get more interesting. getPreviousDayOfWeek(lines 628– 660) works but is a bit complicated. After some thought about what was really going on [G21], I was able to simplify it and use EXPLAINING TEMPORARY VARIABLES [G19] to make it clearer. I also changed it from a static method to an instance method [G18], and got rid of the duplicate instance method [G5] (lines 997–1008).  
Các thuật toán tiếp tục trở nên thú vị hơn. getPreviousDayOfWeek(lines 628–660) hoạt động nhưng hơi phức tạp. Sau khi suy nghĩ về những gì đang thực sự xảy ra [G21], tôi đã có thể đơn giản hóa nó và sử dụng GIẢI THÍCH CÁC BIẾN SỐ TẠM THỜI [G19] để làm cho nó rõ ràng hơn. Tôi cũng đã thay đổi nó từ một phương thức tĩnh thành một phương thức thể hiện [G18] và loại bỏ phương thức thể hiện trùng lặp [G5] (dòng 997–1008).

public DayDate getPreviousDayOfWeek(Day targetDayOfWeek) {

int offsetToTarget = targetDayOfWeek.index - getDayOfWeek().index; if (offsetToTarget >= 0)

offsetToTarget -= 7;

return plusDays(offsetToTarget); }

The exact same analysis and result occurred for getFollowingDayOfWeek(lines 662–693).  
Phân tích và kết quả chính xác tương tự xảy ra với getFollowingDayOfWeek(dòng 662–693).

public DayDate getFollowingDayOfWeek(Day targetDayOfWeek) {

int offsetToTarget = targetDayOfWeek.index - getDayOfWeek().index; if (offsetToTarget <= 0)

<https://www.facebook.com/groups/phattrienlaptrinh>

**Then Make It Right** 281  
281

offsetToTarget += 7;

return plusDays(offsetToTarget); }

The next function is getNearestDayOfWeek(lines 695–726), which we corrected back on page 270. But the changes I made back then aren’t consistent with the current pattern in the last two functions [G11]. So I made it consistent and used some EXPLAINING TEMPO-RARY VARIABLES [G19] to clarify the algorithm.  
Hàm tiếp theo là getNearestDayOfWeek(dòng 695–726), chúng tôi đã sửa lại ở trang 270. Nhưng những thay đổi mà tôi đã thực hiện sau đó không nhất quán với mẫu hiện tại trong hai hàm cuối cùng [G11]. Vì vậy, tôi đã làm cho nó nhất quán và sử dụng một số GIẢI THÍCH BIẾN SỐ NHIỆT ĐỘ [G19] để làm rõ thuật toán.

public DayDate getNearestDayOfWeek(final Day targetDay) {

int offsetToThisWeeksTarget = targetDay.index - getDayOfWeek().index; int offsetToFutureTarget = (offsetToThisWeeksTarget + 7) % 7;

int offsetToPreviousTarget = offsetToFutureTarget - 7;

if (offsetToFutureTarget > 3)

return plusDays(offsetToPreviousTarget); else

return plusDays(offsetToFutureTarget); }

The getEndOfCurrentMonthmethod (lines 728–740) is a little strange because it is an instance method that envies [G14] its own class by taking a DayDateargument. I made it a true instance method and clariﬁed a few names.  
getEndOfCurrentMonthmethod (dòng 728–740) hơi lạ vì nó là một phương thức thể hiện ghen tị với [G14] lớp của chính nó bằng cách lấy một DayDateargument. Tôi đã biến nó thành một phương thức thực thể và làm rõ một vài cái tên.

public DayDate getEndOfMonth() { Month month = getMonth(); int year = getYear();

int lastDay = lastDayOfMonth(month, year);

return DayDateFactory.makeDate(lastDay, month, year); }

Refactoring weekInMonthToString(lines 742–761) turned out to be very interesting indeed. Using the refactoring tools of my IDE, I ﬁrst moved the method to the WeekInMonth enum that I created back on page 275. Then I renamed the method to toString. Next, I changed it from a static method to an instance method. All the tests still passed. (Can you guess where I am going?)  
Việc tái cấu trúc weekInMonthToString(lines 742–761) thực sự rất thú vị. Sử dụng các công cụ tái cấu trúc của IDE, trước tiên tôi chuyển phương thức này sang enum WeekInMonth mà tôi đã tạo ở trang 275. Sau đó, tôi đổi tên phương thức thành toString. Tiếp theo, tôi đã thay đổi nó từ một phương thức tĩnh thành một phương thức thể hiện. Tất cả các bài kiểm tra vẫn thông qua. (Bạn có đoán được tôi đang đi đâu không?)

Next, I deleted the method entirely! Five asserts failed (lines 411–415, Listing B-4, page 374). I changed these lines to use the names of the enumerators (FIRST, SECOND, . . .). All the tests passed. Can you see why? Can you also see why each of these steps was necessary? The refactoring tool made sure that all previous callers of weekInMonthToStringnow called toStringon the weekInMonthenumerator because all enu-merators implement toString to simply return their names. . . .  
Tiếp theo, tôi đã xóa hoàn toàn phương thức này! Năm xác nhận không thành công (dòng 411–415, Liệt kê B-4, trang 374). Tôi đã thay đổi những dòng này để sử dụng tên của các điều tra viên (FIRST, SECOND, . . . .). Tất cả các bài kiểm tra đã thông qua. Bạn có thể thấy tại sao? Bạn cũng có thể thấy tại sao mỗi bước này là cần thiết không? Công cụ tái cấu trúc đảm bảo rằng tất cả những người gọi trước đó của weekInMonthToStringnow đã gọi toStringon the weekInMonthenumerator bởi vì tất cả các bộ đếm thực hiện toString để chỉ trả lại tên của họ. . . .

Unfortunately, I was a bit too clever. As elegant as that wonderful chain of refactor-ings was, I ﬁnally realized that the only users of this function were the tests I had just mod-iﬁed, so I deleted the tests.  
Thật không may, tôi đã hơi quá thông minh. Chuỗi tái cấu trúc tuyệt vời đó rất tao nhã, cuối cùng tôi nhận ra rằng những người dùng duy nhất của chức năng này là các bài kiểm tra mà tôi vừa sửa đổi, vì vậy tôi đã xóa các bài kiểm tra.

Fool me once, shame on you. Fool me twice, shame on me! So after determining that nobody other than the tests called relativeToString(lines 765–781), I simply deleted the function and its tests.  
Đánh lừa tôi một lần, xấu hổ về bạn. Lừa tôi hai lần, xấu hổ về tôi! Vì vậy, sau khi xác định rằng không ai khác ngoài các bài kiểm tra được gọi là relativeToString(dòng 765–781), tôi chỉ cần xóa hàm và các bài kiểm tra của nó.

<https://www.facebook.com/groups/phattrienlaptrinh>

282 **Chapter 16: Refactoring SerialDate**  
282 Chương 16: Tái cấu trúc SerialDate

We have ﬁnally made it to the abstract methods of this abstract class. And the ﬁrst one is as appropriate as they come: toSerial(lines 838–844). Back on page 279 I had changed the name to toOrdinal. Having looked at it in this context, I decided the name should be changed to getOrdinalDay.  
Cuối cùng chúng ta đã làm được với các phương thức trừu tượng của lớp trừu tượng này. Và cái đầu tiên là phù hợp khi chúng xuất hiện: toSerial(dòng 838–844). Quay lại trang 279 tôi đã đổi tên thành toOrdinal. Sau khi xem xét nó trong bối cảnh này, tôi quyết định nên đổi tên thành getOrdinalDay.

The next abstract method is toDate (lines 838–844). It converts a DayDate to a java.util.Date. Why is this method abstract? If we look at its implementation in SpreadsheetDate(lines 198–207, Listing B-5, page 382), we see that it doesn’t depend on anything in the implementation of that class [G6]. So I pushed it up.  
Phương thức trừu tượng tiếp theo là toDate (dòng 838–844). Nó chuyển đổi DayDate thành java.util.Date. Tại sao phương pháp này trừu tượng? Nếu xem xét cách triển khai của nó trong SpreadsheetDate(dòng 198–207, Liệt kê B-5, trang 382), chúng ta sẽ thấy rằng nó không phụ thuộc vào bất kỳ thứ gì trong quá trình triển khai của lớp đó [G6]. Vì vậy, tôi đẩy nó lên.

The getYYYY, getMonth, and getDayOfMonthmethods are nicely abstract. However, the getDayOfWeek method is another one that should be pulled up from SpreadSheetDate because it doesn’t depend on anything that can’t be found in DayDate [G6]. Or does it?  
Các phương thức getYYYY, getMonth và getDayOfMonth khá trừu tượng. Tuy nhiên, phương thức getDayOfWeek là một phương thức khác nên được lấy từ SpreadSheetDate vì nó không phụ thuộc vào bất kỳ thứ gì không thể tìm thấy trong DayDate [G6]. Hay không?

If you look carefully (line 247, Listing B-5, page 382), you’ll see that the algorithm implicitly depends on the origin of the ordinal day (in other words, the day of the week of day 0). So even though this function has no physical dependencies that couldn’t be moved to DayDate, it does have a logical dependency.  
Nếu bạn xem kỹ (dòng 247, Liệt kê B-5, trang 382), bạn sẽ thấy rằng thuật toán hoàn toàn phụ thuộc vào nguồn gốc của ngày thứ tự (nói cách khác, ngày trong tuần của ngày 0). Vì vậy, mặc dù chức năng này không có phụ thuộc vật lý không thể chuyển sang DayDate, nhưng nó có phụ thuộc logic.

Logical dependencies like this bother me [G22]. If something logical depends on the implementation, then something physical should too. Also, it seems to me that the algorithm itself could be generic with a much smaller portion of it dependent on the implementation [G6].  
Các phụ thuộc logic như thế này làm phiền tôi [G22]. Nếu một cái gì đó hợp lý phụ thuộc vào việc triển khai, thì một cái gì đó vật lý cũng vậy. Ngoài ra, đối với tôi, dường như bản thân thuật toán có thể chung chung với một phần nhỏ hơn nhiều của nó phụ thuộc vào việc triển khai [G6].

So I created an abstract method in DayDatenamed getDayOfWeekForOrdinalZeroand implemented it in SpreadsheetDateto return Day.SATURDAY. Then I moved the getDayOfWeek method up to DayDateand changed it to call getOrdinalDayand getDayOfWeekForOrdinal-Zero.  
Vì vậy, tôi đã tạo một phương thức trừu tượng trong DayDatenamed getDayOfWeekForOrdinalZero và triển khai nó trong SpreadsheetDate để trả về Day.SATURDAY. Sau đó, tôi đã chuyển phương thức getDayOfWeek lên DayDate và thay đổi nó để gọi getOrdinalDay và getDayOfWeekForOrdinal-Zero.

public Day getDayOfWeek() {

Day startingDay = getDayOfWeekForOrdinalZero();

int startingOffset = startingDay.index - Day.SUNDAY.index; return Day.make((getOrdinalDay() + startingOffset) % 7 + 1);

}

As a side note, look carefully at the comment on line 895 through line 899. Was this repetition really necessary? As usual, I deleted this comment along with all the others.  
Ngoài ra, hãy xem kỹ nhận xét từ dòng 895 đến dòng 899. Sự lặp lại này có thực sự cần thiết không? Như thường lệ, tôi đã xóa bình luận này cùng với tất cả những bình luận khác.

The next method is compare(lines 902–913). Again, this method is inappropriately abstract [G6], so I pulled the implementation up into DayDate. Also, the name does not communicate enough [N1]. This method actually returns the difference in days since the argument. So I changed the name to daysSince. Also, I noted that there weren’t any tests for this method, so I wrote them.  
Phương pháp tiếp theo là so sánh (dòng 902–913). Một lần nữa, phương pháp này trừu tượng không phù hợp [G6], vì vậy tôi đã triển khai triển khai vào DayDate. Ngoài ra, tên không đủ giao tiếp [N1]. Phương thức này thực sự trả về sự khác biệt về số ngày kể từ đối số. Vì vậy, tôi đã đổi tên thành daysSince. Ngoài ra, tôi lưu ý rằng không có bất kỳ bài kiểm tra nào cho phương pháp này, vì vậy tôi đã viết chúng.

The next six functions (lines 915–980) are all abstract methods that should be imple-mented in DayDate. So I pulled them all up from SpreadsheetDate.  
Sáu hàm tiếp theo (dòng 915–980) đều là các phương thức trừu tượng nên được triển khai trong DayDate. Vì vậy, tôi đã lấy tất cả chúng từ Ngày bảng tính.

The last function, isInRange(lines 982–995) also needs to be pulled up and refac-tored. The switchstatement is a bit ugly [G23] and can be replaced by moving the cases into the DateInterval enum.  
Hàm cuối cùng, isInRange(dòng 982–995) cũng cần được kéo lên và cấu trúc lại. Câu lệnh chuyển đổi hơi xấu [G23] và có thể được thay thế bằng cách di chuyển các trường hợp vào DateInterval enum.

<https://www.facebook.com/groups/phattrienlaptrinh>

**Then Make It Right** 283  
283

public enum DateInterval { OPEN {

public boolean isIn(int d, int left, int right) { return d > left && d < right;

} },

CLOSED\_LEFT {

public boolean isIn(int d, int left, int right) { return d >= left && d < right;

} },

CLOSED\_RIGHT {

public boolean isIn(int d, int left, int right) { return d > left && d <= right;

} },

CLOSED {

public boolean isIn(int d, int left, int right) { return d >= left && d <= right;

} };

public abstract boolean isIn(int d, int left, int right); }

public boolean isInRange(DayDate d1, DayDate d2, DateInterval interval) { int left = Math.min(d1.getOrdinalDay(), d2.getOrdinalDay());

int right = Math.max(d1.getOrdinalDay(), d2.getOrdinalDay()); return interval.isIn(getOrdinalDay(), left, right);

}

That brings us to the end of DayDate. So now we’ll make one more pass over the whole class to see how well it ﬂows.  
Điều đó đưa chúng ta đến cuối DayDate. Vì vậy, bây giờ chúng ta sẽ thực hiện một lần nữa cho cả lớp xem nó trôi chảy như thế nào.

First, the opening comment is long out of date, so I shortened and improved it [C2]. Next, I moved all the remaining enums out into their own ﬁles [G12].  
Đầu tiên, nhận xét mở đầu đã lỗi thời từ lâu, vì vậy tôi đã rút ngắn và cải thiện nó [C2]. Tiếp theo, tôi di chuyển tất cả các enum còn lại vào các tệp riêng của chúng [G12].

Next, I moved the static variable (dateFormatSymbols) and three static methods (getMonthNames*,* isLeapYear*,* lastDayOfMonth) into a new class named DateUtil [G6].  
Tiếp theo, tôi chuyển biến tĩnh (dateFormatSymbols) và ba phương thức tĩnh (getMonthNames, isLeapYear, lastDayOfMonth) vào một lớp mới có tên là DateUtil [G6].

I moved the abstract methods up to the top where they belong [G24].  
Tôi đã di chuyển các phương thức trừu tượng lên trên cùng nơi chúng thuộc về [G24].

I changed Month.make to Month.fromInt[N1] and did the same for all the other enums. I also created a toInt() accessor for all the enums and made the index ﬁeld private.  
Tôi đã thay đổi Month.make thành Month.fromInt[N1] và làm tương tự cho tất cả các enum khác. Tôi cũng đã tạo một trình truy cập toInt() cho tất cả các enum và đặt trường chỉ mục ở chế độ riêng tư.

There was some interesting duplication [G5] in plusYearsand plusMonthsthat I was able to eliminate by extracting a new method named correctLastDayOfMonth, making the all three methods much clearer.  
Có một số trùng lặp thú vị [G5] trong plusYearsand plusMonths mà tôi có thể loại bỏ bằng cách trích xuất một phương thức mới có tên là trueLastDayOfMonth, làm cho cả ba phương thức trở nên rõ ràng hơn nhiều.

I got rid of the magic number 1 [G25], replacing it with Month.JANUARY.toInt()or Day.SUNDAY.toInt(), as appropriate. I spent a little time with SpreadsheetDate, cleaning up the algorithms a bit. The end result is contained in Listing B-7, page 394, through Listing B-16, page 405.  
Tôi đã loại bỏ số ma thuật 1 [G25], thay thế nó bằng Month.JANUARY.toInt() hoặc Day.SUNDAY.toInt(), nếu phù hợp. Tôi đã dành một ít thời gian với SpreadsheetDate, dọn dẹp các thuật toán một chút. Kết quả cuối cùng có trong Liệt kê B-7, trang 394, đến Liệt kê B-16, trang 405.

<https://www.facebook.com/groups/phattrienlaptrinh>

284 **Chapter 16: Refactoring SerialDate**  
284 Chương 16: Tái cấu trúc SerialDate

Interestingly the code coverage in DayDatehas *decreased* to 84.9 percent! This is not because less functionality is being tested; rather it is because the class has shrunk so much that the few uncovered lines have a greater weight. DayDatenow has 45 out of 53 execut-able statements covered by tests. The uncovered lines are so trivial that they weren’t worth testing.  
Điều thú vị là mức độ bao phủ mã trong DayDatehas đã giảm xuống còn 84,9 phần trăm! Điều này không phải vì ít chức năng hơn đang được thử nghiệm; đúng hơn là do lớp đã bị thu hẹp lại quá nhiều nên một số dòng không được che chắn có trọng lượng lớn hơn. DayDatenow có 45 trong số 53 câu lệnh có thể thực thi được kiểm tra. Các dòng không được khám phá quá tầm thường đến mức chúng không đáng để thử nghiệm.

[**Conclusion**](#_page_533_0)  
Phần kết luận

So once again we’ve followed the Boy Scout Rule. We’ve checked the code in a bit cleaner than when we checked it out. It took a little time, but it was worth it. Test coverage was increased, some bugs were ﬁxed, the code was clariﬁed and shrunk. The next person to look at this code will hopefully ﬁnd it easier to deal with than we did. That person will also probably be able to clean it up a bit more than we did.  
Vì vậy, một lần nữa chúng tôi đã tuân theo Quy tắc Hướng đạo sinh. Chúng tôi đã kiểm tra mã rõ ràng hơn một chút so với khi chúng tôi kiểm tra. Phải mất một ít thời gian, nhưng nó đáng giá. Phạm vi kiểm thử đã tăng lên, một số lỗi đã được sửa, mã được làm rõ và thu gọn lại. Người tiếp theo xem đoạn mã này hy vọng sẽ thấy nó dễ xử lý hơn chúng tôi. Người đó cũng có thể sẽ dọn dẹp nó nhiều hơn chúng ta một chút.

[**Bibliography**](#_page_533_0)  
Thư mục

**[GOF]:** *Design Patterns: Elements of Reusable Object Oriented Software*, Gamma et al., Addison-Wesley, 1996.

**[Simmons04]:** *Hardcore Java*, Robert Simmons, Jr., O’Reilly, 2004.

**[Refactoring]:** *Refactoring: Improving the Design of Existing Code*, Martin Fowler et al., Addison-Wesley, 1999.

**[Beck97]:** *Smalltalk Best Practice Patterns*, Kent Beck, Prentice Hall, 1997.

<https://www.facebook.com/groups/phattrienlaptrinh>

[**17**](#_page_533_0)

[**Smells and Heuristics**](#_page_533_0)  
Mùi và Heuristic

In his wonderful book *Refactoring*,1 Martin Fowler identiﬁed many different “Code Smells.” The list that follows includes many of Martin’s smells and adds many more of my own. It also includes other pearls and heuristics that I use to practice my trade.  
Trong cuốn sách tuyệt vời Tái cấu trúc của mình, Martin Fowler đã xác định nhiều “Mùi mật mã” khác nhau. Danh sách sau đây bao gồm nhiều mùi của Martin và thêm nhiều mùi khác của riêng tôi. Nó cũng bao gồm các kinh nghiệm và phỏng đoán khác mà tôi sử dụng để thực hành giao dịch của mình.

1. [Refactoring].  
1. [Tái cấu trúc].

285

<https://www.facebook.com/groups/phattrienlaptrinh>

286 **Chapter 17: Smells and Heuristics**  
286 Chương 17: Mùi và Heuristic

I compiled this list by walking through several different programs and refactoring them. As I made each change, I asked myself *why* I made that change and then wrote the reason down here. The result is a rather long list of things that smell bad to me when I read code.  
Tôi biên soạn danh sách này bằng cách xem qua một số chương trình khác nhau và cấu trúc lại chúng. Khi tôi thực hiện mỗi thay đổi, tôi tự hỏi tại sao tôi lại thực hiện thay đổi đó và sau đó viết lý do ra đây. Kết quả là một danh sách khá dài những thứ khiến tôi khó chịu khi đọc mã.

This list is meant to be read from top to bottom and also to be used as a reference. There is a cross-reference for each heuristic that shows you where it is referenced in the rest of the text in “Appendix C” on page 409.  
Danh sách này được dùng để đọc từ trên xuống dưới và cũng được dùng làm tài liệu tham khảo. Có một tham chiếu chéo cho từng kinh nghiệm cho bạn biết nó được tham chiếu ở đâu trong phần còn lại của văn bản trong “Phụ lục C” trên trang 409.

[**Comments**](#_page_533_0)  
Bình luận

[**C1: *Inappropriate Information***](#_page_533_0)  
C1: Thông tin không phù hợp

It is inappropriate for a comment to hold information better held in a different kind of sys-tem such as your source code control system, your issue tracking system, or any other record-keeping system. Change histories, for example, just clutter up source ﬁles with volumes of historical and uninteresting text. In general, meta-data such as authors, last-modiﬁed-date, SPR number, and so on should not appear in comments. Comments should be reserved for technical notes about the code and design.  
Sẽ không phù hợp nếu một nhận xét chứa thông tin được lưu trữ tốt hơn trong một loại hệ thống khác, chẳng hạn như hệ thống kiểm soát mã nguồn, hệ thống theo dõi vấn đề của bạn hoặc bất kỳ hệ thống lưu giữ hồ sơ nào khác. Ví dụ, lịch sử thay đổi chỉ làm lộn xộn các tệp nguồn với khối lượng văn bản lịch sử và không thú vị. Nói chung, siêu dữ liệu chẳng hạn như tác giả, ngày sửa đổi lần cuối, số SPR, v.v. không nên xuất hiện trong phần bình luận. Nhận xét nên được dành riêng cho ghi chú kỹ thuật về mã và thiết kế.

[**C2: *Obsolete Comment***](#_page_533_0)  
C2: Nhận xét lỗi thời

A comment that has gotten old, irrelevant, and incorrect is obsolete. Comments get old quickly. It is best not to write a comment that will become obsolete. If you ﬁnd an obsolete comment, it is best to update it or get rid of it as quickly as possible. Obsolete comments tend to migrate away from the code they once described. They become ﬂoating islands of irrelevance and misdirection in the code.  
Một nhận xét đã cũ, không liên quan và không chính xác là lỗi thời. Nhận xét cũ đi một cách nhanh chóng. Tốt nhất là không nên viết bình luận sẽ trở nên lỗi thời. Nếu bạn tìm thấy một bình luận lỗi thời, tốt nhất là cập nhật nó hoặc loại bỏ nó càng nhanh càng tốt. Các bình luận lỗi thời có xu hướng di chuyển khỏi mã mà chúng từng mô tả. Chúng trở thành những hòn đảo trôi nổi không liên quan và sai hướng trong mã.

[**C3: *Redundant Comment***](#_page_533_0)  
C3: Nhận xét thừa

A comment is redundant if it describes something that adequately describes itself. For example:  
Một bình luận là dư thừa nếu nó mô tả một cái gì đó mô tả đầy đủ chính nó. Ví dụ:

i++; // increment i

Another example is a Javadoc that says nothing more than (or even less than) the function signature:  
Một ví dụ khác là một Javadoc không nói gì hơn (hoặc thậm chí ít hơn) chữ ký của hàm:

/\*\*

\* @param sellRequest \* @return

\* @throws ManagedComponentException \*/

public SellResponse beginSellItem(SellRequest sellRequest) throws ManagedComponentException  
công khai SellResponse startSellItem(SellRequest sellRequest) ném ManagedComponentException

Comments should say things that the code cannot say for itself.  
Nhận xét nên nói những điều mà mã không thể nói cho chính nó.

<https://www.facebook.com/groups/phattrienlaptrinh>

**Environment** 287  
Môi trường 287

[**C4: *Poorly Written Comment***](#_page_533_0)  
C4: Nhận xét viết kém

A comment worth writing is worth writing well. If you are going to write a comment, take the time to make sure it is the best comment you can write. Choose your words carefully. Use correct grammar and punctuation. Don’t ramble. Don’t state the obvious. Be brief.  
Một nhận xét đáng viết là đáng để viết tốt. Nếu bạn định viết bình luận, hãy dành thời gian để đảm bảo rằng đó là bình luận tốt nhất mà bạn có thể viết. Chọn từ của bạn một cách cẩn thận. Sử dụng đúng ngữ pháp và dấu câu. Đừng lan man. Đừng nêu rõ ràng. Hãy ngắn gọn.

[**C5: *Commented-Out Code***](#_page_533_0)  
C5: Mã nhận xét

It makes me crazy to see stretches of code that are commented out. Who knows how old it is? Who knows whether or not it’s meaningful? Yet no one will delete it because everyone assumes someone else needs it or has plans for it.  
Tôi phát điên khi thấy các đoạn mã được nhận xét. Ai biết nó bao nhiêu tuổi? Ai biết nó có ý nghĩa hay không? Tuy nhiên, sẽ không ai xóa nó đi vì mọi người đều cho rằng người khác cần nó hoặc có kế hoạch cho nó.

That code sits there and rots, getting less and less relevant with every passing day. It calls functions that no longer exist. It uses variables whose names have changed. It follows conventions that are long obsolete. It pollutes the modules that contain it and distracts the people who try to read it. Commented-out code is an *abomination*.  
Mã đó nằm đó và mục nát, ngày càng ít liên quan hơn qua từng ngày. Nó gọi các chức năng không còn tồn tại. Nó sử dụng các biến có tên đã thay đổi. Nó tuân theo các quy ước đã lỗi thời từ lâu. Nó làm ô nhiễm các mô-đun chứa nó và khiến những người cố gắng đọc nó mất tập trung. Mã nhận xét là một điều ghê tởm.

When you see commented-out code, *delete it!* Don’t worry, the source code control system still remembers it. If anyone really needs it, he or she can go back and check out a previous version. Don’t suffer commented-out code to survive.  
Khi bạn thấy mã nhận xét, hãy xóa nó! Đừng lo lắng, hệ thống kiểm soát mã nguồn vẫn ghi nhớ nó. Nếu ai thực sự cần nó, họ có thể quay lại và xem phiên bản trước. Đừng chịu đựng mã nhận xét để tồn tại.

[**Environment**](#_page_533_0)  
Môi trường

[**E1: *Build Requires More Than One Step***](#_page_533_0)  
E1: Xây dựng yêu cầu nhiều hơn một bước

Building a project should be a single trivial operation. You should not have to check many little pieces out from source code control. You should not need a sequence of arcane com-mands or context dependent scripts in order to build the individual elements. You should not have to search near and far for all the various little extra JARs, XML ﬁles, and other artifacts that the system requires. You *should* be able to check out the system with one sim-ple command and then issue one other simple command to build it.  
Xây dựng một dự án nên là một hoạt động tầm thường duy nhất. Bạn không cần phải kiểm tra nhiều phần nhỏ từ kiểm soát mã nguồn. Bạn không cần một chuỗi các lệnh phức tạp hoặc các tập lệnh phụ thuộc vào ngữ cảnh để xây dựng các phần tử riêng lẻ. Bạn không cần phải tìm kiếm gần xa tất cả các tệp JAR, tệp XML bổ sung nhỏ khác nhau và các tạo phẩm khác mà hệ thống yêu cầu. Bạn sẽ có thể kiểm tra hệ thống bằng một lệnh đơn giản và sau đó đưa ra một lệnh đơn giản khác để xây dựng nó.

svn get mySystem cd mySystem

ant all

[**E2:*Tests Require More Than One Step***](#_page_533_0)  
E2: Các bài kiểm tra yêu cầu nhiều hơn một bước

You should be able to run *all* the unit tests with just one command. In the best case you can run all the tests by clicking on one button in your IDE. In the worst case you should be able to issue a single simple command in a shell. Being able to run all the tests is so fundamental and so important that it should be quick, easy, and obvious to do.  
Bạn sẽ có thể chạy tất cả các bài kiểm tra đơn vị chỉ bằng một lệnh. Trong trường hợp tốt nhất, bạn có thể chạy tất cả các bài kiểm tra bằng cách nhấp vào một nút trong IDE của mình. Trong trường hợp xấu nhất, bạn có thể đưa ra một lệnh đơn giản trong trình bao. Việc có thể chạy tất cả các bài kiểm tra là rất cơ bản và quan trọng đến mức nó phải được thực hiện nhanh chóng, dễ dàng và rõ ràng.

<https://www.facebook.com/groups/phattrienlaptrinh>

288 **Chapter 17: Smells and Heuristics**  
288 Chương 17: Mùi và Heuristic

[**Functions**](#_page_533_0)  
Chức năng

[**F1:*Too Many Arguments***](#_page_533_0)  
F1: Quá Nhiều Đối Số

Functions should have a small number of arguments. No argument is best, followed by one, two, and three. More than three is very questionable and should be avoided with prej-udice. (See “Function Arguments” on page 40.)  
Các chức năng nên có một số lượng nhỏ các đối số. Không có đối số nào là tốt nhất, tiếp theo là một, hai và ba. Nhiều hơn ba là rất đáng ngờ và nên tránh dùng prej-udice. (Xem “Đối số hàm” trên trang 40.)

[**F2: *Output Arguments***](#_page_533_0)  
F2: Đối số đầu ra

Output arguments are counterintuitive. Readers expect arguments to be inputs, not out-puts. If your function must change the state of something, have it change the state of the object it is called on. (See “Output Arguments” on page 45.)  
Đối số đầu ra là phản trực giác. Người đọc mong đợi các đối số là đầu vào chứ không phải đầu ra. Nếu chức năng của bạn phải thay đổi trạng thái của một cái gì đó, hãy để nó thay đổi trạng thái của đối tượng mà nó được gọi. (Xem “Đối số đầu ra” trên trang 45.)

[**F3: *Flag Arguments***](#_page_533_0)  
F3: Đối số cờ

Boolean arguments loudly declare that the function does more than one thing. They are confusing and should be eliminated. (See “Flag Arguments” on page 41.)  
Đối số Boolean lớn tiếng tuyên bố rằng hàm thực hiện nhiều hơn một việc. Chúng gây nhầm lẫn và nên được loại bỏ. (Xem “Đối số gắn cờ” trên trang 41.)

[**F4: *Dead Function***](#_page_533_0)  
F4: Chức năng chết

Methods that are never called should be discarded. Keeping dead code around is wasteful. Don’t be afraid to delete the function. Remember, your source code control system still remembers it.  
Các phương thức không bao giờ được gọi nên bị loại bỏ. Giữ mã chết xung quanh là lãng phí. Đừng ngại xóa chức năng. Hãy nhớ rằng, hệ thống kiểm soát mã nguồn của bạn vẫn ghi nhớ nó.

[**General**](#_page_533_0)  
Tổng quan

[**G1: *Multiple Languages in One Source File***](#_page_533_0)  
G1: Nhiều ngôn ngữ trong một tệp nguồn

Today’s modern programming environments make it possible to put many different languages into a single source ﬁle. For example, a Java source ﬁle might contain snippets of XML, HTML, YAML, JavaDoc, English, JavaScript, and so on. For another example, in addition to HTML a JSP ﬁle might contain Java, a tag library syntax, English comments, Javadocs, XML, JavaScript, and so forth. This is confusing at best and carelessly sloppy at worst.  
Môi trường lập trình hiện đại ngày nay cho phép đặt nhiều ngôn ngữ khác nhau vào một tệp nguồn duy nhất. Ví dụ: tệp nguồn Java có thể chứa các đoạn mã XML, HTML, YAML, JavaDoc, tiếng Anh, JavaScript, v.v. Ví dụ khác, ngoài HTML, tệp JSP có thể chứa Java, cú pháp thư viện thẻ, nhận xét tiếng Anh, Javadocs, XML, JavaScript, v.v. Điều này tốt nhất là khó hiểu và tồi tệ nhất là cẩu thả bất cẩn.

The ideal is for a source ﬁle to contain one, and only one, language. Realistically, we will probably have to use more than one. But we should take pains to minimize both the number and extent of extra languages in our source ﬁles.  
Lý tưởng nhất là một tệp nguồn chứa một và chỉ một ngôn ngữ. Trên thực tế, chúng ta có thể sẽ phải sử dụng nhiều hơn một. Nhưng chúng ta nên cố gắng giảm thiểu cả số lượng và mức độ của các ngôn ngữ bổ sung trong các tệp nguồn của mình.

[**G2: *Obvious Behavior Is Unimplemented***](#_page_533_0)  
G2: Hành vi rõ ràng không được thực hiện

Following “The Principle of Least Surprise,”2 any function or class should implement the behaviors that another programmer could reasonably expect. For example, consider a function that translates the name of a day to an enum that represents the day.  
Tuân theo “Nguyên tắc ít ngạc nhiên nhất,”2 bất kỳ chức năng hoặc lớp nào cũng phải triển khai các hành vi mà một lập trình viên khác có thể mong đợi một cách hợp lý. Ví dụ: hãy xem xét một hàm dịch tên của một ngày thành một enum đại diện cho ngày đó.

2. Or “The Principle of Least Astonishment”: [http://en.wikipedia.org/wiki/](http://en.wikipedia.org/wiki/Principle_of_least_astonishment) [Principle\_of\_least\_astonishment](http://en.wikipedia.org/wiki/Principle_of_least_astonishment)  
2. Hoặc “Nguyên tắc ít ngạc nhiên nhất”: http://en.wikipedia.org/wiki/Principle\_of\_least\_astonishment

<https://www.facebook.com/groups/phattrienlaptrinh>

**General** 289  
Tổng hợp 289

Day day = DayDate.StringToDay(String dayName);

We would expect the string "Monday"to be translated to Day.MONDAY. We would also expect the common abbreviations to be translated, and we would expect the function to ignore case.  
Chúng tôi mong muốn chuỗi "Monday" được dịch thành Day.MONDAY. Chúng tôi cũng mong đợi các từ viết tắt phổ biến được dịch và chúng tôi mong đợi chức năng bỏ qua trường hợp.

When an obvious behavior is not implemented, readers and users of the code can no longer depend on their intuition about function names. They lose their trust in the original author and must fall back on reading the details of the code.  
Khi một hành vi rõ ràng không được thực hiện, người đọc và người dùng mã không còn có thể phụ thuộc vào trực giác của họ về tên hàm. Họ mất niềm tin vào tác giả gốc và phải quay lại đọc các chi tiết của mã.

[**G3: *Incorrect Behavior at the Boundaries***](#_page_533_0)  
G3: Hành vi không đúng tại ranh giới

It seems obvious to say that code should behave correctly. The problem is that we seldom realize just how complicated correct behavior is. Developers often write functions that they think will work, and then trust their intuition rather than going to the effort to prove that their code works in all the corner and boundary cases.  
Rõ ràng là mã phải hoạt động đúng. Vấn đề là chúng ta hiếm khi nhận ra hành vi đúng phức tạp như thế nào. Các nhà phát triển thường viết các chức năng mà họ nghĩ sẽ hoạt động, sau đó tin tưởng vào trực giác của họ hơn là cố gắng chứng minh rằng mã của họ hoạt động trong tất cả các trường hợp góc và ranh giới.

There is no replacement for due diligence. Every boundary condition, every corner case, every quirk and exception represents something that can confound an elegant and intuitive algorithm. *Don’t rely on your intuition*. Look for every boundary condition and write a test for it.  
Không có thay thế cho sự siêng năng. Mọi điều kiện biên, mọi trường hợp góc, mọi điều kỳ quặc và ngoại lệ đều đại diện cho một thứ gì đó có thể gây nhầm lẫn cho một thuật toán trực quan và tao nhã. Đừng dựa vào trực giác của bạn. Hãy tìm mọi điều kiện biên và viết bài kiểm tra cho nó.

[**G4: *Overridden Safeties***](#_page_533_0)  
G4: Két an toàn bị ghi đè

Chernobyl melted down because the plant manager overrode each of the safety mecha-nisms one by one. The safeties were making it inconvenient to run an experiment. The result was that the experiment did not get run, and the world saw it’s ﬁrst major civilian nuclear catastrophe.  
Chernobyl tan chảy vì người quản lý nhà máy đã vượt qua từng cơ chế an toàn một. Những chiếc két an toàn khiến việc chạy thử nghiệm trở nên bất tiện. Kết quả là cuộc thử nghiệm đã không được tiến hành và thế giới chứng kiến đây là thảm họa hạt nhân dân sự lớn đầu tiên.

It is risky to override safeties. Exerting manual control over serialVersionUIDmay be necessary, but it is always risky. Turning off certain compiler warnings (or all warnings!) may help you get the build to succeed, but at the risk of endless debugging sessions. Turn-ing off failing tests and telling yourself you’ll get them to pass later is as bad as pretending your credit cards are free money.  
Nó là rủi ro để ghi đè lên két sắt. Việc sử dụng quyền kiểm soát thủ công đối với serialVersionUID có thể là cần thiết, nhưng nó luôn tiềm ẩn rủi ro. Tắt một số cảnh báo của trình biên dịch (hoặc tất cả các cảnh báo!) Có thể giúp bạn xây dựng thành công, nhưng có nguy cơ dẫn đến các phiên gỡ lỗi vô tận. Bỏ qua các bài kiểm tra trượt và tự nhủ rằng sau này bạn sẽ vượt qua chúng cũng tệ như việc giả vờ rằng thẻ tín dụng của bạn là tiền miễn phí.

[**G5: *Duplication***](#_page_533_0)  
G5: Sao chép

This is one of the most important rules in this book, and you should take it very seriously. Virtually every author who writes about software design mentions this rule. Dave Thomas and Andy Hunt called it the DRY3 principle (Don’t Repeat Yourself). Kent Beck made it one of the core principles of Extreme Programming and called it: “Once, and only once.” Ron Jeffries ranks this rule second, just below getting all the tests to pass.  
Đây là một trong những quy tắc quan trọng nhất trong cuốn sách này, và bạn nên thực hiện nó một cách nghiêm túc. Hầu như mọi tác giả viết về thiết kế phần mềm đều đề cập đến quy tắc này. Dave Thomas và Andy Hunt gọi đó là nguyên tắc DRY3 (Đừng lặp lại chính mình). Kent Beck đã biến nó thành một trong những nguyên tắc cốt lõi của Lập trình cực đoan và gọi nó là: “Một lần và chỉ một lần”. Ron Jeffries xếp quy tắc này ở vị trí thứ hai, ngay sau việc vượt qua tất cả các bài kiểm tra.

Every time you see duplication in the code, it represents a missed opportunity for abstraction. That duplication could probably become a subroutine or perhaps another class outright. By folding the duplication into such an abstraction, you increase the vocab-ulary of the language of your design. Other programmers can use the abstract facilities  
Mỗi khi bạn thấy sự trùng lặp trong mã, nó thể hiện một cơ hội trừu tượng bị bỏ lỡ. Sự sao chép đó có thể trở thành một chương trình con hoặc có lẽ là một lớp hoàn toàn khác. Bằng cách gấp bản sao thành một sự trừu tượng như vậy, bạn tăng vốn từ vựng cho ngôn ngữ thiết kế của mình. Các lập trình viên khác có thể sử dụng các cơ sở trừu tượng

3. [PRAG].  
3. [PRAG].

<https://www.facebook.com/groups/phattrienlaptrinh>

290 **Chapter 17: Smells and Heuristics**  
290 Chương 17: Mùi và Heuristic

you create. Coding becomes faster and less error prone because you have raised the abstraction level.  
bạn tạo. Mã hóa trở nên nhanh hơn và ít lỗi hơn vì bạn đã nâng cao mức độ trừu tượng.

The most obvious form of duplication is when you have clumps of identical code that look like some programmers went wild with the mouse, pasting the same code over and over again. These should be replaced with simple methods.  
Hình thức sao chép rõ ràng nhất là khi bạn có các đoạn mã giống hệt nhau trông giống như một số lập trình viên đã sử dụng chuột một cách điên cuồng, dán đi dán lại cùng một đoạn mã. Chúng nên được thay thế bằng các phương pháp đơn giản.

A more subtle form is the switch/caseor if/elsechain that appears again and again in various modules, always testing for the same set of conditions. These should be replaced with polymorphism.  
Một hình thức tinh vi hơn là switch/caseor if/elsechain xuất hiện lặp đi lặp lại trong các mô-đun khác nhau, luôn kiểm tra cùng một tập hợp các điều kiện. Chúng nên được thay thế bằng đa hình.

Still more subtle are the modules that have similar algorithms, but that don’t share similar lines of code. This is still duplication and should be addressed by using the TEM-PLATE METHOD,4 or STRATEGY5 pattern.  
Tinh vi hơn nữa là các mô-đun có thuật toán tương tự, nhưng không chia sẻ các dòng mã tương tự. Đây vẫn là sự trùng lặp và cần được giải quyết bằng cách sử dụng mẫu TEM-PLATE METHOD,4 hoặc STRATEGY5.

Indeed, most of the design patterns that have appeared in the last ﬁfteen years are sim-ply well-known ways to eliminate duplication. So too the Codd Normal Forms are a strat-egy for eliminating duplication in database schemae. OO itself is a strategy for organizing modules and eliminating duplication. Not surprisingly, so is structured programming.  
Thật vậy, hầu hết các mẫu thiết kế đã xuất hiện trong mười lăm năm qua đơn giản là những cách nổi tiếng để loại bỏ sự trùng lặp. Vì vậy, Codd Normal Forms cũng là một chiến lược để loại bỏ sự trùng lặp trong lược đồ cơ sở dữ liệu. Bản thân OO là một chiến lược để tổ chức các mô-đun và loại bỏ sự trùng lặp. Không ngạc nhiên, lập trình có cấu trúc cũng vậy.

I think the point has been made. Find and eliminate duplication wherever you can.  
Tôi nghĩ rằng điểm đã được thực hiện. Tìm và loại bỏ trùng lặp bất cứ nơi nào bạn có thể.

[**G6: *Code at Wrong Level of Abstraction***](#_page_533_0)  
G6: Mã ở mức độ trừu tượng sai

It is important to create abstractions that separate higher level general concepts from lower level detailed concepts. Sometimes we do this by creating abstract classes to hold the higher level concepts and derivatives to hold the lower level concepts. When we do this, we need to make sure that the separation is complete. We want *all* the lower level concepts to be in the derivatives and *all* the higher level concepts to be in the base class.  
Điều quan trọng là tạo ra sự trừu tượng tách biệt các khái niệm chung ở cấp độ cao hơn với các khái niệm chi tiết ở cấp độ thấp hơn. Đôi khi chúng ta làm điều này bằng cách tạo các lớp trừu tượng để chứa các khái niệm cấp cao hơn và các dẫn xuất để chứa các khái niệm cấp thấp hơn. Khi chúng tôi làm điều này, chúng tôi cần đảm bảo rằng quá trình phân tách đã hoàn tất. Chúng tôi muốn tất cả các khái niệm cấp thấp hơn nằm trong các dẫn xuất và tất cả các khái niệm cấp cao hơn nằm trong lớp cơ sở.

For example, constants, variables, or utility functions that pertain only to the detailed implementation should not be present in the base class. The base class should know noth-ing about them.  
Ví dụ, các hằng số, biến hoặc hàm tiện ích chỉ liên quan đến việc triển khai chi tiết sẽ không có mặt trong lớp cơ sở. Lớp cơ sở không nên biết gì về chúng.

This rule also pertains to source ﬁles, components, and modules. Good software design requires that we separate concepts at different levels and place them in different containers. Sometimes these containers are base classes or derivatives and sometimes they are source ﬁles, modules, or components. Whatever the case may be, the separation needs to be complete. We don’t want lower and higher level concepts mixed together.  
Quy tắc này cũng liên quan đến các tệp nguồn, thành phần và mô-đun. Thiết kế phần mềm tốt yêu cầu chúng ta phân tách các khái niệm ở các cấp độ khác nhau và đặt chúng vào các thùng chứa khác nhau. Đôi khi những vùng chứa này là các lớp cơ sở hoặc dẫn xuất và đôi khi chúng là các tệp nguồn, mô-đun hoặc thành phần. Dù trường hợp có thể là gì, sự tách biệt cần phải được hoàn thành. Chúng tôi không muốn các khái niệm cấp thấp hơn và cấp cao hơn trộn lẫn với nhau.

Consider the following code:  
Hãy xem xét đoạn mã sau:

public interface Stack {

Object pop() throws EmptyException;

void push(Object o) throws FullException; double percentFull();

4. [GOF]. 5. [GOF].

<https://www.facebook.com/groups/phattrienlaptrinh>

**General** 291  
Tổng hợp 291

class EmptyException extends Exception {} class FullException extends Exception {}

}

The percentFullfunction is at the wrong level of abstraction. Although there are many implementations of Stackwhere the concept of *fullness* is reasonable, there are other implementations that simply *could not know* how full they are. So the function would be better placed in a derivative interface such as BoundedStack.  
Phần trămFullfunction ở mức trừu tượng sai. Mặc dù có nhiều triển khai Stack trong đó khái niệm đầy đủ là hợp lý, nhưng có những triển khai khác đơn giản là không thể biết chúng đầy đến mức nào. Vì vậy, chức năng sẽ được đặt tốt hơn trong một giao diện phái sinh, chẳng hạn như BoundedStack.

Perhaps you are thinking that the implementation could just return zero if the stack were boundless. The problem with that is that no stack is truly boundless. You cannot really prevent an OutOfMemoryException by checking for  
Có lẽ bạn đang nghĩ rằng việc triển khai chỉ có thể trả về 0 nếu ngăn xếp là vô hạn. Vấn đề với điều đó là không có ngăn xếp nào thực sự vô hạn. Bạn thực sự không thể ngăn OutOfMemoryException bằng cách kiểm tra

stack.percentFull() < 50.0.

Implementing the function to return 0 would be telling a lie.  
Việc triển khai hàm trả về 0 sẽ là nói dối.

The point is that you cannot lie or fake your way out of a misplaced abstraction. Iso-lating abstractions is one of the hardest things that software developers do, and there is no quick ﬁx when you get it wrong.  
Vấn đề là bạn không thể nói dối hoặc giả tạo để thoát khỏi một sự trừu tượng không đúng chỗ. Sự trừu tượng hóa cô lập là một trong những điều khó khăn nhất mà các nhà phát triển phần mềm thực hiện và không có cách khắc phục nhanh chóng khi bạn hiểu sai.

[**G7: *Base Classes Depending on Their Derivatives***](#_page_533_0)  
G7: Các lớp cơ sở tùy thuộc vào các dẫn xuất của chúng

The most common reason for partitioning concepts into base and derivative classes is so that the higher level base class concepts can be independent of the lower level derivative class concepts. Therefore, when we see base classes mentioning the names of their deriva-tives, we suspect a problem. In general, base classes should know nothing about their derivatives.  
Lý do phổ biến nhất để phân vùng các khái niệm thành các lớp cơ sở và lớp phái sinh là để các khái niệm lớp cơ sở cấp cao hơn có thể độc lập với các khái niệm lớp phái sinh cấp thấp hơn. Do đó, khi chúng tôi thấy các lớp cơ sở đề cập đến tên của các dẫn xuất của chúng, chúng tôi nghi ngờ có vấn đề. Nói chung, các lớp cơ sở không nên biết gì về các dẫn xuất của chúng.

There are exceptions to this rule, of course. Sometimes the number of derivatives is strictly ﬁxed, and the base class has code that selects between the derivatives. We see this a lot in ﬁnite state machine implementations. However, in that case the derivatives and base class are strongly coupled and always deploy together in the same jar ﬁle. In the general case we want to be able to deploy derivatives and bases in different jar ﬁles.  
Tất nhiên, có những ngoại lệ đối với quy tắc này. Đôi khi số lượng các dẫn xuất được cố định chặt chẽ và lớp cơ sở có mã để chọn giữa các dẫn xuất. Chúng tôi thấy điều này rất nhiều trong việc triển khai máy trạng thái hữu hạn. Tuy nhiên, trong trường hợp đó, lớp dẫn xuất và lớp cơ sở được kết hợp chặt chẽ và luôn triển khai cùng nhau trong cùng một tệp jar. Trong trường hợp chung, chúng tôi muốn có thể triển khai các đạo hàm và cơ số trong các tệp jar khác nhau.

Deploying derivatives and bases in different jar ﬁles and making sure the base jar ﬁles know nothing about the contents of the derivative jar ﬁles allow us to deploy our systems in discrete and independent components. When such components are modiﬁed, they can be redeployed without having to redeploy the base components. This means that the impact of a change is greatly lessened, and maintaining systems in the ﬁeld is made much simpler.  
Việc triển khai các dẫn xuất và cơ sở trong các tệp jar khác nhau và đảm bảo rằng các tệp jar cơ sở không biết gì về nội dung của các tệp jar dẫn xuất cho phép chúng tôi triển khai các hệ thống của mình trong các thành phần riêng biệt và độc lập. Khi các thành phần như vậy được sửa đổi, chúng có thể được triển khai lại mà không cần phải triển khai lại các thành phần cơ sở. Điều này có nghĩa là tác động của một sự thay đổi sẽ giảm đi rất nhiều và việc duy trì các hệ thống tại hiện trường trở nên đơn giản hơn nhiều.

[**G8:*Too Much Information***](#_page_533_0)  
G8: Quá nhiều thông tin

Well-deﬁned modules have very small interfaces that allow you to do a lot with a little. Poorly deﬁned modules have wide and deep interfaces that force you to use many different gestures to get simple things done. A well-deﬁned interface does not offer very many func-tions to depend upon, so coupling is low. A poorly deﬁned interface provides lots of func-tions that you must call, so coupling is high.  
Các mô-đun được xác định rõ ràng có các giao diện rất nhỏ cho phép bạn làm được nhiều việc chỉ với một chút. Các mô-đun được xác định kém có giao diện rộng và sâu buộc bạn phải sử dụng nhiều cử chỉ khác nhau để hoàn thành những việc đơn giản. Một giao diện được xác định rõ ràng không cung cấp nhiều chức năng phụ thuộc vào, do đó khả năng kết nối thấp. Một giao diện được xác định kém cung cấp rất nhiều chức năng mà bạn phải gọi, do đó khả năng ghép nối cao.

<https://www.facebook.com/groups/phattrienlaptrinh>

292 **Chapter 17: Smells and Heuristics**  
292 Chương 17: Mùi và Heuristic

Good software developers learn to limit what they expose at the interfaces of their classes and modules. The fewer methods a class has, the better. The fewer variables a func-tion knows about, the better. The fewer instance variables a class has, the better.  
Các nhà phát triển phần mềm giỏi học cách hạn chế những gì họ thể hiện ở giao diện của các lớp và mô-đun của họ. Một lớp có càng ít phương thức thì càng tốt. Hàm biết càng ít biến thì càng tốt. Một lớp càng có ít biến đối tượng thì càng tốt.

Hide your data. Hide your utility functions. Hide your constants and your temporaries. Don’t create classes with lots of methods or lots of instance variables. Don’t create lots of protected variables and functions for your subclasses. Concentrate on keeping interfaces very tight and very small. Help keep coupling low by limiting information.  
Ẩn dữ liệu của bạn. Ẩn các chức năng tiện ích của bạn. Ẩn các hằng số và tạm thời của bạn. Đừng tạo các lớp có nhiều phương thức hoặc nhiều biến thể hiện. Đừng tạo nhiều biến và hàm được bảo vệ cho các lớp con của bạn. Tập trung vào việc giữ cho các giao diện rất chặt chẽ và rất nhỏ. Giúp giữ cho khớp nối thấp bằng cách hạn chế thông tin.

[**G9: *Dead Code***](#_page_533_0)  
G9: Mã chết

Dead code is code that isn’t executed. You ﬁnd it in the body of an ifstatement that checks for a condition that can’t happen. You ﬁnd it in the catchblock of a trythat never throws. You ﬁnd it in little utility methods that are never called or switch/caseconditions that never occur.  
Mã chết là mã không được thực thi. Bạn tìm thấy nó trong phần thân của câu lệnh if để kiểm tra một điều kiện không thể xảy ra. Bạn tìm thấy nó trong khối bắt của một lần thử không bao giờ ném. Bạn tìm thấy nó trong các phương thức tiện ích nhỏ không bao giờ được gọi hoặc chuyển đổi/tình huống điều kiện không bao giờ xảy ra.

The problem with dead code is that after awhile it starts to smell. The older it is, the stronger and sourer the odor becomes. This is because dead code is not completely updated when designs change. It still *compiles*, but it does not follow newer conventions or rules. It was written at a time when the system was *different*. When you ﬁnd dead code, do the right thing. Give it a decent burial. Delete it from the system.  
Vấn đề với mã chết là sau một thời gian, nó bắt đầu có mùi. Càng để lâu mùi càng nồng và chua. Điều này là do mã chết không được cập nhật hoàn toàn khi thiết kế thay đổi. Nó vẫn biên dịch, nhưng nó không tuân theo các quy ước hoặc quy tắc mới hơn. Nó được viết vào thời điểm mà hệ thống đã khác. Khi bạn tìm thấy mã chết, hãy làm điều đúng đắn. Hãy chôn cất tử tế. Xóa nó khỏi hệ thống.

[**G10:*Vertical Separation***](#_page_533_0)  
G10:Tách dọc

Variables and function should be deﬁned close to where they are used. Local variables should be declared just above their ﬁrst usage and should have a small vertical scope. We don’t want local variables declared hundreds of lines distant from their usages.  
Các biến và chức năng nên được xác định gần với nơi chúng được sử dụng. Các biến cục bộ nên được khai báo ngay trên lần sử dụng đầu tiên của chúng và nên có phạm vi dọc nhỏ. Chúng tôi không muốn các biến cục bộ được khai báo cách xa hàng trăm dòng so với cách sử dụng của chúng.

Private functions should be deﬁned just below their ﬁrst usage. Private functions belong to the scope of the whole class, but we’d still like to limit the vertical distance between the invocations and deﬁnitions. Finding a private function should just be a matter of scanning downward from the ﬁrst usage.  
Các chức năng riêng nên được xác định ngay bên dưới lần sử dụng đầu tiên của chúng. Các chức năng riêng tư thuộc phạm vi của cả lớp, nhưng chúng tôi vẫn muốn giới hạn khoảng cách theo chiều dọc giữa các lời gọi và định nghĩa. Việc tìm kiếm một chức năng riêng chỉ là vấn đề quét xuống từ lần sử dụng đầu tiên.

[**G11: *Inconsistency***](#_page_533_0)  
G11: Không nhất quán

If you do something a certain way, do all similar things in the same way. This goes back to the principle of least surprise. Be careful with the conventions you choose, and once chosen, be careful to continue to follow them.  
Nếu bạn làm điều gì đó theo một cách nhất định, hãy làm tất cả những điều tương tự theo cùng một cách. Điều này quay trở lại nguyên tắc ít bất ngờ nhất. Hãy cẩn thận với các quy ước bạn chọn, và một khi đã chọn, hãy cẩn thận tiếp tục tuân theo chúng.

If within a particular function you use a variable named response to hold an HttpServletResponse, then use the same variable name consistently in the other functions that use HttpServletResponseobjects. If you name a method processVerificationRequest, then use a similar name, such as processDeletionRequest, for the methods that process other kinds of requests.  
Nếu trong một hàm cụ thể, bạn sử dụng một biến có tên là phản hồi để giữ một HttpServletResponse, thì hãy sử dụng cùng một tên biến một cách nhất quán trong các hàm khác sử dụng các đối tượng HttpServletResponse. Nếu bạn đặt tên cho một phương thức là processVerificationRequest, thì hãy sử dụng một tên tương tự, chẳng hạn như processDeletionRequest, cho các phương thức xử lý các loại yêu cầu khác.

Simple consistency like this, when reliably applied, can make code much easier to read and modify.  
Tính nhất quán đơn giản như thế này, khi được áp dụng một cách đáng tin cậy, có thể làm cho mã dễ đọc và sửa đổi hơn nhiều.

<https://www.facebook.com/groups/phattrienlaptrinh>

**General** 293  
Tổng hợp 293

[**G12: *Clutter***](#_page_533_0)  
G12: Lộn xộn

Of what use is a default constructor with no implementation? All it serves to do is clutter up the code with meaningless artifacts. Variables that aren’t used, functions that are never called, comments that add no information, and so forth. All these things are clutter and should be removed. Keep your source ﬁles clean, well organized, and free of clutter.  
Việc sử dụng một hàm tạo mặc định không có triển khai là gì? Tất cả những gì nó làm là làm lộn xộn mã với các đồ tạo tác vô nghĩa. Các biến không được sử dụng, các hàm không bao giờ được gọi, các chú thích không thêm thông tin, v.v. Tất cả những thứ này là lộn xộn và nên được loại bỏ. Giữ cho các tệp nguồn của bạn sạch sẽ, được tổ chức tốt và không lộn xộn.

[**G13: *Artiﬁcial Coupling***](#_page_609_0)  
G13: Khớp nối nhân tạo

Things that don’t depend upon each other should not be artiﬁcially coupled. For example, general enumsshould not be contained within more speciﬁc classes because this forces the whole application to know about these more speciﬁc classes. The same goes for general purpose static functions being declared in speciﬁc classes.  
Những thứ không phụ thuộc vào nhau không nên được kết hợp một cách giả tạo. Ví dụ, không nên chứa các enum chung trong các lớp cụ thể hơn vì điều này buộc toàn bộ ứng dụng phải biết về các lớp cụ thể hơn này. Điều tương tự cũng xảy ra với các hàm tĩnh có mục đích chung được khai báo trong các lớp cụ thể.

In general an artiﬁcial coupling is a coupling between two modules that serves no direct purpose. It is a result of putting a variable, constant, or function in a temporarily convenient, though inappropriate, location. This is lazy and careless.  
Nói chung, khớp nối nhân tạo là khớp nối giữa hai mô-đun không phục vụ mục đích trực tiếp. Đó là kết quả của việc đặt một biến, hằng hoặc hàm ở một vị trí tạm thời thuận tiện, mặc dù không phù hợp. Đây là lười biếng và bất cẩn.

Take the time to ﬁgure out where functions, constants, and variables ought to be declared. Don’t just toss them in the most convenient place at hand and then leave them there.  
Dành thời gian để tìm ra nơi các hàm, hằng và biến nên được khai báo. Đừng chỉ ném chúng vào nơi thuận tiện nhất trong tầm tay rồi để chúng ở đó.

[**G14: *Feature Envy***](#_page_609_0)  
G14: Ghen tị với tính năng

This is one of Martin Fowler’s code smells.6 The methods of a class should be interested in the variables and functions of the class they belong to, and not the variables and functions of other classes. When a method uses accessors and mutators of some other object to manipulate the data within that object, then it *envies* the scope of the class of that other object. It wishes that it were inside that other class so that it could have direct access to the variables it is manipulating. For example:  
Đây là một trong những mã có mùi của Martin Fowler.6 Các phương thức của một lớp nên quan tâm đến các biến và hàm của lớp mà chúng thuộc về, chứ không phải các biến và hàm của các lớp khác. Khi một phương thức sử dụng các bộ truy cập và bộ biến đổi của một số đối tượng khác để thao tác dữ liệu trong đối tượng đó, thì nó sẽ ghen tị với phạm vi của lớp của đối tượng khác đó. Nó ước rằng nó ở bên trong lớp khác đó để có thể truy cập trực tiếp vào các biến mà nó đang thao tác. Ví dụ:

public class HourlyPayCalculator {

public Money calculateWeeklyPay(HourlyEmployee e) { int tenthRate = e.getTenthRate().getPennies(); int tenthsWorked = e.getTenthsWorked();

int straightTime = Math.min(400, tenthsWorked);

int overTime = Math.max(0, tenthsWorked - straightTime); int straightPay = straightTime \* tenthRate;

int overtimePay = (int)Math.round(overTime\*tenthRate\*1.5); return new Money(straightPay + overtimePay);

} }

The calculateWeeklyPaymethod reaches into the HourlyEmployeeobject to get the data on which it operates. The calculateWeeklyPaymethod *envies* the scope of HourlyEmployee. It “wishes” that it could be inside HourlyEmployee.  
Phương thức tính toánWeeklyPay tiếp cận đối tượng HourlyEmployee để lấy dữ liệu mà nó hoạt động. Phương thức tính toánWeeklyPay phù hợp với phạm vi của HourlyEmployee. Nó “ước gì” rằng nó có thể ở bên trong HourlyEmployee.

6. [Refactoring].  
6. [Tái cấu trúc].

<https://www.facebook.com/groups/phattrienlaptrinh>

294 **Chapter 17: Smells and Heuristics**  
294 Chương 17: Mùi và Heuristic

All else being equal, we want to eliminate Feature Envy because it exposes the internals of one class to another. Sometimes, however, Feature Envy is a necessary evil. Consider the following:  
Tất cả những thứ khác đều bình đẳng, chúng tôi muốn loại bỏ Feature Envy vì nó phơi bày phần bên trong của lớp này sang lớp khác. Tuy nhiên, đôi khi, Feature Envy là một điều ác cần thiết. Hãy xem xét những điều sau đây:

public class HourlyEmployeeReport { private HourlyEmployee employee ;

public HourlyEmployeeReport(HourlyEmployee e) { this.employee = e;

}

String reportHours() { return String.format(

"Name: %s\tHours:%d.%1d\n", employee.getName(), employee.getTenthsWorked()/10, employee.getTenthsWorked()%10);  
"Name: %s\tHours:%d.%1d\n", employee.getName(), employee.getTenthsWorked()/10, employee.getTenthsWorked()%10);

} }

Clearly, the reportHoursmethod envies the HourlyEmployeeclass. On the other hand, we don’t want HourlyEmployeeto have to know about the format of the report. Moving that for-mat string into the HourlyEmployee class would violate several principles of object oriented design.7 It would couple HourlyEmployee to the format of the report, exposing it to changes in that format.  
Rõ ràng, phương thức reportHours ghen tị với lớp HourlyEmployee. Mặt khác, chúng tôi không muốn HourlyEmployee phải biết về định dạng của báo cáo. Di chuyển chuỗi for-mat đó vào lớp HourlyEmployee sẽ vi phạm một số nguyên tắc của thiết kế hướng đối tượng.7 Nó sẽ kết hợp HourlyEmployee với định dạng của báo cáo, khiến báo cáo phải chịu những thay đổi ở định dạng đó.

[**G15: *Selector Arguments***](#_page_609_0)  
G15: Đối số bộ chọn

There is hardly anything more abominable than a dangling falseargument at the end of a function call. What does it mean? What would it change if it were true? Not only is the purpose of a selector argument difﬁcult to remember, each selector argument combines many functions into one. Selector arguments are just a lazy way to avoid splitting a large function into several smaller functions. Consider:  
Hầu như không có điều gì ghê tởm hơn một đối số giả lủng lẳng ở cuối lệnh gọi hàm. Nó có nghĩa là gì? Điều gì sẽ thay đổi nếu nó là sự thật? Mục đích của đối số bộ chọn không chỉ khó nhớ, mà mỗi đối số bộ chọn còn kết hợp nhiều hàm thành một. Các đối số của bộ chọn chỉ là một cách lười biếng để tránh chia một hàm lớn thành nhiều hàm nhỏ hơn. Coi như:

public int calculateWeeklyPay(boolean overtime) { int tenthRate = getTenthRate();

int tenthsWorked = getTenthsWorked();

int straightTime = Math.min(400, tenthsWorked);

int overTime = Math.max(0, tenthsWorked - straightTime); int straightPay = straightTime \* tenthRate;

double overtimeRate = overtime ? 1.5 : 1.0 \* tenthRate; int overtimePay = (int)Math.round(overTime\*overtimeRate); return straightPay + overtimePay;  
tăng ca gấp đôi Tỷ lệ = làm thêm giờ ? 1,5 : 1,0 \* tỷ lệ phần mười; int làm thêm giờPay = (int)Math.round(làm thêm giờ\*Tỷ lệ làm thêm giờ); trả lương thẳng + trả lương ngoài giờ;

}

You call this function with a trueif overtime is paid as time and a half, and with a falseif overtime is paid as straight time. It’s bad enough that you must remember what calculateWeeklyPay(false)means whenever you happen to stumble across it. But the  
Bạn gọi hàm này với trueif làm thêm giờ được trả theo thời gian rưỡi và với falseif làm thêm giờ được trả theo thời gian thẳng. Thật tệ khi bạn phải nhớ tính toánWeeklyPay(false) nghĩa là gì bất cứ khi nào bạn tình cờ bắt gặp nó. Nhưng

7. Speciﬁcally, the Single Responsibility Principle, the Open Closed Principle, and the Common Closure Principle. See [PPP].  
7. Cụ thể là Nguyên tắc chịu trách nhiệm duy nhất, Nguyên tắc đóng mở và Nguyên tắc đóng chung. Xem [PPP].

<https://www.facebook.com/groups/phattrienlaptrinh>

**General** 295  
Tổng hợp 295

real shame of a function like this is that the author missed the opportunity to write the following:  
sự xấu hổ thực sự của một chức năng như thế này là tác giả đã bỏ lỡ cơ hội để viết như sau:

public int straightPay() {

return getTenthsWorked() \* getTenthRate(); }

public int overTimePay() {

int overTimeTenths = Math.max(0, getTenthsWorked() - 400); int overTimePay = overTimeBonus(overTimeTenths);

return straightPay() + overTimePay; }

private int overTimeBonus(int overTimeTenths) {

double bonus = 0.5 \* getTenthRate() \* overTimeTenths; return (int) Math.round(bonus);

}

Of course, selectors need not be boolean. They can be enums, integers, or any other type of argument that is used to select the behavior of the function. In general it is better to have many functions than to pass some code into a function to select the behavior.  
Tất nhiên, bộ chọn không cần phải là boolean. Chúng có thể là enum, số nguyên hoặc bất kỳ loại đối số nào khác được sử dụng để chọn hành vi của hàm. Nói chung, tốt hơn là có nhiều chức năng hơn là chuyển một số mã vào một chức năng để chọn hành vi.

[**G16: *Obscured Intent***](#_page_609_0)  
G16: Ý định bị che khuất

We want code to be as expressive as possible. Run-on expressions, Hungarian notation, and magic numbers all obscure the author’s intent. For example, here is the overTimePay function as it might have appeared:  
Chúng tôi muốn mã càng biểu cảm càng tốt. Các biểu thức run-on, ký hiệu tiếng Hungary và các con số ma thuật đều che khuất ý định của tác giả. Ví dụ: đây là chức năng overTimePay như nó có thể đã xuất hiện:

public int m\_otCalc() { return iThsWkd \* iThsRte +

(int) Math.round(0.5 \* iThsRte \* Math.max(0, iThsWkd - 400)

); }

Small and dense as this might appear, it’s also virtually impenetrable. It is worth tak-ing the time to make the intent of our code visible to our readers.  
Nhỏ và dày đặc như thế này có thể xuất hiện, nó cũng hầu như không thể xuyên thủng. Thật đáng để dành thời gian để hiển thị mục đích của mã của chúng tôi cho độc giả của chúng tôi.

[**G17: *Misplaced Responsibility***](#_page_609_0)  
G17: Đặt nhầm trách nhiệm

One of the most important decisions a software developer can make is where to put code. For example, where should the PIconstant go? Should it be in the Mathclass? Perhaps it belongs in the Trigonometry class? Or maybe in the Circle class?  
Một trong những quyết định quan trọng nhất mà nhà phát triển phần mềm có thể đưa ra là đặt mã ở đâu. Ví dụ, PIconstant nên đi đâu? Nó có nên ở trong Mathclass không? Có lẽ nó thuộc về lớp Lượng giác? Hoặc có thể trong lớp Circle?

The principle of least surprise comes into play here. Code should be placed where a reader would naturally expect it to be. The PIconstant should go where the trig functions are declared. The OVERTIME\_RATE constant should be declared in the HourlyPay-Calculator class.  
Nguyên tắc ít bất ngờ nhất phát huy tác dụng ở đây. Mã nên được đặt ở nơi mà người đọc mong đợi một cách tự nhiên. PIconstant sẽ đi đến nơi khai báo các hàm kích hoạt. Hằng số OVERTIME\_RATE phải được khai báo trong lớp Máy tính trả lương hàng giờ.

Sometimes we get “clever” about where to put certain functionality. We’ll put it in a function that’s convenient for us, but not necessarily intuitive to the reader. For example, perhaps we need to print a report with the total of hours that an employee worked. We  
Đôi khi chúng tôi nhận được "thông minh" về nơi đặt chức năng nhất định. Chúng tôi sẽ đặt nó vào một chức năng thuận tiện cho chúng tôi, nhưng không nhất thiết phải trực quan với người đọc. Ví dụ, có lẽ chúng ta cần in một báo cáo với tổng số giờ mà một nhân viên đã làm việc. Chúng tôi

<https://www.facebook.com/groups/phattrienlaptrinh>

296 **Chapter 17: Smells and Heuristics**  
296 Chương 17: Mùi và Heuristic

could sum up those hours in the code that prints the report, or we could try to keep a run-ning total in the code that accepts time cards.  
có thể tổng hợp số giờ đó trong mã in báo cáo hoặc chúng tôi có thể cố gắng giữ tổng số hiện tại trong mã chấp nhận thẻ chấm công.

One way to make this decision is to look at the names of the functions. Let’s say that our report module has a function named getTotalHours. Let’s also say that the module that accepts time cards has a saveTimeCardfunction. Which of these two functions, by it’s name, implies that it calculates the total? The answer should be obvious.  
Một cách để đưa ra quyết định này là xem tên của các chức năng. Giả sử mô-đun báo cáo của chúng ta có một hàm tên là getTotalHours. Cũng giả sử rằng mô-đun chấp nhận thẻ chấm công có chức năng saveTimeCard. Cái nào trong hai hàm này, theo tên của nó, ngụ ý rằng nó tính tổng? Câu trả lời nên được rõ ràng.

Clearly, there are sometimes performance reasons why the total should be calculated as time cards are accepted rather than when the report is printed. That’s ﬁne, but the names of the functions ought to reﬂect this. For example, there should be a computeRunning-TotalOfHours function in the timecard module.  
Rõ ràng, đôi khi có những lý do về hiệu suất khiến tổng số phải được tính khi thẻ chấm công được chấp nhận thay vì khi báo cáo được in. Điều đó tốt, nhưng tên của các chức năng phải phản ánh điều này. Ví dụ: nên có một hàm computeRunning-TotalOfHours trong mô-đun thẻ chấm công.

[**G18: *Inappropriate Static***](#_page_609_0)  
G18: Tĩnh không phù hợp

Math.max(double a,double b)is a good static method. It does not operate on a single instance; indeed, it would be silly to have to say newMath().max(a,b)or even a.max(b). All the data that maxuses comes from its two arguments, and not from any “owning” object. More to the point, there is almost *no chance* that we’d want Math.maxto be polymorphic.  
Math.max(double a,double b) là một phương pháp tĩnh tốt. Nó không hoạt động trên một phiên bản duy nhất; thực sự, sẽ thật ngớ ngẩn khi phải nói newMath().max(a,b)hoặc thậm chí a.max(b). Tất cả dữ liệu tối đa đến từ hai đối số của nó chứ không phải từ bất kỳ đối tượng "sở hữu" nào. Hơn nữa, gần như không có khả năng chúng ta muốn Math.max trở thành đa hình.

Sometimes, however, we write static functions that should not be static. For example, consider:  
Tuy nhiên, đôi khi chúng ta viết các hàm tĩnh không nên tĩnh. Ví dụ: xem xét:

HourlyPayCalculator.calculatePay(employee, overtimeRate).  
HourlyPayCalculator.calculatePay(nhân viên, tỷ lệ làm thêm giờ).

Again, this seems like a reasonable staticfunction. It doesn’t operate on any particular object and gets all it’s data from it’s arguments. However, there is a reasonable chance that we’ll want this function to be polymorphic. We may wish to implement several different algorithms for calculating hourly pay, for example, OvertimeHourlyPayCalculator and StraightTimeHourlyPayCalculator. So in this case the function should not be static. It should be a nonstatic member function of Employee.  
Một lần nữa, đây có vẻ là một hàm tĩnh hợp lý. Nó không hoạt động trên bất kỳ đối tượng cụ thể nào và lấy tất cả dữ liệu từ các đối số của nó. Tuy nhiên, có một cơ hội hợp lý là chúng ta sẽ muốn chức năng này là đa hình. Chúng tôi có thể muốn triển khai một số thuật toán khác nhau để tính lương theo giờ, ví dụ: Máy tính trả lương làm thêm giờ và Máy tính trả lương theo giờ thẳng. Vì vậy, trong trường hợp này, chức năng không được tĩnh. Nó phải là một chức năng thành viên không cố định của Nhân viên.

In general you should prefer nonstatic methods to static methods. When in doubt, make the function nonstatic. If you really want a function to be static, make sure that there is no chance that you’ll want it to behave polymorphically.  
Nói chung, bạn nên ưu tiên các phương thức không tĩnh hơn các phương thức tĩnh. Khi nghi ngờ, hãy làm cho chức năng không tĩnh. Nếu bạn thực sự muốn một hàm tĩnh, hãy đảm bảo rằng bạn sẽ không muốn nó hoạt động đa hình.

[**G19: *Use Explanatory Variables***](#_page_609_0)  
G19: Sử dụng biến giải thích

Kent Beck wrote about this in his great book *Smalltalk Best Practice Patterns*8 and again more recently in his equally great book *Implementation Patterns*.9 One of the more power-ful ways to make a program readable is to break the calculations up into intermediate val-ues that are held in variables with meaningful names.  
Kent Beck đã viết về điều này trong cuốn sách tuyệt vời của ông Smalltalk Best Practice Patterns8 và một lần nữa gần đây trong cuốn sách tuyệt vời không kém của ông Các mẫu triển khai.9 Một trong những cách mạnh mẽ hơn để làm cho chương trình có thể đọc được là chia các phép tính thành các giá trị trung gian được giữ trong các biến có tên có ý nghĩa.

8. [Beck97], p. 108. 9. [Beck07].

<https://www.facebook.com/groups/phattrienlaptrinh>

**General** 297  
Tổng hợp 297

Consider this example from FitNesse:  
Xem xét ví dụ này từ FitNesse:

Matcher match = headerPattern.matcher(line); if(match.find())

{

String key = match.group(1); String value = match.group(2);

headers.put(key.toLowerCase(), value); }

The simple use of explanatory variables makes it clear that the ﬁrst matched group is the *key,* and the second matched group is the *value*.  
Việc sử dụng đơn giản các biến giải thích cho thấy rõ rằng nhóm phù hợp đầu tiên là khóa và nhóm phù hợp thứ hai là giá trị.

It is hard to overdo this. More explanatory variables are generally better than fewer. It is remarkable how an opaque module can suddenly become transparent simply by break-ing the calculations up into well-named intermediate values.  
Thật khó để lạm dụng điều này. Nhiều biến giải thích thường tốt hơn ít hơn. Điều đáng chú ý là làm thế nào một mô-đun mờ có thể đột nhiên trở nên trong suốt chỉ bằng cách chia nhỏ các phép tính thành các giá trị trung gian được đặt tên rõ ràng.

[**G20: *Function Names Should Say What They Do***](#_page_609_0)  
G20: Tên chức năng nên nói những gì họ làm

Look at this code:  
Nhìn vào mã này:

Date newDate = date.add(5);

Would you expect this to add ﬁve days to the date? Or is it weeks, or hours? Is the date instance changed or does the function just return a new Datewithout changing the old one? *You can’t tell from the call what the function does*.  
Bạn có mong đợi điều này sẽ thêm năm ngày vào ngày không? Hay là vài tuần, hay vài giờ? Phiên bản ngày có bị thay đổi hay hàm chỉ trả về một Ngày mới mà không thay đổi ngày cũ? Bạn không thể biết chức năng đó làm gì từ cuộc gọi.

If the function adds ﬁve days to the date and changes the date, then it should be called addDaysToor increaseByDays. If, on the other hand, the function returns a new date that is ﬁve days later but does not change the date instance, it should be called daysLateror daysSince.  
Nếu hàm thêm năm ngày vào ngày và thay đổi ngày, thì hàm đó sẽ được gọi là addDaysToor boostByDays. Mặt khác, nếu hàm trả về một ngày mới trễ hơn năm ngày nhưng không thay đổi thể hiện ngày, thì hàm đó sẽ được gọi là daysLater hoặc daysSince.

If you have to look at the implementation (or documentation) of the function to know what it does, then you should work to ﬁnd a better name or rearrange the functionality so that it can be placed in functions with better names.  
Nếu bạn phải xem cách triển khai (hoặc tài liệu) của hàm để biết nó làm gì, thì bạn nên tìm một cái tên hay hơn hoặc sắp xếp lại chức năng để nó có thể được đặt trong các hàm có tên hay hơn.

[**G21: *Understand the Algorithm***](#_page_609_0)  
G21: Tìm hiểu thuật toán

Lots of very funny code is written because people don’t take the time to understand the algorithm. They get something to work by plugging in enough ifstatements and ﬂags, without really stopping to consider what is really going on.  
Rất nhiều mã rất buồn cười được viết bởi vì mọi người không dành thời gian để hiểu thuật toán. Họ làm cho một cái gì đó hoạt động bằng cách cắm đủ các câu lệnh if và flag mà không thực sự dừng lại để xem xét điều gì đang thực sự xảy ra.

Programming is often an exploration. You *think* you know the right algorithm for something, but then you wind up ﬁddling with it, prodding and poking at it, until you get it to “work.” How do you know it “works”? Because it passes the test cases you can think of.  
Lập trình thường là một cuộc khám phá. Bạn nghĩ rằng bạn biết thuật toán phù hợp cho một thứ gì đó, nhưng sau đó bạn lại loay hoay với nó, chọc và chọc vào nó, cho đến khi bạn làm cho nó “hoạt động được”. Làm thế nào để bạn biết nó "hoạt động"? Bởi vì nó vượt qua các trường hợp thử nghiệm mà bạn có thể nghĩ ra.

There is nothing wrong with this approach. Indeed, often it is the only way to get a function to do what you think it should. However, it is not sufﬁcient to leave the quotation marks around the word “work.”  
Không có gì sai với cách tiếp cận này. Thật vậy, thường thì đó là cách duy nhất để khiến một chức năng thực hiện những gì bạn nghĩ rằng nó nên làm. Tuy nhiên, để lại dấu ngoặc kép xung quanh từ “công việc” là chưa đủ.

<https://www.facebook.com/groups/phattrienlaptrinh>

298 **Chapter 17: Smells and Heuristics**  
298 Chương 17: Mùi và Heuristic

Before you consider yourself to be done with a function, make sure you *understand* how it works. It is not good enough that it passes all the tests. You must *know*10 that the solution is correct.  
Trước khi bạn cho rằng mình đã hoàn thành một chức năng, hãy đảm bảo rằng bạn hiểu cách thức hoạt động của nó. Nó không đủ tốt để vượt qua tất cả các bài kiểm tra. Bạn phải biết10 rằng giải pháp là chính xác.

Often the best way to gain this knowledge and understanding is to refactor the func-tion into something that is so clean and expressive that it is *obvious* how it works.  
Thông thường, cách tốt nhất để đạt được kiến thức và sự hiểu biết này là tái cấu trúc chức năng thành thứ gì đó rõ ràng và biểu cảm đến mức có thể thấy rõ cách thức hoạt động của nó.

[**G22: *Make Logical Dependencies Physical***](#_page_609_0)  
G22: Biến các phụ thuộc logic thành vật lý

If one module depends upon another, that dependency should be physical, not just logical. The dependent module should not make assumptions (in other words, logical dependen-cies) about the module it depends upon. Rather it should explicitly ask that module for all the information it depends upon.  
Nếu một mô-đun phụ thuộc vào mô-đun khác, thì sự phụ thuộc đó phải là vật lý, không chỉ logic. Mô-đun phụ thuộc không nên đưa ra các giả định (nói cách khác, các phụ thuộc logic) về mô-đun mà nó phụ thuộc. Thay vào đó, nó nên yêu cầu mô-đun đó một cách rõ ràng về tất cả thông tin mà nó phụ thuộc vào.

For example, imagine that you are writing a function that prints a plain text report of hours worked by employees. One class named HourlyReportergathers all the data into a convenient form and then passes it to HourlyReportFormatterto print it. (See Listing 17-1.)  
Ví dụ, hãy tưởng tượng rằng bạn đang viết một hàm in một báo cáo văn bản thuần túy về số giờ làm việc của nhân viên. Một lớp có tên là HourlyReporter tập hợp tất cả dữ liệu vào một biểu mẫu thuận tiện rồi chuyển nó tới HourlyReportFormatter để in ra. (Xem Liệt kê 17-1.)

**Listing 17-1 HourlyReporter.java**  
Liệt kê 17-1 HourlyReporter.java

public class HourlyReporter {

private HourlyReportFormatter formatter; private List<LineItem> page;

private final int PAGE\_SIZE = 55;

public HourlyReporter(HourlyReportFormatter formatter) { this.formatter = formatter;

page = new ArrayList<LineItem>(); }

public void generateReport(List<HourlyEmployee> employees) { for (HourlyEmployee e : employees) {

addLineItemToPage(e);

if (page.size() == PAGE\_SIZE) printAndClearItemList();

}

if (page.size() > 0) printAndClearItemList();

}

private void printAndClearItemList() { formatter.format(page); page.clear();

}

private void addLineItemToPage(HourlyEmployee e) { LineItem item = new LineItem();

item.name = e.getName();

item.hours = e.getTenthsWorked() / 10;

10. There is a difference between knowing how the code works and knowing whether the algorithm will do the job required of it. Being unsure that an algorithm is appropriate is often a fact of life. Being unsure what your code does is just laziness.  
10. Có một sự khác biệt giữa việc biết mã hoạt động như thế nào và biết liệu thuật toán có thực hiện công việc được yêu cầu hay không. Không chắc chắn rằng một thuật toán là phù hợp thường là một thực tế của cuộc sống. Không chắc mã của bạn làm gì chỉ là sự lười biếng.

<https://www.facebook.com/groups/phattrienlaptrinh>

**General** 299  
Tổng hợp 299

**Listing 17-1 (continued)**

**HourlyReporter.java  
HourlyReporter.java**

item.tenths = e.getTenthsWorked() % 10; page.add(item);

}

public class LineItem { public String name; public int hours; public int tenths;

} }

This code has a logical dependency that has not been physicalized. Can you spot it? It is the constant PAGE\_SIZE. Why should the HourlyReporter know the size of the page? Page size should be the responsibility of the HourlyReportFormatter.  
Mã này có một phụ thuộc logic chưa được vật lý hóa. Bạn có thể phát hiện ra nó? Đó là hằng số PAGE\_SIZE. Tại sao HourlyReporter nên biết kích thước của trang? Kích thước trang phải là trách nhiệm của HourlyReportFormatter.

The fact that PAGE\_SIZE is declared in HourlyReporter represents a misplaced responsibility [G17] that causes HourlyReporterto assume that it knows what the page size ought to be. Such an assumption is a logical dependency. HourlyReporterdepends on the fact that HourlyReportFormatter can deal with page sizes of 55. If some implementation of HourlyReportFormatter could not deal with such sizes, then there would be an error.  
Thực tế là PAGE\_SIZE được khai báo trong HourlyReporter thể hiện trách nhiệm không đúng chỗ [G17] khiến HourlyReporter cho rằng nó biết kích thước trang phải là bao nhiêu. Một giả định như vậy là một sự phụ thuộc logic. HourlyReporter phụ thuộc vào thực tế là HourlyReportFormatter có thể xử lý các kích thước trang là 55. Nếu một số triển khai của HourlyReportFormatter không thể xử lý các kích thước như vậy thì sẽ xảy ra lỗi.

We can physicalize this dependency by creating a new method in HourlyReport-Formatternamed getMaxPageSize(). HourlyReporterwill then call that function rather than using the PAGE\_SIZE constant.  
Chúng ta có thể vật lý hóa sự phụ thuộc này bằng cách tạo một phương thức mới trong HourlyReport-Formatternamed getMaxPageSize(). Sau đó, HourlyReporter sẽ gọi hàm đó thay vì sử dụng hằng số PAGE\_SIZE.

[**G23: *Prefer Polymorphism to If/Else or Switch/Case***](#_page_609_0)  
G23: Ưu tiên Đa hình hơn Nếu/Khác hoặc Chuyển đổi/Trường hợp

This might seem a strange suggestion given the topic of Chapter 6. After all, in that chapter I make the point that switch statements are probably appropriate in the parts of the system where adding new functions is more likely than adding new types.  
Đây có vẻ là một gợi ý kỳ lạ với chủ đề của Chương 6. Xét cho cùng, trong chương đó, tôi đã đưa ra quan điểm rằng các câu lệnh chuyển đổi có thể phù hợp trong các bộ phận của hệ thống mà việc thêm các chức năng mới có nhiều khả năng hơn là thêm các loại mới.

First, most people use switch statements because it’s the obvious brute force solution, not because it’s the right solution for the situation. So this heuristic is here to remind us to consider polymorphism before using a switch.  
Đầu tiên, hầu hết mọi người sử dụng các câu lệnh chuyển đổi vì đó là giải pháp vũ phu rõ ràng chứ không phải vì đó là giải pháp phù hợp cho tình huống. Vì vậy, heuristic này ở đây để nhắc nhở chúng ta xem xét tính đa hình trước khi sử dụng switch.

Second, the cases where functions are more volatile than types are relatively rare. So *every* switch statement should be suspect.  
Thứ hai, các trường hợp chức năng dễ bay hơi hơn các loại là tương đối hiếm. Vì vậy, mọi tuyên bố chuyển đổi nên bị nghi ngờ.

I use the following “ONE SWITCH” rule: *There may be no more than one switch state-ment for a given type of selection. The cases in that switch statement must create polymor-phic objects that take the place of other such switch statements in the rest of the system.*  
Tôi sử dụng quy tắc “MỘT CÔNG TẮC” sau: Có thể không có nhiều hơn một trạng thái chuyển đổi cho một loại lựa chọn nhất định. Các trường hợp trong câu lệnh chuyển đổi đó phải tạo ra các đối tượng đa hình thay thế cho các câu lệnh chuyển đổi khác như vậy trong phần còn lại của hệ thống.

[**G24: *Follow Standard Conventions***](#_page_609_0)  
G24: Tuân thủ các quy ước tiêu chuẩn

Every team should follow a coding standard based on common industry norms. This cod-ing standard should specify things like where to declare instance variables; how to name classes, methods, and variables; where to put braces; and so on. The team should not need a document to describe these conventions because their code provides the examples.  
Mỗi nhóm nên tuân theo một tiêu chuẩn mã hóa dựa trên các tiêu chuẩn chung của ngành. Tiêu chuẩn viết mã này nên chỉ định những thứ như nơi khai báo các biến thể hiện; cách đặt tên lớp, phương thức và biến; niềng răng ở đâu; và như thế. Nhóm không cần tài liệu để mô tả các quy ước này vì mã của họ cung cấp các ví dụ.

<https://www.facebook.com/groups/phattrienlaptrinh>

300 **Chapter 17: Smells and Heuristics**  
300 Chương 17: Mùi và Heuristic

Everyone on the team should follow these conventions. This means that each team member must be mature enough to realize that it doesn’t matter a whit where you put your braces so long as you all agree on where to put them.  
Mọi người trong nhóm nên tuân theo các quy ước này. Điều này có nghĩa là mỗi thành viên trong nhóm phải đủ trưởng thành để nhận ra rằng việc bạn đặt niềng răng ở đâu không quan trọng miễn là tất cả các bạn đều đồng ý về vị trí đặt chúng.

If you would like to know what conventions I follow, you’ll see them in the refactored code in Listing B-7 on page 394, through Listing B-14.  
Nếu bạn muốn biết tôi tuân theo những quy ước nào, bạn sẽ thấy chúng trong mã được cấu trúc lại trong Liệt kê B-7 trên trang 394, đến hết Liệt kê B-14.

[**G25: *Replace Magic Numbers with Named Constants***](#_page_609_0)  
G25: Thay thế số ma thuật bằng hằng số được đặt tên

This is probably one of the oldest rules in software development. I remember reading it in the late sixties in introductory COBOL, FORTRAN, and PL/1 manuals. In general it is a bad idea to have raw numbers in your code. You should hide them behind well-named constants.  
Đây có lẽ là một trong những quy tắc lâu đời nhất trong phát triển phần mềm. Tôi nhớ đã đọc nó vào cuối những năm sáu mươi trong hướng dẫn giới thiệu về COBOL, FORTRAN và PL/1. Nói chung, bạn không nên có số liệu thô trong mã của mình. Bạn nên ẩn chúng đằng sau các hằng số được đặt tên tốt.

For example, the number 86,400 should be hidden behind the constant SECONDS\_PER\_DAY. If you are printing 55 lines per page, then the constant 55 should be hid-den behind the constant LINES\_PER\_PAGE.  
Ví dụ: số 86.400 phải được ẩn sau hằng số SECONDS\_PER\_DAY. Nếu bạn đang in 55 dòng trên mỗi trang, thì hằng số 55 phải được ẩn đằng sau hằng số LINES\_PER\_PAGE.

Some constants are so easy to recognize that they don’t always need a named constant to hide behind so long as they are used in conjunction with very self-explanatory code. For example:  
Một số hằng số rất dễ nhận ra rằng chúng không phải lúc nào cũng cần một hằng số được đặt tên để ẩn đằng sau miễn là chúng được sử dụng cùng với mã rất dễ hiểu. Ví dụ:

double milesWalked = feetWalked/5280.0; int dailyPay = hourlyRate \* 8;

double circumference = radius \* Math.PI \* 2;

Do we really need the constants FEET\_PER\_MILE, WORK\_HOURS\_PER\_DAY, and TWOin the above examples? Clearly, the last case is absurd. There are some formulae in which con-stants are simply better written as raw numbers. You might quibble about the WORK\_HOURS\_PER\_DAYcase because the laws or conventions might change. On the other hand, that formula reads so nicely with the 8 in it that I would be reluctant to add 17 extra characters to the readers’ burden. And in the FEET\_PER\_MILEcase, the number 5280 is so very well known and so unique a constant that readers would recognize it even if it stood alone on a page with no context surrounding it.  
Chúng ta có thực sự cần các hằng số FEET\_PER\_MILE, WORK\_HOURS\_PER\_DAY và TWO trong các ví dụ trên không? Rõ ràng, trường hợp cuối cùng là vô lý. Có một số công thức trong đó các hằng số được viết đơn giản hơn dưới dạng số nguyên. Bạn có thể tranh cãi về trường hợp WORK\_HOURS\_PER\_DAY vì luật hoặc quy ước có thể thay đổi. Mặt khác, công thức đó đọc rất hay với số 8 trong đó nên tôi sẽ miễn cưỡng thêm 17 ký tự phụ vào gánh nặng của người đọc. Và trong trường hợp FEET\_PER\_MILE, số 5280 là một hằng số rất nổi tiếng và độc đáo đến mức người đọc sẽ nhận ra nó ngay cả khi nó đứng một mình trên một trang không có ngữ cảnh xung quanh.

Constants like 3.141592653589793 are also very well known and easily recognizable. However, the chance for error is too great to leave them raw. Every time someone sees 3.1415927535890793, they know that it is p*,* and so they fail to scrutinize it. (Did you catch the single-digit error?) We also don’t want people using 3.14, 3.14159, 3.142, and so forth. Therefore, it is a good thing that Math.PI has already been deﬁned for us.  
Các hằng số như 3.141592653589793 cũng rất nổi tiếng và dễ nhận biết. Tuy nhiên, khả năng xảy ra sai sót là quá lớn để có thể để nguyên chúng. Mỗi khi ai đó nhìn thấy 3.1415927535890793, họ biết rằng đó là 3.1415927535890793 và vì vậy họ không xem xét kỹ lưỡng. (Bạn có bắt gặp lỗi một chữ số không?) Chúng tôi cũng không muốn mọi người sử dụng 3.14, 3.14159, 3.142, v.v. Do đó, thật tốt khi Math.PI đã được định nghĩa sẵn cho chúng ta.

The term “Magic Number” does not apply only to numbers. It applies to any token that has a value that is not self-describing. For example:  
Thuật ngữ "Số ma thuật" không chỉ áp dụng cho các con số. Nó áp dụng cho bất kỳ mã thông báo nào có giá trị không tự mô tả. Ví dụ:

assertEquals(7777, Employee.find(“John Doe”).employeeNumber());

There are two magic numbers in this assertion. The ﬁrst is obviously 7777, though what it might mean is not obvious. The second magic number is "John Doe," and again the intent is not clear.  
Có hai con số kỳ diệu trong khẳng định này. Đầu tiên rõ ràng là 7777, mặc dù ý nghĩa của nó không rõ ràng. Con số kỳ diệu thứ hai là "John Doe", và một lần nữa ý định không rõ ràng.

It turns out that "John Doe"is the name of employee #7777 in a well-known test data-base created by our team. Everyone in the team knows that when you connect to this  
Hóa ra "John Doe" là tên của nhân viên #7777 trong cơ sở dữ liệu thử nghiệm nổi tiếng do nhóm của chúng tôi tạo ra. Mọi người trong nhóm đều biết rằng khi bạn kết nối với điều này

<https://www.facebook.com/groups/phattrienlaptrinh>

**General** 301  
Tổng hợp 301

database, it will have several employees already cooked into it with well-known values and attributes. It also turns out that "John Doe"represents the sole hourly employee in that test database. So this test should really read:  
cơ sở dữ liệu, nó sẽ có một số nhân viên đã chuẩn bị sẵn các giá trị và thuộc tính nổi tiếng. Hóa ra "John Doe" đại diện cho nhân viên làm việc theo giờ duy nhất trong cơ sở dữ liệu thử nghiệm đó. Vì vậy, bài kiểm tra này thực sự nên đọc:

assertEquals( HOURLY\_EMPLOYEE\_ID,

Employee.find(HOURLY\_EMPLOYEE\_NAME).employeeNumber());

[**G26: *Be Precise***](#_page_609_0)  
G26: Chính xác

Expecting the ﬁrst match to be the *only* match to a query is probably naive. Using ﬂoating point numbers to represent currency is almost criminal. Avoiding locks and/or transaction management because you don’t think concurrent update is likely is lazy at best. Declaring a variable to be an ArrayListwhen a Listwill due is overly constraining. Making all vari-ables protected by default is not constraining enough.  
Mong đợi kết quả khớp đầu tiên là kết quả khớp duy nhất cho một truy vấn có lẽ là ngây thơ. Sử dụng số dấu phẩy động để đại diện cho tiền tệ gần như là tội phạm. Tránh khóa và/hoặc quản lý giao dịch vì bạn không nghĩ rằng cập nhật đồng thời có thể là lười biếng nhất. Khai báo một biến là một ArrayList khi một Listwill đến hạn bị ràng buộc quá mức. Đặt tất cả các biến được bảo vệ theo mặc định là không đủ ràng buộc.

When you make a decision in your code, make sure you make it *precisely*. Know why you have made it and how you will deal with any exceptions. Don’t be lazy about the pre-cision of your decisions. If you decide to call a function that might return null, make sure you check for null. If you query for what you think is the only record in the database, make sure your code checks to be sure there aren’t others. If you need to deal with cur-rency, use integers11 and deal with rounding appropriately. If there is the possibility of concurrent update, make sure you implement some kind of locking mechanism.  
Khi bạn đưa ra quyết định trong mã của mình, hãy đảm bảo rằng bạn đưa ra quyết định đó một cách chính xác. Biết lý do tại sao bạn đã thực hiện nó và làm thế nào bạn sẽ đối phó với bất kỳ trường hợp ngoại lệ. Đừng lười biếng về việc chuẩn bị trước các quyết định của bạn. Nếu bạn quyết định gọi một hàm có thể trả về giá trị rỗng, hãy đảm bảo bạn kiểm tra giá trị rỗng. Nếu bạn truy vấn bản ghi mà bạn cho là duy nhất trong cơ sở dữ liệu, hãy đảm bảo mã của bạn kiểm tra để chắc chắn không có bản ghi nào khác. Nếu bạn cần xử lý tiền tệ, hãy sử dụng số nguyên11 và xử lý làm tròn một cách thích hợp. Nếu có khả năng cập nhật đồng thời, hãy đảm bảo bạn triển khai một số loại cơ chế khóa.

Ambiguities and imprecision in code are either a result of disagreements or laziness. In either case they should be eliminated.  
Sự mơ hồ và thiếu chính xác trong mã là kết quả của sự bất đồng hoặc sự lười biếng. Trong cả hai trường hợp, chúng nên được loại bỏ.

[**G27: *Structure over Convention***](#_page_609_0)  
G27: Cấu trúc thay vì Công ước

Enforce design decisions with structure over convention. Naming conventions are good, but they are inferior to structures that force compliance. For example, switch/cases with nicely named enumerations are inferior to base classes with abstract methods. No one is forced to implement the switch/case statement the same way each time; but the base classes do enforce that concrete classes have all abstract methods implemented.  
Thực thi các quyết định thiết kế với cấu trúc hơn quy ước. Các quy ước đặt tên là tốt, nhưng chúng kém hơn so với các cấu trúc bắt buộc tuân thủ. Ví dụ: switch/case với kiểu liệt kê được đặt tên độc đáo sẽ kém hơn lớp cơ sở với phương thức trừu tượng. Không ai bị buộc phải thực hiện câu lệnh switch/case theo cùng một cách mỗi lần; nhưng các lớp cơ sở thực thi rằng các lớp cụ thể có tất cả các phương thức trừu tượng được triển khai.

[**G28: *Encapsulate Conditionals***](#_page_609_0)  
G28: Đóng gói điều kiện

Boolean logic is hard enough to understand without having to see it in the context of an if or while statement. Extract functions that explain the intent of the conditional.  
Logic Boolean đủ khó để hiểu mà không cần phải xem nó trong ngữ cảnh của câu lệnh if hoặc while. Trích xuất các chức năng giải thích mục đích của điều kiện.

For example:  
Ví dụ:

if (shouldBeDeleted(timer)) is preferable to

if (timer.hasExpired() && !timer.isRecurrent())

11. Or better yet, a Money class that uses integers.  
11. Hoặc tốt hơn nữa là một lớp Tiền sử dụng số nguyên.

<https://www.facebook.com/groups/phattrienlaptrinh>

302 **Chapter 17: Smells and Heuristics**  
302 Chương 17: Mùi và Heuristic

[**G29:*Avoid Negative Conditionals***](#_page_609_0)  
G29:Tránh câu điều kiện phủ định

Negatives are just a bit harder to understand than positives. So, when possible, condition-als should be expressed as positives. For example:  
Tiêu cực chỉ là một chút khó hiểu hơn tích cực. Vì vậy, khi có thể, các điều kiện nên được thể hiện dưới dạng tích cực. Ví dụ:

if (buffer.shouldCompact()) is preferable to

if (!buffer.shouldNotCompact())

[**G30: *Functions Should Do One Thing***](#_page_609_0)  
G30: Các chức năng nên làm một việc

It is often tempting to create functions that have multiple sections that perform a series of operations. Functions of this kind do more than *one thing*, and should be converted into many smaller functions, each of which does *one thing*.  
Việc tạo các hàm có nhiều phần thực hiện một loạt các thao tác thường rất hấp dẫn. Các chức năng loại này làm nhiều hơn một việc và nên được chuyển đổi thành nhiều chức năng nhỏ hơn, mỗi chức năng thực hiện một việc.

For example:  
Ví dụ:

public void pay() {

for (Employee e : employees) { if (e.isPayday()) {

Money pay = e.calculatePay(); e.deliverPay(pay);

} }

}

This bit of code does three things. It loops over all the employees, checks to see whether each employee ought to be paid, and then pays the employee. This code would be better written as:  
Đoạn mã này thực hiện ba việc. Nó lặp lại tất cả các nhân viên, kiểm tra xem liệu mỗi nhân viên có nên được trả lương hay không và sau đó trả tiền cho nhân viên đó. Mã này sẽ được viết tốt hơn như sau:

public void pay() {

for (Employee e : employees) payIfNecessary(e);

}

private void payIfNecessary(Employee e) { if (e.isPayday())

calculateAndDeliverPay(e); }

private void calculateAndDeliverPay(Employee e) { Money pay = e.calculatePay(); e.deliverPay(pay);

}

Each of these functions does one thing. (See “Do One Thing” on page 35.)  
Mỗi chức năng này làm một việc. (Xem “Làm một việc” ở trang 35.)

[**G31: *Hidden Temporal Couplings***](#_page_609_0)  
G31: Khớp nối thời gian ẩn

Temporal couplings are often necessary, but you should not hide the coupling. Structure the arguments of your functions such that the order in which they should be called is obvi-ous. Consider the following:  
Khớp nối tạm thời thường là cần thiết, nhưng bạn không nên ẩn khớp nối. Cấu trúc các đối số của các chức năng của bạn sao cho thứ tự mà chúng nên được gọi là rõ ràng. Hãy xem xét những điều sau đây:

<https://www.facebook.com/groups/phattrienlaptrinh>

**General** 303  
Tổng hợp 303

public class MoogDiver { Gradient gradient; List<Spline> splines;

public void dive(String reason) { saturateGradient(); reticulateSplines(); diveForMoog(reason);

} ...

}

The order of the three functions is important. You must saturate the gradient before you can reticulate the splines, and only then can you dive for the moog. Unfortunately, the code does not enforce this temporal coupling. Another programmer could call reticulate-Splinesbefore saturateGradientwas called, leading to an UnsaturatedGradientException. A better solution is:  
Thứ tự của ba chức năng là quan trọng. Bạn phải bão hòa gradient trước khi có thể vẽ lại các đường spline và chỉ sau đó bạn mới có thể đi sâu vào moog. Thật không may, mã không thực thi khớp nối thời gian này. Một lập trình viên khác có thể gọi reticulate-Splines trước khi saturateGradient được gọi, dẫn đến một UnsaturatedGradientException. Một giải pháp tốt hơn là:

public class MoogDiver { Gradient gradient; List<Spline> splines;

public void dive(String reason) { Gradient gradient = saturateGradient();

List<Spline> splines = reticulateSplines(gradient); diveForMoog(splines, reason);

} ...

}

This exposes the temporal coupling by creating a bucket brigade. Each function produces a result that the next function needs, so there is no reasonable way to call them out of order.  
Điều này cho thấy khớp nối thời gian bằng cách tạo ra một lữ đoàn xô. Mỗi chức năng tạo ra một kết quả mà chức năng tiếp theo cần, vì vậy không có cách nào hợp lý để gọi chúng không theo thứ tự.

You might complain that this increases the complexity of the functions, and you’d be right. But that extra syntactic complexity exposes the true temporal complexity of the situation.  
Bạn có thể phàn nàn rằng điều này làm tăng độ phức tạp của các chức năng, và bạn đã đúng. Nhưng sự phức tạp thêm về cú pháp đó phơi bày sự phức tạp về thời gian thực sự của tình huống.

Note that I left the instance variables in place. I presume that they are needed by pri-vate methods in the class. Even so, I want the arguments in place to make the temporal coupling explicit.  
Lưu ý rằng tôi đã để nguyên các biến thể hiện. Tôi cho rằng chúng cần thiết cho các phương pháp riêng tư trong lớp. Mặc dù vậy, tôi muốn các đối số được đưa ra để làm cho sự liên kết thời gian trở nên rõ ràng.

[**G32: *Don’t Be Arbitrary***](#_page_609_0)  
G32: Đừng Tùy Ý

Have a reason for the way you structure your code, and make sure that reason is communi-cated by the structure of the code. If a structure appears arbitrary, others will feel empowered to change it. If a structure appears consistently throughout the system, others will use it and preserve the convention. For example, I was recently merging changes to FitNesse and discovered that one of our committers had done this:  
Có lý do cho cách bạn cấu trúc mã của mình và đảm bảo rằng lý do đó được truyền đạt bằng cấu trúc của mã. Nếu một cấu trúc xuất hiện tùy tiện, những người khác sẽ cảm thấy được trao quyền để thay đổi nó. Nếu một cấu trúc xuất hiện một cách nhất quán trong toàn hệ thống, thì những cấu trúc khác sẽ sử dụng cấu trúc đó và duy trì quy ước. Ví dụ: gần đây tôi đã hợp nhất các thay đổi đối với FitNesse và phát hiện ra rằng một trong những người ủy quyền của chúng tôi đã thực hiện việc này:

public class AliasLinkWidget extends ParentWidget {

public static class VariableExpandingWidgetRoot { ...

... }

<https://www.facebook.com/groups/phattrienlaptrinh>

304 **Chapter 17: Smells and Heuristics**  
304 Chương 17: Mùi và Heuristic

The problem with this was that VariableExpandingWidgetRoot had no need to be inside the scope of AliasLinkWidget. Moreover, other unrelated classes made use of AliasLinkWidget.VariableExpandingWidgetRoot. These classes had no need to know about AliasLinkWidget.  
Vấn đề với điều này là VariableExpandingWidgetRoot không cần phải nằm trong phạm vi của AliasLinkWidget. Hơn nữa, các lớp không liên quan khác đã sử dụng AliasLinkWidget.VariableExpandingWidgetRoot. Các lớp này không cần biết về AliasLinkWidget.

Perhaps the programmer had plopped the VariableExpandingWidgetRoot into AliasWidget as a matter of convenience, or perhaps he thought it really needed to be scoped inside AliasWidget. Whatever the reason, the result wound up being arbitrary. Pub-lic classes that are not utilities of some other class should not be scoped inside another class. The convention is to make them public at the top level of their package.  
Có lẽ lập trình viên đã đặt VariableExpandingWidgetRoot vào AliasWidget như một vấn đề thuận tiện, hoặc có lẽ anh ta nghĩ rằng nó thực sự cần được đặt trong AliasWidget. Dù lý do là gì, kết quả cuối cùng là tùy tiện. Các lớp công khai không phải là tiện ích của một số lớp khác không nên nằm trong phạm vi của một lớp khác. Quy ước là đặt chúng ở chế độ công khai ở cấp cao nhất trong gói của chúng.

[**G33: *Encapsulate Boundary Conditions***](#_page_609_0)  
G33: Đóng gói các điều kiện biên

Boundary conditions are hard to keep track of. Put the processing for them in one place. Don’t let them leak all over the code. We don’t want swarms of +1s and -1s scattered hither and yon. Consider this simple example from FIT:  
Các điều kiện biên rất khó theo dõi. Đặt xử lý cho chúng ở một nơi. Đừng để chúng bị rò rỉ trên toàn bộ mã. Chúng tôi không muốn hàng loạt +1 và -1 nằm rải rác khắp nơi. Hãy xem xét ví dụ đơn giản này từ FIT:

if(level + 1 < tags.length) {

parts = new Parse(body, tags, level + 1, offset + endTag); body = null;

}

Notice that level+1appears twice. This is a boundary condition that should be encapsu-lated within a variable named something like nextLevel.  
Lưu ý rằng mức +1 xuất hiện hai lần. Đây là một điều kiện biên nên được gói gọn trong một biến có tên như nextLevel.

int nextLevel = level + 1; if(nextLevel < tags.length) {

parts = new Parse(body, tags, nextLevel, offset + endTag); body = null;

}

[**G34: *Functions Should Descend Only One Level of Abstraction***](#_page_609_0)  
G34: Các chức năng chỉ nên giảm xuống một mức trừu tượng

The statements within a function should all be written at the same level of abstraction, which should be one level below the operation described by the name of the function. This may be the hardest of these heuristics to interpret and follow. Though the idea is plain enough, humans are just far too good at seamlessly mixing levels of abstraction. Consider, for example, the following code taken from FitNesse:  
Tất cả các câu lệnh trong một hàm phải được viết ở cùng một mức trừu tượng, mức này phải thấp hơn một mức so với thao tác được mô tả bằng tên của hàm. Đây có thể là khó khăn nhất trong số những kinh nghiệm này để giải thích và làm theo. Mặc dù ý tưởng là đủ đơn giản, nhưng con người quá giỏi trong việc trộn lẫn các mức độ trừu tượng một cách liền mạch. Ví dụ, hãy xem xét đoạn mã sau được lấy từ FitNesse:

public String render() throws Exception {

StringBuffer html = new StringBuffer("<hr"); if(size > 0)

html.append(" size=\"").append(size + 1).append("\""); html.append(">");

return html.toString(); }

<https://www.facebook.com/groups/phattrienlaptrinh>

**General** 305  
Tổng hợp 305

A moment’s study and you can see what’s going on. This function constructs the HTML tag that draws a horizontal rule across the page. The height of that rule is speciﬁed in the size variable.  
Nghiên cứu một lát và bạn có thể thấy điều gì đang xảy ra. Chức năng này xây dựng thẻ HTML vẽ một quy tắc ngang trên trang. Chiều cao của quy tắc đó được chỉ định trong biến kích thước.

Now look again. This method is mixing at least two levels of abstraction. The ﬁrst is the notion that a horizontal rule has a size. The second is the syntax of the HRtag itself. This code comes from the HruleWidgetmodule in FitNesse. This module detects a row of four or more dashes and converts it into the appropriate HR tag. The more dashes, the larger the size.  
Bây giờ nhìn lại. Phương pháp này đang trộn lẫn ít nhất hai mức độ trừu tượng. Đầu tiên là khái niệm rằng một quy tắc ngang có kích thước. Thứ hai là cú pháp của chính HRtag. Mã này đến từ mô-đun HruleWidget trong FitNesse. Mô-đun này phát hiện một hàng gồm bốn dấu gạch ngang trở lên và chuyển đổi nó thành thẻ nhân sự thích hợp. Càng nhiều dấu gạch ngang, kích thước càng lớn.

I refactored this bit of code as follows. Note that I changed the name of the sizeﬁeld to reﬂect its true purpose. It held the number of extra dashes.  
Tôi đã cấu trúc lại đoạn mã này như sau. Lưu ý rằng tôi đã thay đổi tên của trường kích thước để phản ánh mục đích thực sự của nó. Nó chứa số lượng dấu gạch ngang bổ sung.

public String render() throws Exception {

HtmlTag hr = new HtmlTag("hr"); if (extraDashes > 0)

hr.addAttribute("size", hrSize(extraDashes)); return hr.html();

}

private String hrSize(int height) {

int hrSize = height + 1;

return String.format("%d", hrSize); }

This change separates the two levels of abstraction nicely.The renderfunction simply con-structs an HR tag, without having to know anything about the HTML syntax of that tag. The HtmlTag module takes care of all the nasty syntax issues.  
Sự thay đổi này tách biệt hai cấp độ trừu tượng một cách độc đáo. Hàm kết xuất chỉ đơn giản là tạo cấu trúc thẻ nhân sự mà không cần phải biết bất kỳ điều gì về cú pháp HTML của thẻ đó. Mô-đun HtmlTag xử lý tất cả các vấn đề khó chịu về cú pháp.

Indeed, by making this change I caught a subtle error. The original code did not put the closing slash on the HR tag, as the XHTML standard would have it. (In other words, it emitted <hr> instead of <hr/>.) The HtmlTag module had been changed to conform to XHTML long ago.  
Thật vậy, bằng cách thực hiện thay đổi này, tôi đã mắc phải một lỗi nhỏ. Mã ban đầu không đặt dấu gạch chéo đóng trên thẻ nhân sự, vì tiêu chuẩn XHTML sẽ có nó. (Nói cách khác, nó phát ra <hr> thay vì <hr/>.) Mô-đun HtmlTag đã được thay đổi để phù hợp với XHTML từ lâu.

Separating levels of abstraction is one of the most important functions of refactor-ing, and it’s one of the hardest to do well. As an example, look at the code below. This was my ﬁrst attempt at separating the abstraction levels in the HruleWidget.render method.  
Tách các mức độ trừu tượng là một trong những chức năng quan trọng nhất của quá trình tái cấu trúc và là một trong những chức năng khó thực hiện tốt nhất. Ví dụ, hãy xem đoạn mã dưới đây. Đây là nỗ lực đầu tiên của tôi trong việc tách các mức độ trừu tượng trong phương thức HruleWidget.render.

public String render() throws Exception {

HtmlTag hr = new HtmlTag("hr"); if (size > 0) {

hr.addAttribute("size", ""+(size+1)); }

return hr.html(); }

My goal, at this point, was to create the necessary separation and get the tests to pass. I accomplished that goal easily, but the result was a function that *still* had mixed levels of abstraction. In this case the mixed levels were the construction of the HR tag and the  
Mục tiêu của tôi, tại thời điểm này, là tạo ra sự tách biệt cần thiết và vượt qua các bài kiểm tra. Tôi đã hoàn thành mục tiêu đó một cách dễ dàng, nhưng kết quả là một chức năng vẫn có nhiều mức độ trừu tượng khác nhau. Trong trường hợp này, các cấp độ hỗn hợp là việc xây dựng thẻ nhân sự và

<https://www.facebook.com/groups/phattrienlaptrinh>

306 **Chapter 17: Smells and Heuristics**  
306 Chương 17: Mùi và Heuristic

interpretation and formatting of the sizevariable. This points out that when you break a function along lines of abstraction, you often uncover new lines of abstraction that were obscured by the previous structure.  
giải thích và định dạng của sizevariable. Điều này chỉ ra rằng khi bạn ngắt một chức năng dọc theo các dòng trừu tượng, bạn thường phát hiện ra các dòng trừu tượng mới bị che khuất bởi cấu trúc trước đó.

[**G35: *Keep Conﬁgurable Data at High Levels***](#_page_609_0)  
G35: Giữ dữ liệu có thể lập cấu hình ở mức cao

If you have a constant such as a default or conﬁguration value that is known and expected at a high level of abstraction, do not bury it in a low-level function. Expose it as an argu-ment to that low-level function called from the high-level function. Consider the following code from FitNesse:  
Nếu bạn có một hằng số chẳng hạn như giá trị mặc định hoặc cấu hình đã biết và được mong đợi ở mức độ trừu tượng cao, đừng chôn vùi nó trong một hàm mức thấp. Hiển thị nó như một đối số cho chức năng cấp thấp được gọi từ chức năng cấp cao. Hãy xem xét đoạn mã sau từ FitNesse:

public static void main(String[] args) throws Exception {

Arguments arguments = parseCommandLine(args); ...

}

public class Arguments {

public static final String DEFAULT\_PATH = ".";

public static final String DEFAULT\_ROOT = "FitNesseRoot"; public static final int DEFAULT\_PORT = 80;

public static final int DEFAULT\_VERSION\_DAYS = 14; ...

}

The command-line arguments are parsed in the very ﬁrst executable line of FitNesse. The default values of those arguments are speciﬁed at the top of the Argumentclass. You don’t have to go looking in low levels of the system for statements like this one:  
Các đối số dòng lệnh được phân tích cú pháp trong dòng thực thi đầu tiên của FitNesse. Các giá trị mặc định của các đối số đó được chỉ định ở trên cùng của Argumentclass. Bạn không cần phải tìm kiếm các câu lệnh như thế này ở các cấp độ thấp của hệ thống:

if (arguments.port == 0) // use 80 by default

The conﬁguration constants reside at a very high level and are easy to change. They get passed down to the rest of the application. The lower levels of the application do not own the values of these constants.  
Các hằng số cấu hình nằm ở mức rất cao và dễ dàng thay đổi. Chúng được chuyển xuống phần còn lại của ứng dụng. Các cấp thấp hơn của ứng dụng không sở hữu các giá trị của các hằng số này.

[**G36:*Avoid Transitive Navigation***](#_page_609_0)  
G36:Tránh điều hướng chuyển tiếp

In general we don’t want a single module to know much about its collaborators. More spe-ciﬁcally, if Acollaborates with B, and Bcollaborates with C, we don’t want modules that use A to know about C. (For example, we don’t want a.getB().getC().doSomething();.)  
Nói chung, chúng tôi không muốn một mô-đun biết nhiều về các cộng tác viên của nó. Cụ thể hơn, nếu A cộng tác với B và B cộng tác với C, chúng tôi không muốn các mô-đun sử dụng A biết về C. (Ví dụ: chúng tôi không muốn a.getB().getC().doSomething( );.)

This is sometimes called the Law of Demeter. The Pragmatic Programmers call it “Writing Shy Code.”12 In either case it comes down to making sure that modules know only about their immediate collaborators and do not know the navigation map of the whole system.  
Điều này đôi khi được gọi là Luật Demeter. Các Lập trình viên Thực dụng gọi nó là “Viết mã nhút nhát.”12 Trong cả hai trường hợp, điều quan trọng là phải đảm bảo rằng các mô-đun chỉ biết về các cộng tác viên trực tiếp của chúng và không biết bản đồ điều hướng của toàn bộ hệ thống.

If many modules used some form of the statement a.getB().getC(), then it would be difﬁcult to change the design and architecture to interpose a Qbetween Band C. You’d  
Nếu nhiều mô-đun sử dụng một số dạng của câu lệnh a.getB().getC(), thì sẽ khó thay đổi thiết kế và kiến trúc để xen kẽ một Qbetween Band C. Bạn sẽ

12. [PRAG], p. 138.

<https://www.facebook.com/groups/phattrienlaptrinh>

**Java** 307  
java307

have to ﬁnd every instance of a.getB().getC() and convert it to a.getB().getQ().getC(). This is how architectures become rigid. Too many modules know too much about the architecture.  
phải tìm mọi phiên bản của a.getB().getC() và chuyển nó thành a.getB().getQ().getC(). Đây là cách kiến trúc trở nên cứng nhắc. Quá nhiều mô-đun biết quá nhiều về kiến trúc.

Rather we want our immediate collaborators to offer all the services we need. We should not have to roam through the object graph of the system, hunting for the method we want to call. Rather we should simply be able to say:  
Thay vào đó, chúng tôi muốn các cộng tác viên trực tiếp của mình cung cấp tất cả các dịch vụ mà chúng tôi cần. Chúng ta không cần phải đi lang thang trong biểu đồ đối tượng của hệ thống, tìm kiếm phương thức mà chúng ta muốn gọi. Thay vào đó, chúng ta chỉ cần có thể nói:

myCollaborator.doSomething().

[**Java**](#_page_609_0)  
Java

[**J1:*Avoid Long Import Lists by Using Wildcards***](#_page_609_0)  
J1:Tránh danh sách nhập dài bằng cách sử dụng ký tự đại diện

If you use two or more classes from a package, then import the whole package with  
Nếu bạn sử dụng hai hoặc nhiều lớp từ một gói, thì hãy nhập toàn bộ gói với

import package.\*;

Long lists of imports are daunting to the reader. We don’t want to clutter up the tops of our modules with 80 lines of imports. Rather we want the imports to be a concise statement about which packages we collaborate with.  
Danh sách dài các hàng nhập khẩu gây khó khăn cho người đọc. Chúng tôi không muốn làm lộn xộn các đầu mô-đun của mình với 80 dòng nhập. Thay vào đó, chúng tôi muốn quá trình nhập là một tuyên bố ngắn gọn về những gói mà chúng tôi cộng tác.

Speciﬁc imports are hard dependencies, whereas wildcard imports are not. If you spe-ciﬁcally import a class, then that class *must* exist. But if you import a package with a wild-card, no particular classes need to exist. The import statement simply adds the package to the search path when hunting for names. So no true dependency is created by such imports, and they therefore serve to keep our modules less coupled.  
Nhập cụ thể là phụ thuộc cứng, trong khi nhập ký tự đại diện thì không. Nếu bạn nhập một lớp cụ thể, thì lớp đó phải tồn tại. Nhưng nếu bạn nhập một gói có ký tự đại diện, thì không cần tồn tại lớp cụ thể nào. Câu lệnh nhập chỉ cần thêm gói vào đường dẫn tìm kiếm khi tìm tên. Vì vậy, không có sự phụ thuộc thực sự nào được tạo ra bởi các lần nhập như vậy và do đó, chúng phục vụ để giữ cho các mô-đun của chúng ta ít bị ghép nối hơn.

There are times when the long list of speciﬁc imports can be useful. For example, if you are dealing with legacy code and you want to ﬁnd out what classes you need to build mocks and stubs for, you can walk down the list of speciﬁc imports to ﬁnd out the true qualiﬁed names of all those classes and then put the appropriate stubs in place. However, this use for speciﬁc imports is very rare. Furthermore, most modern IDEs will allow you to convert the wildcarded imports to a list of speciﬁc imports with a single command. So even in the legacy case it’s better to import wildcards.  
Đôi khi danh sách dài các mục nhập cụ thể có thể hữu ích. Ví dụ, nếu bạn đang xử lý mã kế thừa và bạn muốn tìm ra những lớp nào bạn cần xây dựng mô hình và sơ khai cho nó, bạn có thể đi xuống danh sách các mục nhập cụ thể để tìm ra tên đủ điều kiện thực sự của tất cả các lớp đó và sau đó đặt các sơ khai thích hợp tại chỗ. Tuy nhiên, việc sử dụng này cho nhập khẩu cụ thể là rất hiếm. Hơn nữa, hầu hết các IDE hiện đại sẽ cho phép bạn chuyển đổi các lần nhập ký tự đại diện thành danh sách các lần nhập cụ thể bằng một lệnh duy nhất. Vì vậy, ngay cả trong trường hợp kế thừa, tốt hơn là nhập các ký tự đại diện.

Wildcard imports can sometimes cause name conﬂicts and ambiguities. Two classes with the same name, but in different packages, will need to be speciﬁcally imported, or at least speciﬁcally qualiﬁed when used. This can be a nuisance but is rare enough that using wildcard imports is still generally better than speciﬁc imports.  
Nhập ký tự đại diện đôi khi có thể gây ra xung đột và mơ hồ về tên. Hai lớp có cùng tên, nhưng trong các gói khác nhau, sẽ cần phải được nhập cụ thể hoặc ít nhất là đủ điều kiện cụ thể khi sử dụng. Điều này có thể gây phiền toái nhưng hiếm khi sử dụng nhập ký tự đại diện nói chung vẫn tốt hơn so với nhập cụ thể.

[**J2: *Don’t Inherit Constants***](#_page_609_0)  
J2: Không kế thừa các hằng số

I have seen this several times and it always makes me grimace. A programmer puts some constants in an interface and then gains access to those constants by inheriting that inter-face. Take a look at the following code:  
Tôi đã thấy điều này nhiều lần và nó luôn khiến tôi nhăn nhó. Một lập trình viên đặt một số hằng số trong một giao diện và sau đó giành quyền truy cập vào các hằng số đó bằng cách kế thừa giao diện đó. Hãy xem đoạn mã sau:

public class HourlyEmployee extends Employee { private int tenthsWorked;

private double hourlyRate;

<https://www.facebook.com/groups/phattrienlaptrinh>

308 **Chapter 17: Smells and Heuristics**  
308 Chương 17: Mùi và Heuristic

public Money calculatePay() {

int straightTime = Math.min(tenthsWorked, TENTHS\_PER\_WEEK); int overTime = tenthsWorked - straightTime;

return new Money(

hourlyRate \* (tenthsWorked + OVERTIME\_RATE \* overTime) );

} ...

}

Where did the constants TENTHS\_PER\_WEEKand OVERTIME\_RATEcome from? They might have come from class Employee; so let’s take a look at that:  
Các hằng số TENTHS\_PER\_WEEK và OVERTIME\_RATE đến từ đâu? Họ có thể đến từ lớp Nhân viên; vì vậy hãy xem xét điều đó:

public abstract class Employee implements PayrollConstants { public abstract boolean isPayday();

public abstract Money calculatePay(); public abstract void deliverPay(Money pay);

}

Nope, not there. But then where? Look closely at class Employee. It implements PayrollConstants.  
Không, không có ở đó. Nhưng sau đó ở đâu? Nhìn kỹ vào lớp Nhân viên. Nó thực hiện PayrollConstants.

public interface PayrollConstants {

public static final int TENTHS\_PER\_WEEK = 400; public static final double OVERTIME\_RATE = 1.5;

}

This is a hideous practice! The constants are hidden at the top of the inheritance hierarchy. Ick! Don’t use inheritance as a way to cheat the scoping rules of the language. Use a static import instead.  
Đây là một thực hành ghê tởm! Các hằng số được ẩn ở trên cùng của hệ thống phân cấp thừa kế. Ốm! Đừng sử dụng tính kế thừa như một cách để gian lận các quy tắc phạm vi của ngôn ngữ. Thay vào đó, hãy sử dụng nhập tĩnh.

**import static PayrollConstants.\*;**

public class HourlyEmployee extends Employee { private int tenthsWorked;

private double hourlyRate;

public Money calculatePay() {

int straightTime = Math.min(tenthsWorked, TENTHS\_PER\_WEEK); int overTime = tenthsWorked - straightTime;

return new Money(

hourlyRate \* (tenthsWorked + OVERTIME\_RATE \* overTime) );

} ...

}

[**J3: *Constants versus Enums***](#_page_609_0)  
J3: Hằng số so với Enums

Now that enums have been added to the language (Java 5), use them! Don’t keep using the old trick of public static final ints. The meaning of ints can get lost. The meaning of enums cannot, because they belong to an enumeration that is named.  
Giờ đây, các enum đã được thêm vào ngôn ngữ (Java 5), hãy sử dụng chúng! Đừng tiếp tục sử dụng thủ thuật cũ về int tĩnh cuối cùng công khai. Ý nghĩa của ints có thể bị mất. Ý nghĩa của enums không thể, bởi vì chúng thuộc về một kiểu liệt kê được đặt tên.

What’s more, study the syntax for enums carefully. They can have methods and ﬁelds. This makes them very powerful tools that allow much more expression and ﬂexibility than ints. Consider this variation on the payroll code:  
Hơn nữa, hãy nghiên cứu kỹ cú pháp của enums. Họ có thể có các phương pháp và lĩnh vực. Điều này làm cho chúng trở thành những công cụ rất mạnh cho phép biểu đạt và linh hoạt hơn nhiều so với int. Hãy xem xét biến thể này trên mã bảng lương:

<https://www.facebook.com/groups/phattrienlaptrinh>

**Names** 309  
Tên 309

public class HourlyEmployee extends Employee { private int tenthsWorked;

**HourlyPayGrade grade;**

public Money calculatePay() {

int straightTime = Math.min(tenthsWorked, TENTHS\_PER\_WEEK); int overTime = tenthsWorked - straightTime;

return new Money(

**grade.rate()**\* (tenthsWorked + OVERTIME\_RATE \* overTime) );

} ...

}

public enum HourlyPayGrade { APPRENTICE {

public double rate() { return 1.0;

} },

LEUTENANT\_JOURNEYMAN { public double rate() {

return 1,2; }

}, JOURNEYMAN {

public double rate() { return 1.5;

} },

MASTER {

public double rate() { return 2.0;

} };

public abstract double rate(); }

[**Names**](#_page_609_0)  
tên

[**N1: *Choose Descriptive Names***](#_page_609_0)  
N1: Chọn Tên Mô Tả

Don’t be too quick to choose a name. Make sure the name is descriptive. Remember that meanings tend to drift as software evolves, so frequently reevaluate the appropriateness of the names you choose.  
Đừng quá vội vàng trong việc chọn một cái tên. Hãy chắc chắn rằng tên là mô tả. Hãy nhớ rằng ý nghĩa có xu hướng thay đổi khi phần mềm phát triển, vì vậy hãy thường xuyên đánh giá lại mức độ phù hợp của các tên bạn chọn.

This is not just a “feel-good” recommendation. Names in software are 90 percent of what make software readable. You need to take the time to choose them wisely and keep them relevant. Names are too important to treat carelessly.  
Đây không chỉ là một khuyến nghị "cảm thấy tốt". Tên trong phần mềm là 90 phần trăm những gì làm cho phần mềm có thể đọc được. Bạn cần dành thời gian để chọn chúng một cách khôn ngoan và giữ cho chúng phù hợp. Tên là quá quan trọng để đối xử bất cẩn.

Consider the code below. What does it do? If I show you the code with well-chosen names, it will make perfect sense to you, but like this it’s just a hodge-podge of symbols and magic numbers.  
Hãy xem xét mã dưới đây. Nó làm gì? Nếu tôi cho bạn xem mã với những cái tên được lựa chọn kỹ lưỡng, nó sẽ hoàn toàn có ý nghĩa đối với bạn, nhưng giống như thế này, nó chỉ là một mớ hỗn độn của các ký hiệu và con số ma thuật.

<https://www.facebook.com/groups/phattrienlaptrinh>

310 **Chapter 17: Smells and Heuristics**  
310 Chương 17: Mùi và Heuristic

public int x() { int q = 0; int z = 0;

for (int kk = 0; kk < 10; kk++) { if (l[z] == 10)

{

q += 10 + (l[z + 1] + l[z + 2]); z += 1;

}

else if (l[z] + l[z + 1] == 10) {

q += 10 + l[z + 2]; z += 2;

} else {

q += l[z] + l[z + 1]; z += 2;

} }

return q; }

Here is the code the way it should be written. This snippet is actually less complete than the one above. Yet you can infer immediately what it is trying to do, and you could very likely write the missing functions based on that inferred meaning. The magic num-bers are no longer magic, and the structure of the algorithm is compellingly descriptive.  
Đây là mã theo cách nó nên được viết. Đoạn mã này thực sự kém hoàn chỉnh hơn đoạn mã trên. Tuy nhiên, bạn có thể suy luận ngay lập tức nó đang cố gắng làm gì và rất có thể bạn có thể viết các hàm còn thiếu dựa trên ý nghĩa được suy luận đó. Những con số kỳ diệu không còn kỳ diệu nữa và cấu trúc của thuật toán được mô tả một cách thuyết phục.

public int score() { int score = 0; int frame = 0;

for (int frameNumber = 0; frameNumber < 10; frameNumber++) { if (isStrike(frame)) {

score += 10 + nextTwoBallsForStrike(frame); frame += 1;

} else if (isSpare(frame)) {

score += 10 + nextBallForSpare(frame); frame += 2;

} else {

score += twoBallsInFrame(frame); frame += 2;

} }

return score; }

The power of carefully chosen names is that they overload the structure of the code with description. That overloading sets the readers’ expectations about what the other functions in the module do. You can infer the implementation of isStrike()by looking at the code above. When you read the isStrikemethod, it will be “pretty much what you expected.”13  
Sức mạnh của những cái tên được lựa chọn cẩn thận là chúng làm quá tải cấu trúc của mã với mô tả. Sự quá tải đó đặt ra kỳ vọng của người đọc về chức năng của các chức năng khác trong mô-đun. Bạn có thể suy ra việc triển khai isStrike() bằng cách xem mã ở trên. Khi bạn đọc phương thức isStrike, nó sẽ “khá giống những gì bạn mong đợi.”13

private boolean isStrike(int frame) { return rolls[frame] == 10;

}

13. See Ward Cunningham’s quote on page 11.  
13. Xem phần trích dẫn của Ward Cunningham ở trang 11.

<https://www.facebook.com/groups/phattrienlaptrinh>

**Names** 311  
Tên 311

[**N2: *Choose Names at the Appropriate Level of Abstraction***](#_page_609_0)  
N2: Chọn tên ở mức độ trừu tượng phù hợp

Don’t pick names that communicate implementation; choose names the reﬂect the level of abstraction of the class or function you are working in. This is hard to do. Again, people are just too good at mixing levels of abstractions. Each time you make a pass over your code, you will likely ﬁnd some variable that is named at too low a level. You should take the opportunity to change those names when you ﬁnd them. Making code readable requires a dedication to continuous improvement. Consider the Modem interface below:  
Đừng chọn những cái tên truyền đạt việc triển khai; chọn tên phản ánh mức độ trừu tượng của lớp hoặc chức năng bạn đang làm việc. Điều này khó thực hiện. Một lần nữa, mọi người quá giỏi trong việc trộn lẫn các mức độ trừu tượng. Mỗi khi bạn chuyển qua mã của mình, bạn có thể sẽ tìm thấy một số biến được đặt tên ở mức quá thấp. Bạn nên tận dụng cơ hội để thay đổi những tên đó khi bạn tìm thấy chúng. Làm cho mã có thể đọc được đòi hỏi sự cống hiến để cải tiến liên tục. Xét giao diện Modem bên dưới:

public interface Modem {

boolean dial(String phoneNumber); boolean disconnect();

boolean send(char c); char recv();

String getConnectedPhoneNumber(); }

At ﬁrst this looks ﬁne. The functions all seem appropriate. Indeed, for many applications they are. But now consider an application in which some modems aren’t connected by dialling. Rather they are connected permanently by hard wiring them together (think of the cable modems that provide Internet access to most homes nowadays). Perhaps some are connected by sending a port number to a switch over a USB connection. Clearly the notion of phone numbers is at the wrong level of abstraction. A better naming strategy for this scenario might be:  
Lúc đầu, điều này có vẻ ổn. Tất cả các chức năng có vẻ phù hợp. Thật vậy, đối với nhiều ứng dụng, chúng là như vậy. Nhưng bây giờ hãy xem xét một ứng dụng trong đó một số modem không được kết nối bằng quay số. Thay vào đó, chúng được kết nối vĩnh viễn bằng cách nối dây cứng với nhau (nghĩ về modem cáp cung cấp truy cập Internet cho hầu hết các gia đình hiện nay). Có lẽ một số được kết nối bằng cách gửi số cổng tới một công tắc qua kết nối USB. Rõ ràng khái niệm về số điện thoại ở mức độ trừu tượng sai. Một chiến lược đặt tên tốt hơn cho kịch bản này có thể là:

public interface Modem {

boolean connect(String connectionLocator); boolean disconnect();

boolean send(char c); char recv();

String getConnectedLocator(); }

Now the names don’t make any commitments about phone numbers. They can still be used for phone numbers, or they could be used for any other kind of connection strategy.  
Bây giờ tên không đưa ra bất kỳ cam kết nào về số điện thoại. Chúng vẫn có thể được sử dụng cho các số điện thoại hoặc chúng có thể được sử dụng cho bất kỳ loại chiến lược kết nối nào khác.

[**N3: *Use Standard Nomenclature Where Possible***](#_page_609_0)  
N3: Sử dụng danh pháp tiêu chuẩn nếu có thể

Names are easier to understand if they are based on existing convention or usage. For exam-ple, if you are using the DECORATOR pattern, you should use the word Decorator in the names of the decorating classes. For example, AutoHangupModemDecoratormight be the name of a class that decorates a Modem with the ability to automatically hang up at the end of a session.  
Tên sẽ dễ hiểu hơn nếu chúng dựa trên quy ước hoặc cách sử dụng hiện có. Ví dụ, nếu bạn đang sử dụng mẫu DECORATOR, bạn nên sử dụng từ Decorator trong tên của các lớp trang trí. Ví dụ: AutoHangupModemDecoratorcó thể là tên của một lớp trang trí Modem với khả năng tự động treo khi kết thúc phiên.

Patterns are just one kind of standard. In Java, for example, functions that convert objects to string representations are often named toString. It is better to follow conven-tions like these than to invent your own.  
Các mẫu chỉ là một loại tiêu chuẩn. Ví dụ, trong Java, các hàm chuyển đổi các đối tượng thành biểu diễn chuỗi thường được đặt tên là toString. Tốt hơn là tuân theo những quy ước như thế này hơn là phát minh ra quy ước của riêng bạn.

Teams will often invent their own standard system of names for a particular project. Eric Evans refers to this as a *ubiquitous language* for the project.14 Your code should use  
Các nhóm thường sẽ phát minh ra hệ thống tên tiêu chuẩn của riêng họ cho một dự án cụ thể. Eric Evans coi đây là ngôn ngữ phổ biến cho dự án.14 Mã của bạn nên sử dụng

14. [DDD].

<https://www.facebook.com/groups/phattrienlaptrinh>

312 **Chapter 17: Smells and Heuristics**  
312 Chương 17: Mùi và Heuristic

the terms from this language extensively. In short, the more you can use names that are overloaded with special meanings that are relevant to your project, the easier it will be for readers to know what your code is talking about.  
các thuật ngữ từ ngôn ngữ này rộng rãi. Nói tóm lại, bạn càng có thể sử dụng những cái tên quá tải với những ý nghĩa đặc biệt có liên quan đến dự án của bạn, thì người đọc càng dễ dàng biết mã của bạn đang nói về điều gì.

[**N4: *Unambiguous Names***](#_page_609_0)  
N4: Tên rõ ràng

Choose names that make the workings of a function or variable unambiguous. Consider this example from FitNesse:  
Chọn những tên làm cho hoạt động của hàm hoặc biến trở nên rõ ràng. Xem xét ví dụ này từ FitNesse:

private String doRename() throws Exception {

if(refactorReferences) renameReferences();

renamePage();

pathToRename.removeNameFromEnd(); pathToRename.addNameToEnd(newName); return PathParser.render(pathToRename);

}

The name of this function does not say what the function does except in broad and vague terms. This is emphasized by the fact that there is a function named renamePageinside the function named doRename! What do the names tell you about the difference between the two functions? Nothing.  
Tên của chức năng này không nói chức năng đó làm gì ngoại trừ các thuật ngữ chung chung và mơ hồ. Điều này được nhấn mạnh bởi thực tế là có một hàm có tên renamePage bên trong hàm có tên doRename! Những cái tên cho bạn biết gì về sự khác biệt giữa hai chức năng? Không có gì.

A better name for that function is renamePageAndOptionallyAllReferences. This may seem long, and it is, but it’s only called from one place in the module, so it’s explanatory value outweighs the length.  
Tên tốt hơn cho hàm đó là renamePageAndOptionallyAllReferences. Điều này có vẻ dài, và đúng như vậy, nhưng nó chỉ được gọi từ một vị trí trong mô-đun, vì vậy giá trị giải thích của nó vượt xa độ dài.

[**N5: *Use Long Names for Long Scopes***](#_page_609_0)  
N5: Sử dụng tên dài cho phạm vi dài

The length of a name should be related to the length of the scope. You can use very short variable names for tiny scopes, but for big scopes you should use longer names.  
Độ dài của tên phải liên quan đến độ dài của phạm vi. Bạn có thể sử dụng tên biến rất ngắn cho phạm vi nhỏ, nhưng đối với phạm vi lớn, bạn nên sử dụng tên dài hơn.

Variable names like iand jare just ﬁne if their scope is ﬁve lines long. Consider this snippet from the old standard “Bowling Game”:  
Các tên biến như iand jare chỉ phù hợp nếu phạm vi của chúng dài năm dòng. Hãy xem đoạn trích này từ “Trò chơi Bowling” tiêu chuẩn cũ:

private void rollMany(int n, int pins) {

for (int i=0; i<n; i++) g.roll(pins);

}

This is perfectly clear and would be obfuscated if the variable iwere replaced with some-thing annoying like rollCount. On the other hand, variables and functions with short names lose their meaning over long distances. So the longer the scope of the name, the longer and more precise the name should be.  
Điều này hoàn toàn rõ ràng và sẽ bị xáo trộn nếu biến i được thay thế bằng thứ gì đó khó chịu như rollCount. Mặt khác, các biến và hàm có tên ngắn sẽ mất ý nghĩa trong một khoảng cách dài. Vì vậy, phạm vi của tên càng dài thì tên càng dài và chính xác.

[**N6:*Avoid Encodings***](#_page_609_0)  
N6:Tránh mã hóa

Names should not be encoded with type or scope information. Preﬁxes such as m\_or f are useless in today’s environments. Also project and/or subsystem encodings such as  
Tên không nên được mã hóa bằng thông tin loại hoặc phạm vi. Các tiền tố như m\_or f vô dụng trong môi trường ngày nay. Ngoài ra dự án và/hoặc mã hóa hệ thống con như

<https://www.facebook.com/groups/phattrienlaptrinh>

**Tests** 313  
Kiểm tra 313

vis\_(for visual imaging system) are distracting and redundant. Again, today’s environ-ments provide all that information without having to mangle the names. Keep your names free of Hungarian pollution.  
vis\_(đối với hệ thống hình ảnh trực quan) gây mất tập trung và dư thừa. Một lần nữa, môi trường ngày nay cung cấp tất cả thông tin đó mà không cần phải xáo trộn tên. Giữ tên của bạn không bị ô nhiễm Hungary.

[**N7: *Names Should Describe Side-Effects***](#_page_609_0)  
N7: Tên nên mô tả tác dụng phụ

Names should describe everything that a function, variable, or class is or does. Don’t hide side effects with a name. Don’t use a simple verb to describe a function that does more than just that simple action. For example, consider this code from TestNG:  
Tên nên mô tả mọi thứ mà một hàm, biến hoặc lớp là hoặc thực hiện. Đừng che giấu các tác dụng phụ bằng một cái tên. Đừng sử dụng một động từ đơn giản để mô tả một chức năng không chỉ là hành động đơn giản đó. Ví dụ: xem xét mã này từ TestNG:

public ObjectOutputStream getOos() throws IOException { if (m\_oos == null) {

m\_oos = new ObjectOutputStream(m\_socket.getOutputStream()); }

return m\_oos; }

This function does a bit more than get an “oos”; it creates the “oos” if it hasn’t been cre-ated already. Thus, a better name might be createOrReturnOos.  
Chức năng này thực hiện nhiều hơn một chút ngoài việc nhận được một "oos"; nó tạo ra tiếng “oos” nếu nó chưa được tạo. Vì vậy, một cái tên tốt hơn có thể là createOrReturnOos.

[**Tests**](#_page_685_0)  
bài kiểm tra

[**T1: *Insufﬁcient Tests***](#_page_685_0)  
T1: Kiểm tra không đầy đủ

How many tests should be in a test suite? Unfortunately, the metric many programmers use is “That seems like enough.” A test suite should test everything that could possibly break. The tests are insufﬁcient so long as there are conditions that have not been explored by the tests or calculations that have not been validated.  
Có bao nhiêu bài kiểm tra nên có trong một bộ kiểm tra? Thật không may, thước đo mà nhiều lập trình viên sử dụng là “Có vẻ như thế là đủ.” Một bộ thử nghiệm nên kiểm tra mọi thứ có thể bị hỏng. Các bài kiểm tra là không đủ chừng nào còn có những điều kiện chưa được khám phá bằng các bài kiểm tra hoặc các tính toán chưa được xác thực.

[**T2: *Use a Coverage Tool!***](#_page_685_0)  
T2: Sử dụng Công cụ Bảo hiểm!

Coverage tools reports gaps in your testing strategy. They make it easy to ﬁnd modules, classes, and functions that are insufﬁciently tested. Most IDEs give you a visual indication, marking lines that are covered in green and those that are uncovered in red. This makes it quick and easy to ﬁnd if or catch statements whose bodies haven’t been checked.  
Các công cụ bảo hiểm báo cáo những lỗ hổng trong chiến lược thử nghiệm của bạn. Chúng giúp dễ dàng tìm thấy các mô-đun, lớp và chức năng chưa được kiểm tra đầy đủ. Hầu hết các IDE cung cấp cho bạn một dấu hiệu trực quan, đánh dấu các dòng có màu xanh lục và những dòng không có màu đỏ. Điều này làm cho nó nhanh chóng và dễ dàng tìm thấy nếu hoặc nắm bắt các câu lệnh chưa được kiểm tra.

[**T3: *Don’t Skip Trivial Tests***](#_page_685_0)  
T3: Đừng bỏ qua các bài kiểm tra tầm thường

They are easy to write and their documentary value is higher than the cost to produce them.  
Chúng dễ viết và giá trị tài liệu của chúng cao hơn chi phí sản xuất chúng.

[**T4:*An Ignored Test Is a Question about an Ambiguity***](#_page_685_0)  
T4: Một bài kiểm tra bị bỏ qua là một câu hỏi về sự mơ hồ

Sometimes we are uncertain about a behavioral detail because the requirements are unclear. We can express our question about the requirements as a test that is commented out, or as a test that annotated with @Ignore. Which you choose depends upon whether the ambiguity is about something that would compile or not.  
Đôi khi chúng tôi không chắc chắn về một chi tiết hành vi vì các yêu cầu không rõ ràng. Chúng tôi có thể bày tỏ câu hỏi của mình về các yêu cầu dưới dạng bài kiểm tra được nhận xét hoặc dưới dạng bài kiểm tra được chú thích bằng @Ignore. Việc bạn chọn tùy thuộc vào việc sự mơ hồ có phải là về thứ gì đó sẽ biên dịch hay không.

<https://www.facebook.com/groups/phattrienlaptrinh>

314 **Chapter 17: Smells and Heuristics**  
314 Chương 17: Mùi và Heuristic

[**T5:*Test Boundary Conditions***](#_page_685_0)  
T5: Kiểm tra điều kiện biên

Take special care to test boundary conditions. We often get the middle of an algorithm right but misjudge the boundaries.  
Đặc biệt cẩn thận để kiểm tra các điều kiện biên. Chúng ta thường hiểu đúng phần giữa của một thuật toán nhưng đánh giá sai các ranh giới.

[**T6: *Exhaustively Test Near Bugs***](#_page_685_0)  
T6: Kiểm tra toàn diện các lỗi gần

Bugs tend to congregate. When you ﬁnd a bug in a function, it is wise to do an exhaustive test of that function. You’ll probably ﬁnd that the bug was not alone.  
Lỗi có xu hướng tụ tập. Khi bạn tìm thấy lỗi trong một chức năng, bạn nên tiến hành kiểm tra toàn diện chức năng đó. Bạn có thể sẽ thấy rằng lỗi không đơn độc.

[**T7: *Patterns of Failure Are Revealing***](#_page_685_0)  
T7: Các kiểu thất bại đang bộc lộ

Sometimes you can diagnose a problem by ﬁnding patterns in the way the test cases fail. This is another argument for making the test cases as complete as possible. Complete test cases, ordered in a reasonable way, expose patterns.  
Đôi khi bạn có thể chẩn đoán một vấn đề bằng cách tìm ra các mẫu trong cách các ca kiểm thử thất bại. Đây là một lập luận khác để làm cho các trường hợp thử nghiệm hoàn chỉnh nhất có thể. Hoàn thành các trường hợp thử nghiệm, sắp xếp theo cách hợp lý, hiển thị các mẫu.

As a simple example, suppose you noticed that all tests with an input larger than ﬁve characters failed? Or what if any test that passed a negative number into the second argu-ment of a function failed? Sometimes just seeing the pattern of red and green on the test report is enough to spark the “Aha!” that leads to the solution. Look back at page 267 to see an interesting example of this in the SerialDate example.  
Lấy một ví dụ đơn giản, giả sử bạn nhận thấy rằng tất cả các kiểm tra có đầu vào lớn hơn năm ký tự đều không thành công? Hoặc điều gì sẽ xảy ra nếu bất kỳ thử nghiệm nào chuyển một số âm vào đối số thứ hai của hàm không thành công? Đôi khi chỉ cần nhìn thấy mẫu màu đỏ và xanh lá cây trên báo cáo thử nghiệm là đủ để kích hoạt “Aha!” đó dẫn đến giải pháp. Nhìn lại trang 267 để xem một ví dụ thú vị về điều này trong ví dụ SerialDate.

[**T8:*Test Coverage Patterns Can Be Revealing***](#_page_685_0)  
T8: Các mẫu bao phủ thử nghiệm có thể được tiết lộ

Looking at the code that is or is not executed by the passing tests gives clues to why the failing tests fail.  
Nhìn vào mã được thực thi hoặc không được thực thi bởi các bài kiểm tra vượt qua sẽ đưa ra manh mối về lý do tại sao các bài kiểm tra không thành công lại thất bại.

[**T9:*Tests Should Be Fast***](#_page_685_0)  
T9: Các bài kiểm tra phải nhanh

A slow test is a test that won’t get run. When things get tight, it’s the slow tests that will be dropped from the suite. So *do what you must* to keep your tests fast.  
Một bài kiểm tra chậm là một bài kiểm tra sẽ không được chạy. Khi mọi thứ trở nên căng thẳng, các bài kiểm tra chậm sẽ bị loại bỏ khỏi bộ ứng dụng. Vì vậy, hãy làm những gì bạn phải làm để giữ cho bài kiểm tra của bạn nhanh chóng.

[**Conclusion**](#_page_685_0)  
Phần kết luận

This list of heuristics and smells could hardly be said to be complete. Indeed, I’m not sure that such a list can *ever* be complete. But perhaps completeness should not be the goal, because what this list *does* do is imply a value system.  
Khó có thể nói là đầy đủ danh sách các kinh nghiệm và mùi vị này. Thật vậy, tôi không chắc rằng một danh sách như vậy có thể đầy đủ hay không. Nhưng có lẽ mục tiêu không phải là sự hoàn thiện, bởi vì điều mà danh sách này làm là ám chỉ một hệ thống giá trị.

Indeed, that value system has been the goal, and the topic, of this book. Clean code is not written by following a set of rules. You don’t become a software craftsman by learn-ing a list of heuristics. Professionalism and craftsmanship come from values that drive disciplines.  
Thật vậy, hệ thống giá trị đó là mục tiêu và chủ đề của cuốn sách này. Mã sạch không được viết bằng cách tuân theo một bộ quy tắc. Bạn không trở thành một nghệ nhân phần mềm bằng cách học một danh sách các phương pháp phỏng đoán. Tính chuyên nghiệp và tay nghề thủ công đến từ các giá trị thúc đẩy kỷ luật.

<https://www.facebook.com/groups/phattrienlaptrinh>

**Bibliography** 315  
Thư mục 315

[**Bibliography**](#_page_685_0)  
Thư mục

**[Refactoring]:** *Refactoring: Improving the Design of Existing Code*, Martin Fowler et al., Addison-Wesley, 1999.

**[PRAG]:** *The Pragmatic Programmer*, Andrew Hunt, Dave Thomas, Addison-Wesley, 2000.

**[GOF]:** *Design Patterns: Elements of Reusable Object Oriented Software*, Gamma et al., Addison-Wesley, 1996.

**[Beck97]:** *Smalltalk Best Practice Patterns*, Kent Beck, Prentice Hall, 1997.

**[Beck07]:** *Implementation Patterns*, Kent Beck, Addison-Wesley, 2008.

**[PPP]:** *Agile Software Development: Principles, Patterns, and Practices*, Robert C. Martin, Prentice Hall, 2002.

**[DDD]:** *Domain Driven Design*, Eric Evans, Addison-Wesley, 2003.

<https://www.facebook.com/groups/phattrienlaptrinh>

*This page intentionally left blank*

<https://www.facebook.com/groups/phattrienlaptrinh>

[**Appendix A**](#_page_685_0)  
Phụ lục A

[**Concurrency II**](#_page_685_0)  
đồng thời II

by Brett L. Schuchert

This appendix supports and ampliﬁes the *Concurrency* chapter on page 177. It is written as a series of independent topics and you can generally read them in any order. There is some duplication between sections to allow for such reading.  
Phụ lục này hỗ trợ và mở rộng chương Đồng thời ở trang 177. Nó được viết dưới dạng một loạt các chủ đề độc lập và bạn thường có thể đọc chúng theo bất kỳ thứ tự nào. Có một số trùng lặp giữa các phần để cho phép đọc như vậy.

[**Client/Server Example**](#_page_685_0)  
Ví dụ máy khách/máy chủ

Imagine a simple client/server application. A server sits and waits listening on a socket for a client to connect. A client connects and sends a request.  
Hãy tưởng tượng một ứng dụng máy khách/máy chủ đơn giản. Một máy chủ ngồi và chờ lắng nghe trên một ổ cắm để máy khách kết nối. Một khách hàng kết nối và gửi yêu cầu.

[**The Server**](#_page_685_0)  
máy chủ

Here is a simpliﬁed version of a server application. Full source for this example is avail-able starting on page 343, *Client/Server Nonthreaded*.  
Đây là một phiên bản đơn giản hóa của một ứng dụng máy chủ. Nguồn đầy đủ cho ví dụ này có sẵn bắt đầu từ trang 343, Client/Server Nonthreaded.

ServerSocket serverSocket = new ServerSocket(8009);

while (keepProcessing) { try {

Socket socket = serverSocket.accept(); process(socket);

} catch (Exception e) { handle(e);

} }

317

<https://www.facebook.com/groups/phattrienlaptrinh>

318 **Appendix A: Concurrency II**  
318 Phụ lục A: Tương tranh II

This simple application waits for a connection, processes an incoming message, and then again waits for the next client request to come in. Here’s client code that connects to this server:  
Ứng dụng đơn giản này chờ kết nối, xử lý một tin nhắn đến và sau đó lại đợi yêu cầu tiếp theo của máy khách đến. Đây là mã máy khách kết nối với máy chủ này:

private void connectSendReceive(int i) { try {

Socket socket = new Socket("localhost", PORT); MessageUtils.sendMessage(socket, Integer.toString(i)); MessageUtils.getMessage(socket);

socket.close();

} catch (Exception e) { e.printStackTrace();

} }

How well does this client/server pair perform? How can we formally describe that perfor-mance? Here’s a test that asserts that the performance is “acceptable”:  
Cặp máy khách/máy chủ này hoạt động tốt như thế nào? Làm thế nào chúng ta có thể chính thức mô tả hiệu suất đó? Đây là một bài kiểm tra khẳng định rằng hiệu suất là "chấp nhận được":

@Test(timeout = 10000)

public void shouldRunInUnder10Seconds() throws Exception { Thread[] threads = createThreads(); startAllThreadsw(threads); waitForAllThreadsToFinish(threads);

}

The setup is left out to keep the example simple (see “ClientTest.java” on page 344). This test asserts that it should complete within 10,000 milliseconds.  
Việc thiết lập được bỏ qua để giữ cho ví dụ đơn giản (xem “ClientTest.java” trên trang 344). Thử nghiệm này khẳng định rằng nó sẽ hoàn thành trong vòng 10.000 mili giây.

This is a classic example of validating the throughput of a system. This system should complete a series of client requests in ten seconds. So long as the server can process each individual client request in time, the test will pass.  
Đây là một ví dụ cổ điển về xác thực thông lượng của một hệ thống. Hệ thống này sẽ hoàn thành một loạt yêu cầu của khách hàng trong mười giây. Miễn là máy chủ có thể xử lý kịp thời từng yêu cầu của khách hàng, bài kiểm tra sẽ vượt qua.

What happens if the test fails? Short of developing some kind of event polling loop, there is not much to do within a single thread that will make this code any faster. Will using multiple threads solve the problem? It might, but we need to know where the time is being spent. There are two possibilities:  
Điều gì xảy ra nếu thử nghiệm không thành công? Không cần phát triển một số loại vòng lặp bỏ phiếu sự kiện, không có nhiều việc phải làm trong một luồng đơn lẻ sẽ làm cho mã này nhanh hơn. Sẽ sử dụng nhiều chủ đề giải quyết vấn đề? Có thể, nhưng chúng ta cần biết thời gian đang được sử dụng vào đâu. Có hai khả năng:

**•** I/O—using a socket, connecting to a database, waiting for virtual memory swapping, and so on.  
• I/O—sử dụng ổ cắm, kết nối với cơ sở dữ liệu, chờ trao đổi bộ nhớ ảo, v.v.

**•** Processor—numerical calculations, regular expression processing, garbage collection, and so on.  
• Bộ xử lý—tính toán số, xử lý biểu thức chính quy, thu gom rác, v.v.

Systems typically have some of each, but for a given operation one tends to dominate. If the code is processor bound, more processing hardware can improve throughput, making our test pass. But there are only so many CPU cycles available, so adding threads to a processor-bound problem will not make it go faster.  
Các hệ thống thường có một số trong số đó, nhưng đối với một hoạt động nhất định, một hoạt động có xu hướng chiếm ưu thế. Nếu mã bị ràng buộc bởi bộ xử lý, thì phần cứng xử lý nhiều hơn có thể cải thiện thông lượng, khiến bài kiểm tra của chúng tôi vượt qua. Nhưng chỉ có rất nhiều chu kỳ CPU khả dụng, do đó, việc thêm các luồng vào một vấn đề liên quan đến bộ xử lý sẽ không làm cho nó chạy nhanh hơn.

On the other hand, if the process is I/O bound, then concurrency can increase efﬁ-ciency. When one part of the system is waiting for I/O, another part can use that wait time to process something else, making more effective use of the available CPU.  
Mặt khác, nếu quá trình bị ràng buộc I/O, thì đồng thời có thể tăng hiệu quả. Khi một phần của hệ thống đang chờ I/O, phần khác có thể sử dụng thời gian chờ đó để xử lý thứ khác, giúp sử dụng hiệu quả hơn CPU có sẵn.

<https://www.facebook.com/groups/phattrienlaptrinh>

**Client/Server Example** 319  
Máy khách/Máy chủ Ví dụ 319

[**Adding Threading**](#_page_685_0)  
Thêm luồng

Assume for the moment that the performance test fails. How can we improve the through-put so that the performance test passes? If the process method of the server is I/O bound, then here is one way to make the server use threads (just change the processMessage):  
Tạm thời giả sử rằng bài kiểm tra hiệu suất không thành công. Làm cách nào chúng tôi có thể cải thiện thông lượng để bài kiểm tra hiệu suất vượt qua? Nếu phương thức xử lý của máy chủ bị ràng buộc I/O, thì đây là một cách để làm cho máy chủ sử dụng các luồng (chỉ cần thay đổi processMessage):

void process(final Socket socket) { if (socket == null)

return;

Runnable clientHandler = new Runnable() { public void run() {

try {

String message = MessageUtils.getMessage(socket); MessageUtils.sendMessage(socket, "Processed: " + message); closeIgnoringException(socket);

} catch (Exception e) { e.printStackTrace();

} }

};

Thread clientConnection = new Thread(clientHandler); clientConnection.start();

}

Assume that this change causes the test to pass;1 the code is complete, correct?  
Giả sử rằng thay đổi này khiến bài kiểm tra vượt qua;1 mã đã hoàn tất, đúng không?

[**Server Observations**](#_page_685_0)  
Quan sát máy chủ

The updated server completes the test successfully in just over one second. Unfortunately, this solution is a bit naive and introduces some new problems.  
Máy chủ được cập nhật hoàn thành bài kiểm tra thành công chỉ trong hơn một giây. Thật không may, giải pháp này hơi ngây thơ và đưa ra một số vấn đề mới.

How many threads might our server create? The code sets no limit, so the we could feasibly hit the limit imposed by the Java Virtual Machine (JVM). For many simple sys-tems this may sufﬁce. But what if the system is meant to support many users on the public net? If too many users connect at the same time, the system might grind to a halt.  
Máy chủ của chúng tôi có thể tạo bao nhiêu chủ đề? Mã này không đặt giới hạn, vì vậy chúng tôi có thể đạt được giới hạn do Máy ảo Java (JVM) áp đặt một cách khả thi. Đối với nhiều hệ thống đơn giản, điều này có thể đủ. Nhưng nếu hệ thống nhằm hỗ trợ nhiều người dùng trên mạng công cộng thì sao? Nếu quá nhiều người dùng kết nối cùng một lúc, hệ thống có thể bị đình trệ.

But set the behavioral problem aside for the moment. The solution shown has prob-lems of cleanliness and structure. How many responsibilities does the server code have?  
Nhưng hãy đặt vấn đề hành vi sang một bên vào lúc này. Giải pháp hiển thị có vấn đề về độ sạch và cấu trúc. Mã máy chủ có bao nhiêu trách nhiệm?

**•** Socket connection management **•** Client processing  
• Quản lý kết nối ổ cắm • Xử lý máy khách

**•** Threading policy  
• Chính sách phân luồng

**•** Server shutdown policy  
• Chính sách tắt máy chủ

Unfortunately, all these responsibilities live in the process function. In addition, the code crosses many different levels of abstraction. So, small as the process function is, it needs to be repartitioned.  
Thật không may, tất cả những trách nhiệm này nằm trong chức năng quy trình. Ngoài ra, mã vượt qua nhiều mức độ trừu tượng khác nhau. Vì vậy, chức năng quá trình nhỏ như vậy, nó cần được phân vùng lại.

1. You can verify that for yourself by trying out the before and after code. Review the nonthreaded code starting on page 343. Review the threaded code starting on page 346.  
1. Bạn có thể tự xác minh điều đó bằng cách thử mã trước và sau. Xem lại mã không phân luồng bắt đầu từ trang 343. Xem lại mã phân luồng bắt đầu từ trang 346.

<https://www.facebook.com/groups/phattrienlaptrinh>

320 **Appendix A: Concurrency II**  
320 Phụ lục A: Tương tranh II

The server has several reasons to change; therefore it violates the Single Responsibility Principle. To keep concurrent systems clean, thread management should be kept to a few, well-controlled places. What’s more, any code that manages threads should do nothing other than thread management. Why? If for no other reason than that tracking down con-currency issues is hard enough without having to unwind other nonconcurrency issues at the same time.  
Máy chủ có một số lý do để thay đổi; do đó nó vi phạm Nguyên tắc Trách nhiệm duy nhất. Để giữ cho các hệ thống đồng thời sạch sẽ, việc quản lý luồng nên được giữ ở một số nơi được kiểm soát tốt. Hơn nữa, bất kỳ mã nào quản lý luồng không được làm gì khác ngoài quản lý luồng. Tại sao? Nếu không vì lý do nào khác thì việc theo dõi các vấn đề đồng tiền là đủ khó mà không cần phải giải quyết các vấn đề không đồng thời khác cùng một lúc.

If we create a separate class for each of the responsibilities listed above, including the thread management responsibility, then when we change the thread management strategy, the change will impact less overall code and will not pollute the other responsibilities. This also makes it much easier to test all the other responsibilities without having to worry about threading. Here is an updated version that does just that:  
Nếu chúng ta tạo một lớp riêng biệt cho từng trách nhiệm được liệt kê ở trên, bao gồm cả trách nhiệm quản lý luồng, thì khi chúng ta thay đổi chiến lược quản lý luồng, thay đổi đó sẽ ít ảnh hưởng đến mã tổng thể hơn và sẽ không làm ô nhiễm các trách nhiệm khác. Điều này cũng làm cho việc kiểm tra tất cả các trách nhiệm khác trở nên dễ dàng hơn nhiều mà không phải lo lắng về phân luồng. Đây là một phiên bản cập nhật thực hiện điều đó:

public void run() {

while (keepProcessing) { try {

ClientConnection clientConnection = connectionManager.awaitClient(); ClientRequestProcessor requestProcessor

= new ClientRequestProcessor(clientConnection); clientScheduler.schedule(requestProcessor);

} catch (Exception e) { e.printStackTrace();

}

} connectionManager.shutdown();

}

This now focuses all things thread-related into one place, clientScheduler. If there are concurrency problems, there is just one place to look:  
Điều này hiện tập trung tất cả những thứ liên quan đến luồng vào một nơi, clientScheduler. Nếu có vấn đề về đồng thời, chỉ có một nơi để xem xét:

public interface ClientScheduler {

void schedule(ClientRequestProcessor requestProcessor); }

The current policy is easy to implement:  
Chính sách hiện tại rất dễ thực hiện:

public class ThreadPerRequestScheduler implements ClientScheduler { public void schedule(final ClientRequestProcessor requestProcessor) {

Runnable runnable = new Runnable() { public void run() {

requestProcessor.process(); }

};

Thread thread = new Thread(runnable); thread.start();

} }

Having isolated all the thread management into a single place, it is much easier to change the way we control threads. For example, moving to the Java 5 Executor framework involves writing a new class and plugging it in (Listing A-1).  
Sau khi cô lập tất cả việc quản lý luồng vào một nơi duy nhất, việc thay đổi cách chúng ta kiểm soát luồng sẽ dễ dàng hơn nhiều. Ví dụ: chuyển sang khung công tác Java 5 Executor liên quan đến việc viết một lớp mới và cắm nó vào (Liệt kê A-1).

<https://www.facebook.com/groups/phattrienlaptrinh>

**Possible Paths of Execution** 321  
Các đường thực hiện khả thi 321

**Listing A-1 ExecutorClientScheduler.java**  
Liệt kê A-1 ExecutorClientScheduler.java

import java.util.concurrent.Executor; import java.util.concurrent.Executors;

public class ExecutorClientScheduler implements ClientScheduler { Executor executor;

public ExecutorClientScheduler(int availableThreads) { executor = Executors.newFixedThreadPool(availableThreads);

}

public void schedule(final ClientRequestProcessor requestProcessor) { Runnable runnable = new Runnable() {

public void run() { requestProcessor.process();

} };

executor.execute(runnable); }

}

[**Conclusion**](#_page_685_0)  
Phần kết luận

Introducing concurrency in this particular example demonstrates a way to improve the throughput of a system and one way of validating that throughput through a testing frame-work. Focusing all concurrency code into a small number of classes is an example of applying the Single Responsibility Principle. In the case of concurrent programming, this becomes especially important because of its complexity.  
Giới thiệu đồng thời trong ví dụ cụ thể này minh họa một cách để cải thiện thông lượng của hệ thống và một cách để xác thực thông lượng đó thông qua một khung làm việc thử nghiệm. Tập trung tất cả mã đồng thời vào một số lượng nhỏ các lớp là một ví dụ về việc áp dụng Nguyên tắc Trách nhiệm Đơn lẻ. Trong trường hợp lập trình đồng thời, điều này trở nên đặc biệt quan trọng vì tính phức tạp của nó.

[**Possible Paths of Execution**](#_page_685_0)  
Các đường thực hiện có thể

Review the method incrementValue, a one-line Java method with no looping or branching:  
Xem lại phương thức incrementValue, một phương thức Java một dòng không có vòng lặp hoặc phân nhánh:

public class IdGenerator { int lastIdUsed;

public int incrementValue() { return ++lastIdUsed;

} }

Ignore integer overﬂow and assume that only one thread has access to a single instance of IdGenerator. In this case there is a single path of execution and a single guaranteed result:  
Bỏ qua lỗi tràn số nguyên và giả sử rằng chỉ một luồng có quyền truy cập vào một phiên bản duy nhất của IdGenerator. Trong trường hợp này, có một đường dẫn thực thi duy nhất và một kết quả được đảm bảo duy nhất:

**•** The value returned is equal to the value of lastIdUsed, both of which are one greater than just before calling the method.  
• Giá trị được trả về bằng với giá trị của lastIdUsed, cả hai đều lớn hơn một giá trị ngay trước khi gọi phương thức.

<https://www.facebook.com/groups/phattrienlaptrinh>

322 **Appendix A: Concurrency II**  
322 Phụ lục A: Tương tranh II

What happens if we use two threads and leave the method unchanged? What are the possible outcomes if each thread calls incrementValue once? How many possible paths of execution are there? First, the outcomes (assume lastIdUsed starts with a value of 93):  
Điều gì xảy ra nếu chúng ta sử dụng hai luồng và giữ nguyên phương thức? Kết quả có thể xảy ra nếu mỗi chuỗi gọi số incrementValue một lần là gì? Có bao nhiêu con đường thực hiện có thể? Đầu tiên, kết quả (giả sử lastIdUsed bắt đầu với giá trị là 93):

**•** Thread 1 gets the value of 94, thread 2 gets the value of 95, and lastIdUsed is now 95. **•** Thread 1 gets the value of 95, thread 2 gets the value of 94, and lastIdUsed is now 95. **•** Thread 1 gets the value of 94, thread 2 gets the value of 94, and lastIdUsed is now 94.  
• Chủ đề 1 nhận giá trị 94, chủ đề 2 nhận giá trị 95 và lastIdUsed hiện là 95. • Chủ đề 1 nhận giá trị 95, chủ đề 2 nhận giá trị 94 và lastIdUsed hiện là 95. • Chủ đề 1 nhận giá trị là 94, luồng 2 nhận giá trị là 94 và lastIdUsed hiện là 94.

The ﬁnal result, while surprising, is possible. To see how these different results are possi-ble, we need to understand the number of possible paths of execution and how the Java Virtual Machine executes them.  
Kết quả cuối cùng, trong khi đáng ngạc nhiên, là có thể. Để xem những kết quả khác nhau này có thể xảy ra như thế nào, chúng ta cần hiểu số lượng đường dẫn thực thi có thể có và cách Máy ảo Java thực thi chúng.

[**Number of Paths**](#_page_685_0)  
Số đường dẫn

To calculate the number of possible execution paths, we’ll start with the generated byte-code. The one line of java (return ++lastIdUsed;) becomes eight byte-code instructions. It is possible for the two threads to interleave the execution of these eight instructions the way a card dealer interleaves cards as he shufﬂes a deck.2 Even with only eight cards in each hand, there are a remarkable number of shufﬂed outcomes.  
Để tính toán số lượng đường dẫn thực thi có thể, chúng ta sẽ bắt đầu với mã byte được tạo. Một dòng java (return ++lastIdUsed;) trở thành tám hướng dẫn mã byte. Hai luồng có thể xen kẽ việc thực hiện tám hướng dẫn này giống như cách một người chia bài xen kẽ các quân bài khi anh ta xáo trộn một cỗ bài.2 Ngay cả khi mỗi tay chỉ có tám quân bài, vẫn có một số lượng đáng kể các kết quả được xáo trộn.

For this simple case of *N* instructions in a sequence, no looping or conditionals, and *T* threads, the total number of possible execution paths is equal to --------------*N*!

**Calculating the Possible Orderings**  
Tính toán các đơn đặt hàng có thể

This comes from an email from Uncle Bob to Brett:  
Điều này xuất phát từ một email của chú Bob gửi cho Brett:

With *N* steps and *T* threads there are *T* \**N* total steps. Prior to each step there is a context switch that chooses between the *T* threads. Each path can thus be represented as a string of digits denoting the context switches. Given steps A and B and threads 1 and 2, the six possible paths are 1122, 1212, 1221, 2112, 2121, and 2211. Or, in terms of steps it is A1B1A2B2, A1A2B1B2, A1A2B2B1, A2A1B1B2, A2A1B2B1, and A2B2A1B1. For three threads the sequence is 112233, 112323, 113223, 113232, 112233, 121233, 121323, 121332, 123132, 123123, . . . .  
Với N bước và T luồng, có tổng số T \* N bước. Trước mỗi bước, có một công tắc ngữ cảnh chọn giữa các luồng T. Do đó, mỗi đường dẫn có thể được biểu diễn dưới dạng một chuỗi các chữ số biểu thị các chuyển ngữ cảnh. Với các bước A và B và các luồng 1 và 2, sáu đường dẫn có thể là 1122, 1212, 1221, 2112, 2121 và 2211. Hoặc, theo các bước, đó là A1B1A2B2, A1A2B1B2, A1A2B2B1, A2A1B1B2, A2A1B2B1 và A2B2A1B1. Đối với ba luồng, chuỗi là 112233, 112323, 113223, 113232, 112233, 121233, 121323, 121332, 123132, 123123, . . . .

One characteristic of these strings is that there must always be *N* instances of each *T*. So the string 111111 is invalid because it has six instances of 1 and zero instances of 2 and 3.  
Một đặc điểm của các chuỗi này là phải luôn có N phiên bản của mỗi T. Vì vậy, chuỗi 111111 không hợp lệ vì nó có sáu phiên bản 1 và không có phiên bản 2 và 3.

2. This is a bit of a simpliﬁcation. However, for the purpose of this discussion, we can use this simplifying model.  
2. Đây là một chút đơn giản hóa. Tuy nhiên, với mục đích của cuộc thảo luận này, chúng ta có thể sử dụng mô hình đơn giản hóa này.

<https://www.facebook.com/groups/phattrienlaptrinh>

**Possible Paths of Execution** 323  
Các đường thực hiện khả thi 323

**Calculating the Possible Orderings (continued)  
Tính toán các thứ tự có thể có (tiếp theo)**

So we want the permutations of *N* 1’s, *N* 2’s, . . . and *N T’*s. This is really just the permutations of *N* \* *T* things taken *N* \* *T* at a time, which is (*N* \* *T*)!, but with all the duplicates removed. So the trick is to count the duplicates and subtract that from (*N* \* *T* )!.  
Vì vậy, chúng tôi muốn các hoán vị của N 1, N 2, . . . và N T's. Đây thực sự chỉ là hoán vị của N \* T thứ được lấy N \* T tại một thời điểm, tức là (N \* T)!, nhưng đã loại bỏ tất cả các bản sao. Vì vậy, mẹo là đếm số lần trùng lặp và trừ đi số đó khỏi (N \* T )!.

Given two steps and two threads, how many duplicates are there? Each four-digit string has two 1s and two 2s. Each of those pairs could be swapped without changing the sense of the string. You could swap the 1s or the 2s both, or neither. So there are four isomorphs for each string, which means that there are three duplicates. So three out of four of the options are duplicates; alternatively one of four of the permutations are NOT dupli-cates. 4! \* .25 = 6. So this reasoning seems to work.  
Cho hai bước và hai luồng, có bao nhiêu bản sao? Mỗi chuỗi bốn chữ số có hai số 1 và hai số 2. Mỗi cặp trong số đó có thể được hoán đổi mà không làm thay đổi ý nghĩa của chuỗi. Bạn có thể hoán đổi cả 1 hoặc 2 hoặc không. Vì vậy, có bốn đẳng cấu cho mỗi chuỗi, có nghĩa là có ba bản sao. Vì vậy, ba trong số bốn tùy chọn là trùng lặp; cách khác, một trong bốn hoán vị KHÔNG trùng lặp. 4! \* .25 = 6. Vì vậy, suy luận này có vẻ hiệu quả.

How many duplicates are there? In the case where *N* = 2 and *T* = 2, I could swap the 1s, the 2s, or both. In the case where *N* = 2 and *T* = 3, I could swap the 1s, the 2s, the 3s, 1s and 2s, 1s and 3s, or 2s and 3s. Swap-ping is just the permutations of *N*. Let’s say there are *P* permutations of *N*. The number of different ways to arrange those permutations are *P*\*\**T*.  
Có bao nhiêu bản sao? Trong trường hợp N = 2 và T = 2, tôi có thể đổi chỗ 1, 2 hoặc cả hai. Trong trường hợp N = 2 và T = 3, tôi có thể hoán đổi các số 1, 2, 3, 1 và 2, 1 và 3 hoặc 2 và 3. Hoán vị chỉ là hoán vị của N. Giả sử có P hoán vị của N. Số cách khác nhau để sắp xếp các hoán vị đó là P\*\*T.

So the number of possible isomorphs is *N*!\*\**T*. And so the number of paths is (*T*\**N*)!/(*N*!\*\**T*). Again, in our *T* = 2, *N* = 2 case we get 6 (24/4).  
Vì vậy, số lượng các đồng phân có thể có là N!\*\*T. Và do đó, số đường dẫn là (T\*N)!/(N!\*\*T). Một lần nữa, trong trường hợp T = 2, N = 2 của chúng tôi, chúng tôi nhận được 6 (24/4).

For *N* = 2 and *T* = 3 we get 720/8 = 90. For *N* = 3 and *T* = 3 we get 9!/6^3 = 1680.  
Với N = 2 và T = 3, chúng ta có 720/8 = 90. Với N = 3 và T = 3, chúng ta có 9!/6^3 = 1680.

For our simple case of one line of Java code, which equates to eight lines of byte-code and two threads, the total number of possible paths of execution is 12,870. If the type of lastIdUsed is a long, then every read/write becomes two operations instead of one, and the number of possible orderings becomes 2,704,156.  
Đối với trường hợp đơn giản của chúng tôi về một dòng mã Java, tương đương với tám dòng mã byte và hai luồng, tổng số đường dẫn thực thi có thể có là 12.870. Nếu loại lastIdUsed dài, thì mỗi lần đọc/ghi sẽ trở thành hai thao tác thay vì một và số lần đặt hàng có thể trở thành 2.704.156.

What happens if we make one change to this method?  
Điều gì xảy ra nếu chúng ta thực hiện một thay đổi đối với phương pháp này?

public **synchronized** void incrementValue() { ++lastIdUsed;

}

The number of possible execution pathways becomes two for two threads and N! in the general case.  
Số lượng đường dẫn thực hiện có thể trở thành hai cho hai luồng và N! trong trường hợp chung.

[**Digging Deeper**](#_page_685_0)  
Đào sâu hơn

What about the surprising result that two threads could both call the method once (before we added synchronized) and get the same numeric result? How is that possible? First things ﬁrst.  
Còn kết quả đáng ngạc nhiên là hai luồng có thể gọi phương thức một lần (trước khi chúng tôi thêm đồng bộ hóa) và nhận được cùng một kết quả số thì sao? Làm thế nào là có thể? Điều đầu tiên đầu tiên.

What is an atomic operation? We can deﬁne an atomic operation as any operation that is uninterruptable. For example, in the following code, line 5, where 0 is assigned to lastid, is atomic because according to the Java Memory model, assignment to a 32-bit value is uninterruptable.  
một hoạt động nguyên tử là gì? Chúng ta có thể định nghĩa một hoạt động nguyên tử là bất kỳ hoạt động nào không bị gián đoạn. Ví dụ: trong đoạn mã sau, dòng 5, trong đó 0 được gán cho lastid, là nguyên tử vì theo mô hình Bộ nhớ Java, việc gán cho giá trị 32 bit là không thể bị gián đoạn.

<https://www.facebook.com/groups/phattrienlaptrinh>

324 **Appendix A: Concurrency II**  
324 Phụ lục A: Tương tranh II

01: public class Example { 02: int lastId;

03:

04: public void resetId() { 05: value = 0;

06: } 07:

08: public int getNextId() { 09: ++value;

10: } 11:}

What happens if we change type of lastId from int to long? Is line 5 still atomic? Not according to the JVM speciﬁcation. It could be atomic on a particular processor, but according to the JVM speciﬁcation, assignment to any 64-bit value requires two 32-bit assignments. This means that between the ﬁrst 32-bit assignment and the second 32-bit assignment, some other thread could sneak in and change one of the values.  
Điều gì xảy ra nếu chúng ta thay đổi loại lastId từ int thành long? Là dòng 5 vẫn còn nguyên tử? Không theo đặc tả JVM. Nó có thể là nguyên tử trên một bộ xử lý cụ thể, nhưng theo đặc tả JVM, việc gán cho bất kỳ giá trị 64-bit nào cũng yêu cầu hai phép gán 32-bit. Điều này có nghĩa là giữa lần gán 32 bit đầu tiên và lần gán 32 bit thứ hai, một số luồng khác có thể lẻn vào và thay đổi một trong các giá trị.

What about the pre-increment operator, ++, on line 9? The pre-increment operator can be interrupted, so it is not atomic. To understand, let’s review the byte-code of both of these methods in detail.  
Còn toán tử tăng trước, ++, trên dòng 9 thì sao? Toán tử tăng trước có thể bị gián đoạn, vì vậy nó không phải là nguyên tử. Để hiểu rõ, hãy xem xét chi tiết mã byte của cả hai phương pháp này.

Before we go any further, here are three deﬁnitions that will be important:  
Trước khi chúng ta tiếp tục, đây là ba định nghĩa sẽ rất quan trọng:

**•** Frame—Every method invocation requires a frame. The frame includes the return address, any parameters passed into the method and the local variables deﬁned in the method. This is a standard technique used to deﬁne a call stack, which is used by modern languages to allow for basic function/method invocation and to allow for recursive invocation.  
• Frame—Mọi lời gọi phương thức đều yêu cầu một frame. Khung bao gồm địa chỉ trả về, bất kỳ tham số nào được truyền vào phương thức và các biến cục bộ được xác định trong phương thức. Đây là một kỹ thuật tiêu chuẩn được sử dụng để xác định ngăn xếp cuộc gọi, được sử dụng bởi các ngôn ngữ hiện đại để cho phép gọi hàm/phương thức cơ bản và cho phép gọi đệ quy.

**•** Local variable—Any variables deﬁned in the scope of the method. All nonstatic meth-ods have at least one variable, **this**, which represents the current object, the object that received the most recent message (in the current thread), which caused the method invocation.  
• Biến cục bộ—Bất kỳ biến nào được định nghĩa trong phạm vi của phương thức. Tất cả các phương thức không tĩnh đều có ít nhất một biến, this, đại diện cho đối tượng hiện tại, đối tượng đã nhận được thông báo gần đây nhất (trong chuỗi hiện tại), gây ra lời gọi phương thức.

**•** Operand stack—Many of the instructions in the Java Virtual Machine take parame-ters. The operand stack is where those parameters are put. The stack is a standard last-in, ﬁrst-out (LIFO) data structure.  
• Ngăn xếp toán hạng—Nhiều lệnh trong Máy ảo Java lấy tham số. Ngăn xếp toán hạng là nơi đặt các tham số đó. Ngăn xếp là cấu trúc dữ liệu nhập trước xuất trước (LIFO) tiêu chuẩn.

Here is the byte-code generated for resetId():  
Đây là mã byte được tạo cho resetId():

**Mnemonic  
trí nhớ**

ALOAD 0

**Description**  
Sự miêu tả

Load the 0th variable onto the operand stack. What is the 0th variable? It is **this**., the current object. When the method was called, the receiver of the message, an instance of Example, was pushed into the local variable array of the frame created for method invocation. This is always the ﬁrst variable put in every instance method.  
Nạp biến thứ 0 vào ngăn xếp toán hạng. Biến thứ 0 là gì? Nó là cái này, đối tượng hiện tại. Khi phương thức được gọi, người nhận thông báo, một thể hiện của Ví dụ, được đẩy vào mảng biến cục bộ của khung được tạo để gọi phương thức. Đây luôn là biến đầu tiên được đặt trong mọi phương thức thể hiện.

**Operand Stack After**  
Ngăn xếp toán hạng sau

this

<https://www.facebook.com/groups/phattrienlaptrinh>

**Possible Paths of Execution** 325  
Các đường thực hiện khả thi 325

**Mnemonic  
trí nhớ**

ICONST\_0

PUTFIELD lastId

**Description**  
Sự miêu tả

Put the constant value 0 onto the operand stack.  
Đặt giá trị không đổi 0 vào ngăn xếp toán hạng.

Store the top value on the stack (which is 0) into the ﬁeld value of the object referred to by the object reference one away from the top of the stack, **this**.  
Lưu trữ giá trị trên cùng của ngăn xếp (là 0) vào giá trị trường của đối tượng được tham chiếu bởi tham chiếu đối tượng cách xa đỉnh của ngăn xếp, this.

**Operand Stack After**  
Ngăn xếp toán hạng sau

this, 0

<empty>

These three instructions are guaranteed to be atomic because, although the thread executing them could be interrupted after any one of them, the information for the PUTFIELD instruction (the constant value 0 on the top of the stack and the reference to this one below the top, along with the ﬁeld value) cannot be touched by another thread. So when the assignment occurs, we are guaranteed that the value 0 will be stored in the ﬁeld value. The operation is atomic. The operands all deal with information local to the method, so there is no interference between multiple threads.  
Ba lệnh này được đảm bảo là nguyên tử bởi vì, mặc dù luồng thực thi chúng có thể bị gián đoạn sau bất kỳ lệnh nào trong số chúng, thông tin cho lệnh PUTFIELD (giá trị không đổi 0 ở trên cùng của ngăn xếp và tham chiếu đến lệnh này ở dưới cùng , cùng với giá trị trường) không thể được chạm vào bởi một luồng khác. Vì vậy, khi phép gán xảy ra, chúng tôi được đảm bảo rằng giá trị 0 sẽ được lưu trữ trong giá trị trường. Các hoạt động là nguyên tử. Tất cả các toán hạng đều xử lý thông tin cục bộ của phương thức, do đó không có sự can thiệp giữa nhiều luồng.

So if these three instructions are executed by ten threads, there are 4.38679733629e+24 possible orderings. However, there is only one possible outcome, so the different orderings are irrelevant. It just so happens that the same outcome is guaranteed for longs in this case as well. Why? All ten threads are assigning a constant value. Even if they interleave with each other, the end result is the same.  
Vì vậy, nếu ba hướng dẫn này được thực hiện bởi mười luồng, thì có 4,38679733629e+24 thứ tự có thể. Tuy nhiên, chỉ có một kết quả có thể xảy ra, vì vậy các thứ tự khác nhau là không liên quan. Nó chỉ xảy ra rằng kết quả tương tự cũng được đảm bảo trong thời gian dài trong trường hợp này. Tại sao? Tất cả mười luồng đang gán một giá trị không đổi. Ngay cả khi chúng xen kẽ với nhau, kết quả cuối cùng là như nhau.

With the ++ operation in the getNextId method, there are going to be problems. Assume that lastId holds 42 at the beginning of this method. Here is the byte-code for this new method:  
Với thao tác ++ trong phương thức getNextId, sẽ có vấn đề. Giả sử rằng lastId giữ 42 khi bắt đầu phương thức này. Đây là mã byte cho phương thức mới này:

**Mnemonic  
trí nhớ**

ALOAD 0 DUP

GETFIELD lastId

ICONST\_1 IADD

DUP\_X1 PUTFIELD value

IRETURN

**Description**  
Sự miêu tả

Load this onto the operand stack  
Tải cái này lên ngăn xếp toán hạng

Copy the top of the stack. We now have two copies of this on the operand stack.  
Sao chép phần trên cùng của ngăn xếp. Bây giờ chúng ta có hai bản sao này trên ngăn xếp toán hạng.

Retrieve the value of the ﬁeld lastId from the object pointed to on the top of the stack (this) and store that value back on to the stack.  
Truy xuất giá trị của trường lastId từ đối tượng được trỏ tới trên đỉnh ngăn xếp (cái này) và lưu trữ giá trị đó trở lại ngăn xếp.

Push the integer constant 1 on the stack.  
Đẩy hằng số nguyên 1 vào ngăn xếp.

Integer add the top two values on the operand stack and store the result back on to the operand stack.  
Số nguyên cộng hai giá trị trên cùng vào ngăn xếp toán hạng và lưu kết quả trở lại vào ngăn xếp toán hạng.

Duplicate the value 43 and put it before this.  
Nhân đôi giá trị 43 và đặt nó trước giá trị này.

Store the top value on the operand stack, 43, into the ﬁeld value of the current object, represented by the next-to-top value on the operand stack, this.  
Lưu trữ giá trị trên cùng trên ngăn xếp toán hạng, 43, vào giá trị trường của đối tượng hiện tại, được biểu thị bằng giá trị từ đầu đến cuối trên ngăn xếp toán hạng, this.

return the top (and only) value on the stack.  
trả về giá trị trên cùng (và duy nhất) trên ngăn xếp.

**Operand Stack After**  
Ngăn xếp toán hạng sau

this this, this

this, 42

this, 42, 1 this, 43

43, this, 43 43

<empty>

<https://www.facebook.com/groups/phattrienlaptrinh>

326 **Appendix A: Concurrency II**  
326 Phụ lục A: Tương tranh II

Imagine the case where the ﬁrst thread completes the ﬁrst three instructions, up to and including GETFIELD, and then it is interrupted. A second thread takes over and performs the entire method, incrementing lastId by one; it gets 43 back. Then the ﬁrst thread picks up where it left off; 42 is still on the operand stack because that was the value of lastId when it executed GETFIELD. It adds one to get 43 again and stores the result. The value 43 is returned to the ﬁrst thread as well. The result is that one of the increments is lost because the ﬁrst thread stepped on the second thread after the second thread interrupted the ﬁrst thread.  
Hãy tưởng tượng trường hợp luồng đầu tiên hoàn thành ba hướng dẫn đầu tiên, bao gồm cả GETFIELD, và sau đó nó bị gián đoạn. Một luồng thứ hai tiếp quản và thực hiện toàn bộ phương thức, tăng lastId lên một; nó được 43 trở lại. Sau đó, chuỗi đầu tiên bắt đầu từ nơi nó đã dừng lại; 42 vẫn nằm trên ngăn xếp toán hạng vì đó là giá trị của lastId khi nó thực thi GETFIELD. Nó thêm một lần nữa để nhận được 43 và lưu trữ kết quả. Giá trị 43 cũng được trả về luồng đầu tiên. Kết quả là một trong các phần gia tăng bị mất do luồng đầu tiên giẫm lên luồng thứ hai sau khi luồng thứ hai ngắt luồng thứ nhất.

Making the getNexId() method synchronized ﬁxes this problem.  
Việc đồng bộ hóa phương thức getNexId() sẽ khắc phục sự cố này.

[**Conclusion**](#_page_685_0)  
Phần kết luận

An intimate understanding of byte-code is not necessary to understand how threads can step on each other. If you can understand this one example, it should demonstrate the pos-sibility of multiple threads stepping on each other, which is enough knowledge.  
Hiểu sâu về mã byte là không cần thiết để hiểu cách các luồng có thể dẫm lên nhau. Nếu bạn có thể hiểu một ví dụ này, thì nó sẽ chứng minh khả năng có nhiều luồng giẫm lên nhau, thế là đủ kiến thức.

That being said, what this trivial example demonstrates is a need to understand the memory model enough to know what is and is not safe. It is a common misconception that the ++ (pre- or post-increment) operator is atomic, and it clearly is not. This means you need to know:  
Điều đó đang được nói, những gì ví dụ tầm thường này chứng minh là cần phải hiểu mô hình bộ nhớ đủ để biết điều gì là an toàn và không an toàn. Một quan niệm sai lầm phổ biến là toán tử ++ (tăng trước hoặc sau) là nguyên tử và rõ ràng là không phải vậy. Điều này có nghĩa là bạn cần biết:

**•** Where there are shared objects/values  
• Nơi có các đối tượng/giá trị được chia sẻ

**•** The code that can cause concurrent read/update issues **•** How to guard such concurrent issues from happening  
• Mã có thể gây ra sự cố đọc/cập nhật đồng thời • Cách bảo vệ các sự cố đồng thời đó khỏi xảy ra

[**Knowing Your Library**](#_page_685_0)  
Biết thư viện của bạn

[**Executor Framework**](#_page_685_0)  
Khung thực thi

As demonstrated in the ExecutorClientScheduler.java on page 321, the Executor frame-work introduced in Java 5 allows for sophisticated execution using thread pools. This is a class in the java.util.concurrent package.  
Như đã trình bày trong ExecutorClientScheduler.java ở trang 321, khung Executor được giới thiệu trong Java 5 cho phép thực thi tinh vi bằng cách sử dụng nhóm luồng. Đây là một lớp trong gói java.util.concurrent.

If you are creating threads and are not using a thread pool or *are* using a hand-written one, you should consider using the Executor. It will make your code cleaner, easier to fol-low, and smaller.  
Nếu bạn đang tạo chuỗi và không sử dụng nhóm chuỗi hoặc đang sử dụng nhóm viết tay, bạn nên cân nhắc sử dụng Executor. Nó sẽ làm cho mã của bạn sạch hơn, dễ theo dõi hơn và nhỏ hơn.

The Executor framework will pool threads, resize automatically, and recreate threads if necessary. It also supports *futures,* a common concurrent programming construct. The Executor framework works with classes that implement Runnable and also works with classes that implement the Callable interface. A Callable looks like a Runnable, but it can return a result, which is a common need in multithreaded solutions.  
Khung Executor sẽ tổng hợp các luồng, tự động thay đổi kích thước và tạo lại các luồng nếu cần. Nó cũng hỗ trợ hợp đồng tương lai, một cấu trúc lập trình đồng thời phổ biến. Khung Executor hoạt động với các lớp triển khai Runnable và cũng hoạt động với các lớp triển khai giao diện Callable. Callable trông giống như Runnable, nhưng nó có thể trả về kết quả, đây là nhu cầu phổ biến trong các giải pháp đa luồng.

A *future* is handy when code needs to execute multiple, independent operations and wait for both to ﬁnish:  
Một tương lai rất hữu ích khi mã cần thực thi nhiều hoạt động độc lập và đợi cả hai hoạt động kết thúc:

public String processRequest(String message) throws Exception { Callable<String> makeExternalCall = new Callable<String>() {

<https://www.facebook.com/groups/phattrienlaptrinh>

**Knowing Your Library** 327  
Biết thư viện của bạn 327

public String call() throws Exception { String result = "";

// make external request return result;

} };

Future<String> result = executorService.submit(makeExternalCall); String partialResult = doSomeLocalProcessing();

return result.get() + partialResult; }

In this example, the method starts executing the makeExternalCall object. The method con-tinues other processing. The ﬁnal line calls result.get(), which blocks until the future completes.  
Trong ví dụ này, phương thức bắt đầu thực thi đối tượng makeExternalCall. Phương pháp tiếp tục xử lý khác. Dòng cuối cùng gọi result.get(), chặn cho đến khi tương lai hoàn thành.

[**Nonblocking Solutions**](#_page_685_0)  
Giải pháp không chặn

The Java 5 VM takes advantage of modern processor design, which supports reliable, nonblocking updates. Consider, for example, a class that uses synchronization (and there-fore blocking) to provide a thread-safe update of a value:  
Máy ảo Java 5 tận dụng lợi thế của thiết kế bộ xử lý hiện đại, hỗ trợ các bản cập nhật không chặn, đáng tin cậy. Ví dụ, hãy xem xét một lớp sử dụng đồng bộ hóa (và do đó chặn) để cung cấp bản cập nhật an toàn cho luồng của một giá trị:

public class ObjectWithValue { private int value;

public void synchronized incrementValue() { ++value; } public int getValue() { return value; }

}

Java 5 has a series of new classes for situations like this: AtomicBoolean, AtomicInteger, and AtomicReference are three examples; there are several more. We can rewrite the above code to use a nonblocking approach as follows:  
Java 5 có một loạt các lớp mới cho các tình huống như sau: AtomicBoolean, AtomicInteger và AtomicReference là ba ví dụ; có nhiều hơn nữa. Chúng ta có thể viết lại đoạn mã trên để sử dụng phương pháp không chặn như sau:

public class ObjectWithValue {

private AtomicInteger value = new AtomicInteger(0);

public void incrementValue() { value.incrementAndGet();

}

public int getValue() { return value.get();

} }

Even though this uses an object instead of a primitive and sends messages like incrementAndGet() instead of ++, the performance of this class will nearly always beat the previous version. In some cases it will only be slightly faster, but the cases where it will be slower are virtually nonexistent.  
Mặc dù điều này sử dụng một đối tượng thay vì nguyên thủy và gửi các thông báo như incrementAndGet() thay vì ++, nhưng hiệu suất của lớp này sẽ gần như luôn đánh bại phiên bản trước. Trong một số trường hợp, nó sẽ chỉ nhanh hơn một chút, nhưng những trường hợp chậm hơn thì hầu như không có.

How is this possible? Modern processors have an operation typically called *Compare and Swap (CAS)*. This operation is analogous to optimistic locking in databases, whereas the synchronized version is analogous to pessimistic locking.  
Sao có thể như thế được? Bộ xử lý hiện đại có một hoạt động thường được gọi là So sánh và Hoán đổi (CAS). Hoạt động này tương tự như khóa lạc quan trong cơ sở dữ liệu, trong khi phiên bản được đồng bộ hóa tương tự như khóa bi quan.

<https://www.facebook.com/groups/phattrienlaptrinh>

328 **Appendix A: Concurrency II**  
328 Phụ lục A: Tương tranh II

The synchronized keyword always acquires a lock, even when a second thread is not trying to update the same value. Even though the performance of intrinsic locks has improved from version to version, they are still costly.  
Từ khóa được đồng bộ hóa luôn có khóa, ngay cả khi chuỗi thứ hai không cố cập nhật cùng một giá trị. Mặc dù hiệu suất của khóa nội tại đã được cải thiện từ phiên bản này sang phiên bản khác, nhưng chúng vẫn rất tốn kém.

The nonblocking version starts with the assumption that multiple threads generally do not modify the same value often enough that a problem will arise. Instead, it efﬁciently detects whether such a situation has occurred and retries until the update happens success-fully.This detection is almost always less costly than acquiring a lock, even in moderate to high contention situations.  
Phiên bản không chặn bắt đầu với giả định rằng nhiều luồng thường không sửa đổi cùng một giá trị đủ thường xuyên để phát sinh sự cố. Thay vào đó, nó phát hiện một cách hiệu quả liệu một tình huống như vậy đã xảy ra hay chưa và thử lại cho đến khi cập nhật diễn ra thành công. Việc phát hiện này hầu như luôn ít tốn kém hơn so với việc giành được một khóa, ngay cả trong các tình huống tranh chấp từ trung bình đến cao.

How does the Virtual Machine accomplish this? The CAS operation is atomic. Logi-cally, the CAS operation looks something like the following:  
Máy ảo thực hiện việc này như thế nào? Hoạt động của CAS là nguyên tử. Về mặt logic, thao tác CAS trông giống như sau:

int variableBeingSet;

void simulateNonBlockingSet(int newValue) { int currentValue;

do {

currentValue = variableBeingSet

} while(currentValue != compareAndSwap(currentValue, newValue)); }

int synchronized compareAndSwap(int currentValue, int newValue) { if(variableBeingSet == currentValue) {

variableBeingSet = newValue; return currentValue;

}

return variableBeingSet; }

When a method attempts to update a shared variable, the CAS operation veriﬁes that the variable getting set still has the last known value. If so, then the variable is changed. If not, then the variable is not set because another thread managed to get in the way. The method making the attempt (using the CAS operation) sees that the change was not made and retries.  
Khi một phương thức cố gắng cập nhật một biến được chia sẻ, thao tác CAS sẽ xác minh rằng biến được đặt vẫn có giá trị đã biết cuối cùng. Nếu vậy, thì biến được thay đổi. Nếu không, thì biến không được đặt vì một chuỗi khác đã quản lý để cản trở. Phương pháp thực hiện thử (sử dụng thao tác CAS) thấy rằng thay đổi không được thực hiện và thử lại.

[**Nonthread-Safe Classes**](#_page_685_0)  
Các lớp không an toàn theo luồng

There are some classes that are inherently not thread safe. Here are a few examples:  
Có một số lớp vốn không an toàn cho luồng. Dưới đây là một vài ví dụ:

**•** SimpleDateFormat

**•** Database Connections **•** Containers in java.util **•** Servlets  
• Kết nối cơ sở dữ liệu • Bộ chứa trong java.util • Servlet

Note that some collection classes have individual methods that are thread-safe. However, any operation that involves calling more than one method is not. For example, if you do not want to replace something in a HashTable because it is already there, you might write the following code:  
Lưu ý rằng một số lớp bộ sưu tập có các phương thức riêng lẻ an toàn cho luồng. Tuy nhiên, bất kỳ thao tác nào liên quan đến việc gọi nhiều hơn một phương thức thì không. Ví dụ: nếu bạn không muốn thay thế thứ gì đó trong HashTable vì nó đã có sẵn, bạn có thể viết đoạn mã sau:

if(!hashTable.containsKey(someKey)) { hashTable.put(someKey, new SomeValue());

}

<https://www.facebook.com/groups/phattrienlaptrinh>

**Dependencies Between Methods Can Break Concurrent Code** 329  
Sự phụ thuộc giữa các phương thức có thể phá vỡ mã đồng thời 329

Each individual method is thread-safe. However, another thread might add a value in between the containsKey and put calls. There are several options to ﬁx this problem.  
Mỗi phương thức riêng lẻ đều an toàn cho luồng. Tuy nhiên, một chuỗi khác có thể thêm một giá trị vào giữa các cuộc gọi chứaKey và đặt. Có một số tùy chọn để khắc phục vấn đề này.

**•** Lock the HashTable ﬁrst, and make sure all other users of the HashTable do the same— client-based locking:  
• Khóa HashTable trước và đảm bảo rằng tất cả những người dùng HashTable khác cũng làm như vậy— khóa dựa trên máy khách:

synchronized(map) { if(!map.conainsKey(key))

map.put(key,value); }

**•** Wrap the HashTable in its own object and use a different API—server-based locking using an ADAPTER:  
• Bọc HashTable trong đối tượng riêng của nó và sử dụng API khác—khóa dựa trên máy chủ bằng cách sử dụng ADAPTER:

public class WrappedHashtable<K, V> {

private Map<K, V> map = new Hashtable<K, V>();

public synchronized void putIfAbsent(K key, V value) { if (map.containsKey(key))

map.put(key, value); }

}

**•** Use the thread-safe collections:  
• Sử dụng các bộ sưu tập thread-safe:

ConcurrentHashMap<Integer, String> map = new ConcurrentHashMap<Integer, String>();

map.putIfAbsent(key, value);

The collections in java.util.concurrent have operations like putIfAbsent() to accommo-date such operations.  
Các bộ sưu tập trong java.util.concurrent có các hoạt động như putIfAbsent() để cập nhật các hoạt động đó.

[**Dependencies Between Methods**](#_page_685_0)  
Sự phụ thuộc giữa các phương thức

[**Can Break Concurrent Code**](#_page_685_0)  
Có thể phá mã đồng thời

Here is a trivial example of a way to introduce dependencies between methods:  
Đây là một ví dụ tầm thường về cách giới thiệu các phụ thuộc giữa các phương thức:

public class IntegerIterator implements Iterator<Integer> private Integer nextValue = 0;

public synchronized boolean hasNext() { return nextValue < 100000;

}

public synchronized Integer next() { if (nextValue == 100000)

throw new IteratorPastEndException(); return nextValue++;

}

public synchronized Integer getNextValue() { return nextValue;

} }

Here is some code to use this IntegerIterator:  
Đây là một số mã để sử dụng IntegerIterator này:

IntegerIterator iterator = new IntegerIterator(); while(iterator.hasNext()) {

<https://www.facebook.com/groups/phattrienlaptrinh>

330 **Appendix A: Concurrency II**  
330 Phụ lục A: Tương tranh II

int nextValue = iterator.next(); // do something with nextValue

}

If one thread executes this code, there will be no problem. But what happens if two threads attempt to share a single instance of IngeterIterator with the intent that each thread will process the values it gets, but that each element of the list is processed only once? Most of the time, nothing bad happens; the threads happily share the list, processing the elements they are given by the iterator and stopping when the iterator is complete. However, there is a small chance that, at the end of the iteration, the two threads will interfere with each other and cause one thread to go beyond the end of the iterator and throw an exception.  
Nếu một luồng thực thi mã này, sẽ không có vấn đề gì. Nhưng điều gì sẽ xảy ra nếu hai luồng cố gắng chia sẻ một phiên bản duy nhất của IngeterIterator với ý định rằng mỗi luồng sẽ xử lý các giá trị mà nó nhận được, nhưng mỗi phần tử của danh sách chỉ được xử lý một lần? Hầu hết thời gian, không có gì xấu xảy ra; các luồng vui vẻ chia sẻ danh sách, xử lý các phần tử mà chúng được cung cấp bởi trình lặp và dừng khi trình lặp hoàn tất. Tuy nhiên, có một khả năng nhỏ là khi kết thúc quá trình lặp, hai luồng sẽ can thiệp lẫn nhau và khiến một luồng vượt ra ngoài điểm cuối của trình lặp và đưa ra một ngoại lệ.

Here’s the problem: Thread 1 asks the question hasNext(), which returns true. Thread 1 gets preempted and then Thread 2 asks the same question, which is still true. Thread 2 then calls next(), which returns a value as expected but has a side effect of making hasNext() return false. Thread 1 starts up again, thinking hasNext() is still true, and then calls next(). Even though the individual methods are synchronized, the client uses ***two*** methods.  
Đây là vấn đề: Chủ đề 1 đặt câu hỏi hasNext(), câu hỏi này trả về true. Chủ đề 1 được ưu tiên và sau đó Chủ đề 2 hỏi cùng một câu hỏi, điều này vẫn đúng. Chủ đề 2 sau đó gọi next(), trả về một giá trị như mong đợi nhưng có tác dụng phụ là làm cho hasNext() trả về false. Chủ đề 1 khởi động lại, nghĩ rằng hasNext() vẫn đúng và sau đó gọi next(). Mặc dù các phương pháp riêng lẻ được đồng bộ hóa, khách hàng sử dụng hai phương pháp.

This is a real problem and an example of the kinds of problems that crop up in con-current code. In this particular situation this problem is especially subtle because the only time where this causes a fault is when it happens during the ﬁnal iteration of the iterator. If the threads happen to break just right, then one of the threads could go beyond the end of the iterator. This is the kind of bug that happens long after a system has been in pro-duction, and it is hard to track down.  
Đây là một vấn đề thực sự và là một ví dụ về các loại vấn đề nảy sinh trong mã đồng thời. Trong tình huống cụ thể này, vấn đề này đặc biệt tinh tế bởi vì lần duy nhất mà điều này gây ra lỗi là khi nó xảy ra trong lần lặp cuối cùng của trình vòng lặp. Nếu các luồng xảy ra sự cố vừa phải, thì một trong các luồng có thể vượt ra ngoài phần cuối của trình vòng lặp. Đây là loại lỗi xảy ra rất lâu sau khi hệ thống được đưa vào sản xuất và rất khó để theo dõi.

You have three options:  
Bạn có ba lựa chọn:

**•** Tolerate the failure.  
• Chịu đựng thất bại.

**•** Solve the problem by changing the client: client-based locking  
• Giải quyết vấn đề bằng cách thay đổi máy khách: khóa dựa trên máy khách

**•** Solve the problem by changing the server, which additionally changes the client: server-based locking  
• Giải quyết vấn đề bằng cách thay đổi máy chủ, đồng thời thay đổi máy khách: khóa dựa trên máy chủ

[**Tolerate the Failure**](#_page_685_0)  
chịu đựng sự thất bại

Sometimes you can set things up such that the failure causes no harm. For example, the above client could catch the exception and clean up. Frankly, this is a bit sloppy. It’s rather like cleaning up memory leaks by rebooting at midnight.  
Đôi khi bạn có thể thiết lập mọi thứ sao cho lỗi không gây hại gì. Ví dụ: ứng dụng khách ở trên có thể bắt ngoại lệ và dọn dẹp. Thành thật mà nói, điều này là một chút cẩu thả. Nó giống như dọn dẹp rò rỉ bộ nhớ bằng cách khởi động lại lúc nửa đêm.

[**Client-Based Locking**](#_page_685_0)  
Khóa dựa trên máy khách

To make IntegerIterator work correctly with multiple threads, change this client (and every other client) as follows:  
Để làm cho IntegerIterator hoạt động chính xác với nhiều luồng, hãy thay đổi ứng dụng khách này (và mọi ứng dụng khách khác) như sau:

IntegerIterator iterator = new IntegerIterator();

while (true) { int nextValue;

<https://www.facebook.com/groups/phattrienlaptrinh>

**Dependencies Between Methods Can Break Concurrent Code** 331  
Sự phụ thuộc giữa các phương thức có thể phá vỡ mã đồng thời 331

synchronized (iterator) { if (!iterator.hasNext())

break;

nextValue = iterator.next(); }

doSometingWith(nextValue); }

Each client introduces a lock via the synchronized keyword. This duplication violates the DRY principle, but it might be necessary if the code uses non-thread-safe third-party tools.  
Mỗi khách hàng giới thiệu một khóa thông qua từ khóa được đồng bộ hóa. Sự sao chép này vi phạm nguyên tắc DRY, nhưng có thể cần thiết nếu mã sử dụng các công cụ của bên thứ ba không an toàn cho luồng.

This strategy is risky because all programmers who use the server must remember to lock it before using it and unlock it when done. Many (many!) years ago I worked on a system that employed client-based locking on a shared resource. The resource was used in hundreds of different places throughout the code. One poor programmer forgot to lock the resource in one of those places.  
Chiến lược này rất rủi ro vì tất cả các lập trình viên sử dụng máy chủ đều phải nhớ khóa nó trước khi sử dụng và mở khóa khi hoàn tất. Nhiều (nhiều!) năm trước, tôi đã làm việc trên một hệ thống sử dụng khóa dựa trên ứng dụng khách trên một tài nguyên được chia sẻ. Tài nguyên đã được sử dụng ở hàng trăm nơi khác nhau trong toàn bộ mã. Một lập trình viên kém cỏi đã quên khóa tài nguyên ở một trong những nơi đó.

The system was a multi-terminal time-sharing system running accounting software for Local 705 of the trucker’s union. The computer was in a raised-ﬂoor, environment-controlled room 50 miles north of the Local 705 headquarters. At the headquarters they had dozens of data entry clerks typing union dues postings into the terminals. The termi-nals were connected to the computer using dedicated phone lines and 600bps half-duplex modems. (This was a very, *very* long time ago.)  
Hệ thống này là một hệ thống chia sẻ thời gian đa đầu cuối chạy phần mềm kế toán cho Local 705 của hiệp hội tài xế xe tải. Máy tính được đặt trong một căn phòng có sàn nâng, được kiểm soát môi trường, cách trụ sở 705 Địa phương 50 dặm về phía bắc. Tại trụ sở chính, họ có hàng tá nhân viên nhập liệu đánh các thông báo về hội phí công đoàn vào các thiết bị đầu cuối. Các thiết bị đầu cuối được kết nối với máy tính bằng đường dây điện thoại chuyên dụng và modem bán song công 600bps. (Đây là một thời gian rất, rất lâu trước đây.)

About once per day, one of the terminals would “lock up.” There was no rhyme or rea-son to it. The lock up showed no preference for particular terminals or particular times. It was as though there were someone rolling dice choosing the time and terminal to lock up. Sometimes more than one terminal would lock up. Sometimes days would go by without any lock-ups.  
Khoảng một lần mỗi ngày, một trong các thiết bị đầu cuối sẽ bị khóa. Không có vần điệu hay lý do cho nó. Việc khóa không hiển thị ưu tiên cho các thiết bị đầu cuối cụ thể hoặc thời gian cụ thể. Như thể có ai đó tung xúc xắc chọn thời gian và thiết bị đầu cuối để khóa. Đôi khi nhiều hơn một thiết bị đầu cuối sẽ bị khóa. Đôi khi nhiều ngày sẽ trôi qua mà không có bất kỳ khóa nào.

At ﬁrst the only solution was a reboot. But reboots were tough to coordinate. We had to call the headquarters and get everyone to ﬁnish what they were doing on all the termi-nals. Then we could shut down and restart. If someone was doing something important that took an hour or two, the locked up terminal simply had to stay locked up.  
Lúc đầu, giải pháp duy nhất là khởi động lại. Nhưng việc khởi động lại rất khó phối hợp. Chúng tôi phải gọi cho trụ sở chính và yêu cầu mọi người hoàn thành công việc họ đang làm trên tất cả các thiết bị đầu cuối. Sau đó, chúng tôi có thể tắt và khởi động lại. Nếu ai đó đang làm việc gì đó quan trọng trong một hoặc hai giờ, thì thiết bị đầu cuối bị khóa chỉ cần tiếp tục bị khóa.

After a few weeks of debugging we found that the cause was a ring-buffer counter that had gotten out of sync with its pointer. This buffer controlled output to the terminal. The pointer value indicated that the buffer was empty, but the counter said it was full. Because it was empty, there was nothing to display; but because it was also full, nothing could be added to the buffer to be displayed on the screen.  
Sau một vài tuần gỡ lỗi, chúng tôi phát hiện ra rằng nguyên nhân là do bộ đếm bộ đệm vòng không đồng bộ với con trỏ của nó. Bộ đệm này kiểm soát đầu ra cho thiết bị đầu cuối. Giá trị con trỏ chỉ ra rằng bộ đệm trống, nhưng bộ đếm cho biết nó đã đầy. Bởi vì nó trống rỗng, không có gì để hiển thị; nhưng vì nó cũng đã đầy nên không thể thêm gì vào bộ đệm để hiển thị trên màn hình.

So we knew why the terminals were locking, but we didn’t know why the ring buffer was getting out of sync. So we added a hack to work around the problem. It was possible to read the front panel switches on the computer. (This was a very, very, *very* long time ago.) We wrote a little trap function that detected when one of these switches was thrown and then looked for a ring buffer that was both empty and full. If one was found, it reset that buffer to empty. *Voila!* The locked-up terminal(s) started displaying again.  
Vì vậy, chúng tôi biết lý do tại sao các thiết bị đầu cuối bị khóa, nhưng chúng tôi không biết tại sao bộ đệm vòng không đồng bộ. Vì vậy, chúng tôi đã thêm một bản hack để khắc phục sự cố. Có thể đọc các công tắc bảng điều khiển phía trước trên máy tính. (Điều này đã xảy ra cách đây rất, rất, rất lâu rồi.) Chúng tôi đã viết một chức năng bẫy nhỏ để phát hiện khi một trong các công tắc này được ném ra và sau đó tìm kiếm bộ đệm vòng vừa trống vừa đầy. Nếu một cái được tìm thấy, nó sẽ đặt lại bộ đệm đó thành trống. Thì đấy! (Các) thiết bị đầu cuối bị khóa bắt đầu hiển thị lại.

So now we didn’t have to reboot the system when a terminal locked up. The Local would simply call us and tell us we had a lock-up, and then we just walked into the com-puter room and ﬂicked a switch.  
Vì vậy, bây giờ chúng tôi không phải khởi động lại hệ thống khi thiết bị đầu cuối bị khóa. Người địa phương chỉ cần gọi cho chúng tôi và thông báo rằng chúng tôi đã bị khóa máy, sau đó chúng tôi chỉ cần bước vào phòng máy tính và bật công tắc.

<https://www.facebook.com/groups/phattrienlaptrinh>

332 **Appendix A: Concurrency II**  
332 Phụ lục A: Tương tranh II

Of course sometimes they worked on the weekends, and we didn’t. So we added a function to the scheduler that checked all the ring buffers once per minute and reset any that were both empty and full. This caused the displays to unclog before the Local could even get on the phone.  
Tất nhiên đôi khi họ làm việc vào cuối tuần, còn chúng tôi thì không. Vì vậy, chúng tôi đã thêm một chức năng vào bộ lập lịch để kiểm tra tất cả các bộ đệm đổ chuông một lần mỗi phút và đặt lại bất kỳ bộ đệm nào trống và đầy. Điều này khiến màn hình không bị tắc nghẽn trước khi Local thậm chí có thể truy cập vào điện thoại.

It was several more weeks of poring over page after page of monolithic assembly lan-guage code before we found the culprit. We had done the math and calculated that the fre-quency of the lock-ups was consistent with a single unprotected use of the ring buffer. So all we had to do was ﬁnd that one faulty usage. Unfortunately, this was so very long ago that we didn’t have search tools or cross references or any other kind of automated help. We simply had to pore over listings.  
Phải mất thêm vài tuần nữa để nghiền ngẫm hết trang này đến trang khác của mã ngôn ngữ lắp ráp nguyên khối trước khi chúng tôi tìm ra thủ phạm. Chúng tôi đã tính toán và tính toán rằng tần suất khóa phù hợp với một lần sử dụng bộ đệm vòng không được bảo vệ. Vì vậy, tất cả những gì chúng tôi phải làm là tìm ra cách sử dụng bị lỗi đó. Thật không may, điều này đã xảy ra cách đây rất lâu nên chúng tôi không có các công cụ tìm kiếm hoặc tham khảo chéo hoặc bất kỳ hình thức trợ giúp tự động nào khác. Chúng tôi chỉ đơn giản là phải nghiền ngẫm các danh sách.

I learned an important lesson that cold Chicago winter of 1971. Client-based locking really blows.  
Tôi đã học được một bài học quan trọng vào mùa đông lạnh giá năm 1971 ở Chicago. Khóa dựa trên máy khách thực sự gây ấn tượng mạnh.

[**Server-Based Locking**](#_page_685_0)  
Khóa dựa trên máy chủ

The duplication can be removed by making the following changes to IntegerIterator:  
Sự trùng lặp có thể được loại bỏ bằng cách thực hiện các thay đổi sau đối với IntegerIterator:

public class IntegerIteratorServerLocked { private Integer nextValue = 0;

public synchronized Integer getNextOrNull() { if (nextValue < 100000)

return nextValue++; else

return null; }

}

And the client code changes as well:  
Và mã máy khách cũng thay đổi:

while (true) {

Integer nextValue = iterator.getNextOrNull(); if (next == null)

break;

// do something with nextValue }

In this case we actually change the API of our class to be multithread aware.3 The client needs to perform a null check instead of checking hasNext().  
Trong trường hợp này, chúng tôi thực sự thay đổi API của lớp để nhận biết đa luồng.3 Máy khách cần thực hiện kiểm tra null thay vì kiểm tra hasNext().

In general you should prefer server-based locking for these reasons:  
Nói chung, bạn nên thích khóa dựa trên máy chủ vì những lý do sau:

**•** It reduces repeated code—Client-based locking forces each client to lock the server properly. By putting the locking code into the server, clients are free to use the object and not worry about writing additional locking code.  
• Giảm mã lặp lại—Khóa dựa trên máy khách buộc mỗi máy khách phải khóa máy chủ đúng cách. Bằng cách đặt mã khóa vào máy chủ, khách hàng có thể tự do sử dụng đối tượng và không phải lo lắng về việc viết mã khóa bổ sung.

3. In fact, the Iterator interface is inherently not thread-safe. It was never designed to be used by multiple threads, so this should come as no surprise.  
3. Trên thực tế, giao diện Iterator vốn không an toàn cho luồng. Nó không bao giờ được thiết kế để được sử dụng bởi nhiều luồng, vì vậy điều này sẽ không có gì ngạc nhiên.

<https://www.facebook.com/groups/phattrienlaptrinh>

**Increasing Throughput** 333  
Tăng thông lượng 333

**•** It allows for better performance—You can swap out a thread-safe server for a non-thread safe one in the case of single-threaded deployment, thereby avoiding all overhead.  
• Nó cho phép hiệu suất tốt hơn—Bạn có thể hoán đổi máy chủ an toàn theo luồng lấy máy chủ không an toàn theo luồng trong trường hợp triển khai đơn luồng, do đó tránh được tất cả chi phí hoạt động.

**•** It reduces the possibility of error—All it takes is for one programmer to forget to lock properly.  
• Nó làm giảm khả năng xảy ra lỗi—Tất cả những gì cần thiết là một lập trình viên quên khóa đúng cách.

**•** It enforces a single policy—The policy is in one place, the server, rather than many places, each client.  
• Nó thực thi một chính sách duy nhất—Chính sách này ở một nơi, máy chủ, thay vì nhiều nơi, mỗi máy khách.

**•** It reduces the scope of the shared variables—The client is not aware of them or how they are locked. All of that is hidden in the server. When things break, the number of places to look is smaller.  
• Nó làm giảm phạm vi của các biến được chia sẻ—Máy khách không biết về chúng hoặc cách chúng bị khóa. Tất cả điều đó được ẩn trong máy chủ. Khi mọi thứ bị hỏng, số lượng nơi để xem xét ít hơn.

What if you do not own the server code?  
Nếu bạn không sở hữu mã máy chủ thì sao?

**•** Use an ADAPTER to change the API and add locking  
• Sử dụng ADAPTER để thay đổi API và thêm khóa

public class ThreadSafeIntegerIterator {

private IntegerIterator iterator = new IntegerIterator();

public synchronized Integer getNextOrNull() { if(iterator.hasNext())

return iterator.next(); return null;

} }

**•** OR better yet, use the thread-safe collections with extended interfaces  
• HOẶC tốt hơn nữa, hãy sử dụng các bộ sưu tập an toàn cho luồng với các giao diện mở rộng

[**Increasing Throughput**](#_page_685_0)  
Tăng thông lượng

Let’s assume that we want to go out on the net and read the contents of a set of pages from a list of URLs. As each page is read, we will parse it to accumulate some statistics. Once all the pages are read, we will print a summary report.  
Giả sử rằng chúng ta muốn truy cập mạng và đọc nội dung của một tập hợp các trang từ danh sách các URL. Khi mỗi trang được đọc, chúng tôi sẽ phân tích cú pháp trang đó để tích lũy một số thống kê. Khi tất cả các trang được đọc, chúng tôi sẽ in một báo cáo tóm tắt.

The following class returns the contents of one page, given a URL.  
Lớp sau trả về nội dung của một trang, được cung cấp một URL.

public class PageReader { //...

public String getPageFor(String url) { HttpMethod method = new GetMethod(url);

try { httpClient.executeMethod(method);

String response = method.getResponseBodyAsString(); return response;

} catch (Exception e) { handle(e);

} finally { method.releaseConnection();

} }

}

<https://www.facebook.com/groups/phattrienlaptrinh>

334 **Appendix A: Concurrency II**  
334 Phụ lục A: Tương tranh II

The next class is the iterator that provides the contents of the pages based on an iterator of URLs:  
Lớp tiếp theo là trình lặp cung cấp nội dung của các trang dựa trên trình lặp URL:

public class PageIterator { private PageReader reader; private URLIterator urls;

public PageIterator(PageReader reader, URLIterator urls) { this.urls = urls;

this.reader = reader; }

public synchronized String getNextPageOrNull() { if (urls.hasNext())

getPageFor(urls.next()); else

return null; }

public String getPageFor(String url) { return reader.getPageFor(url);

} }

An instance of the PageIterator can be shared between many different threads, each one using it’s own instance of the PageReader to read and parse the pages it gets from the iterator.  
Một thể hiện của PageIterator có thể được chia sẻ giữa nhiều luồng khác nhau, mỗi luồng sử dụng phiên bản riêng của PageReader để đọc và phân tích cú pháp các trang mà nó nhận được từ trình vòng lặp.

Notice that we’ve kept the synchronized block very small. It contains just the critical section deep inside the PageIterator. It is always better to synchronize as little as possible as opposed to synchronizing as much as possible.  
Lưu ý rằng chúng tôi đã giữ cho khối được đồng bộ hóa rất nhỏ. Nó chỉ chứa phần quan trọng sâu bên trong PageIterator. Đồng bộ hóa càng ít càng tốt luôn tốt hơn là đồng bộ hóa càng nhiều càng tốt.

[**Single-Thread Calculation of Throughput**](#_page_685_0)  
Tính toán thông lượng đơn luồng

Now lets do some simple calculations. For the purpose of argument, assume the following:  
Bây giờ hãy thực hiện một số phép tính đơn giản. Với mục đích tranh luận, giả sử như sau:

**•** I/O time to retrieve a page (average): 1 second  
• Thời gian I/O để truy xuất một trang (trung bình): 1 giây

**•** Processing time to parse page (average): .5 seconds  
• Thời gian xử lý để phân tích trang (trung bình): 0,5 giây

**•** I/O requires 0 percent of the CPU while processing requires 100 percent.  
• I/O yêu cầu 0 phần trăm CPU trong khi quá trình xử lý yêu cầu 100 phần trăm.

For *N* pages being processed by a single thread, the total execution time is 1.5 sec-onds \* *N*. Figure A-1 shows a snapshot of 13 pages or about 19.5 seconds.  
Đối với N trang đang được xử lý bởi một luồng đơn, tổng thời gian thực hiện là 1,5 giây \* N. Hình A-1 hiển thị ảnh chụp nhanh 13 trang hoặc khoảng 19,5 giây.

**Figure A-1** Single thread  
Hình A-1 Sợi đơn

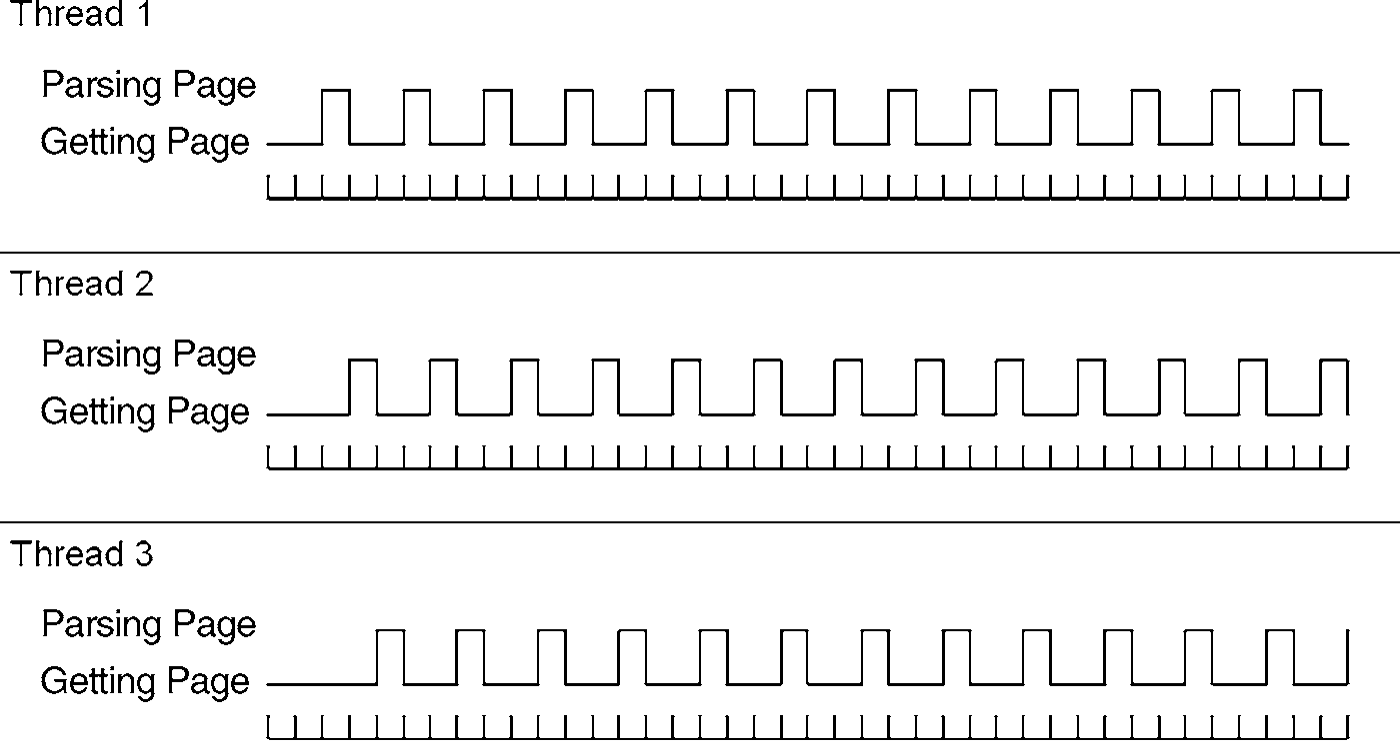
<https://www.facebook.com/groups/phattrienlaptrinh>

**Deadlock** 335  
Bế tắc 335

[**Multithread Calculation of Throughput**](#_page_685_0)  
Tính toán thông lượng đa luồng

If it is possible to retrieve pages in any order and process the pages independently, then it is possible to use multiple threads to increase throughput. What happens if we use three threads? How many pages can we acquire in the same time?  
Nếu có thể truy xuất các trang theo bất kỳ thứ tự nào và xử lý các trang một cách độc lập thì có thể sử dụng nhiều luồng để tăng thông lượng. Điều gì xảy ra nếu chúng ta sử dụng ba chủ đề? Có bao nhiêu trang chúng ta có thể có được trong cùng một thời điểm?

As you can see in Figure A-2, the multithreaded solution allows the process-bound parsing of the pages to overlap with the I/O-bound reading of the pages. In an idealized world this means that the processor is fully utilized. Each one-second page read is over-lapped with two parses. Thus, we can process two pages per second, which is three times the throughput of the single-threaded solution.  
Như bạn có thể thấy trong Hình A-2, giải pháp đa luồng cho phép phân tích cú pháp theo giới hạn quy trình của các trang trùng lặp với việc đọc các trang theo giới hạn I/O. Trong một thế giới lý tưởng hóa, điều này có nghĩa là bộ xử lý được sử dụng hết công suất. Mỗi trang đọc một giây được xếp chồng lên nhau với hai phân tích cú pháp. Do đó, chúng tôi có thể xử lý hai trang mỗi giây, gấp ba lần thông lượng của giải pháp đơn luồng.

**Figure A-2**  
Hình A-2

Three concurrent threads  
Ba chủ đề đồng thời

[**Deadlock**](#_page_685_0)  
Bế tắc

Imagine a Web application with two shared resource pools of some ﬁnite size:  
Hãy tưởng tượng một ứng dụng Web với hai nhóm tài nguyên được chia sẻ có kích thước hữu hạn nào đó:

**•** A pool of database connections for local work in process storage **•** A pool of MQ connections to a master repository  
• Nhóm kết nối cơ sở dữ liệu cho công việc cục bộ trong lưu trữ quy trình • Nhóm kết nối MQ tới kho lưu trữ chính

Assume there are two operations in this application, create and update:  
Giả sử có hai thao tác trong ứng dụng này, tạo và cập nhật:

**•** Create—Acquire connection to master repository and database. Talk to service master repository and then store work in local work in process database.  
• Tạo—Nhận kết nối tới kho lưu trữ chính và cơ sở dữ liệu. Nói chuyện với kho lưu trữ chính của dịch vụ và sau đó lưu trữ công việc trong công việc cục bộ trong cơ sở dữ liệu quy trình.

<https://www.facebook.com/groups/phattrienlaptrinh>

336 **Appendix A: Concurrency II**  
336 Phụ lục A: Tương tranh II

**•** Update—Acquire connection to database and then master repository. Read from work in process database and then send to the master repository  
• Cập nhật—Nhận kết nối tới cơ sở dữ liệu và sau đó là kho lưu trữ chính. Đọc từ công việc trong cơ sở dữ liệu quy trình và sau đó gửi đến kho lưu trữ chính

What happens when there are more users than the pool sizes? Consider each pool has a size of ten.  
Điều gì xảy ra khi có nhiều người dùng hơn kích thước nhóm? Hãy xem xét mỗi hồ bơi có kích thước mười.

**•** Ten users attempt to use create, so all ten database connections are acquired, and each thread is interrupted after acquiring a database connection but before acquiring a con-nection to the master repository.  
• Mười người dùng cố gắng sử dụng tạo, vì vậy tất cả mười kết nối cơ sở dữ liệu đều được lấy và mỗi luồng bị gián đoạn sau khi có được kết nối cơ sở dữ liệu nhưng trước khi có được kết nối với kho lưu trữ chính.

**•** Ten users attempt to use update, so all ten master repository connections are acquired, and each thread is interrupted after acquiring the master repository but before acquir-ing a database connection.  
• Mười người dùng cố gắng sử dụng bản cập nhật, vì vậy tất cả mười kết nối kho lưu trữ chính đều có được và mỗi luồng bị gián đoạn sau khi có được kho lưu trữ chính nhưng trước khi có được kết nối cơ sở dữ liệu.

**•** Now the ten “create” threads must wait to acquire a master repository connection, but the ten “update” threads must wait to acquire a database connection.  
• Bây giờ, mười luồng “tạo” phải đợi để có được kết nối kho lưu trữ chính, nhưng mười luồng “cập nhật” phải đợi để có được kết nối cơ sở dữ liệu.

**•** Deadlock. The system never recovers.  
• Bế tắc. Hệ thống không bao giờ phục hồi.

This might sound like an unlikely situation, but who wants a system that freezes solid every other week? Who wants to debug a system with symptoms that are so difﬁcult to reproduce? This is the kind of problem that happens in the ﬁeld, then takes weeks to solve.  
Điều này nghe có vẻ như là một tình huống khó xảy ra, nhưng ai lại muốn một hệ thống bị đóng băng cách hai tuần? Ai muốn gỡ lỗi một hệ thống với các dấu hiệu rất khó tái tạo? Đây là loại vấn đề xảy ra tại hiện trường, sau đó phải mất hàng tuần để giải quyết.

A typical “solution” is to introduce debugging statements to ﬁnd out what is happen-ing. Of course, the debug statements change the code enough so that the deadlock happens in a different situation and takes months to again occur.4  
Một “giải pháp” điển hình là đưa ra các câu lệnh sửa lỗi để tìm hiểu điều gì đang xảy ra. Tất nhiên, các câu lệnh gỡ lỗi thay đổi mã đủ để bế tắc xảy ra trong một tình huống khác và mất nhiều tháng để xảy ra lần nữa.4

To really solve the problem of deadlock, we need to understand what causes it. There are four conditions required for deadlock to occur:  
Để thực sự giải quyết vấn đề bế tắc, chúng ta cần hiểu nguyên nhân gây ra nó. Có bốn điều kiện cần thiết để bế tắc xảy ra:

**•** Mutual exclusion **•** Lock & wait  
• Loại trừ lẫn nhau • Khóa và chờ

**•** No preemption **•** Circular wait  
• Không ưu tiên • Chờ tròn

[**Mutual Exclusion**](#_page_685_0)  
Loại trừ lẫn nhau

Mutual exclusion occurs when multiple threads need to use the same resources and those resources  
Loại trừ lẫn nhau xảy ra khi nhiều luồng cần sử dụng cùng một tài nguyên và các tài nguyên đó

**•** Cannot be used by multiple threads at the same time. **•** Are limited in number.  
• Không thể được sử dụng bởi nhiều luồng cùng một lúc. • Số lượng có hạn.

A common example of such a resource is a database connection, a ﬁle open for write, a record lock, or a semaphore.  
Một ví dụ phổ biến của tài nguyên như vậy là kết nối cơ sở dữ liệu, tệp mở để ghi, khóa bản ghi hoặc semaphore.

4. For example, someone adds some debugging output and the problem “disappears.” The debugging code “ﬁxes” the problem so it remains in the system.  
4. Ví dụ: ai đó thêm một số đầu ra gỡ lỗi và sự cố “biến mất”. Mã gỡ lỗi "khắc phục" sự cố để nó vẫn còn trong hệ thống.

<https://www.facebook.com/groups/phattrienlaptrinh>

**Deadlock** 337  
Bế tắc 337

[**Lock & Wait**](#_page_685_0)  
Khóa & Chờ

Once a thread acquires a resource, it will not release the resource until it has acquired all of the other resources it requires and has completed its work.  
Khi một luồng có được tài nguyên, nó sẽ không giải phóng tài nguyên cho đến khi nó có được tất cả các tài nguyên khác mà nó yêu cầu và đã hoàn thành công việc của mình.

[**No Preemption**](#_page_761_0)  
Không có quyền ưu tiên

One thread cannot take resources away from another thread. Once a thread holds a resource, the only way for another thread to get it is for the holding thread to release it.  
Một luồng không thể lấy tài nguyên từ luồng khác. Khi một luồng nắm giữ tài nguyên, cách duy nhất để một luồng khác lấy được tài nguyên là luồng đang giữ phải giải phóng nó.

[**Circular Wait**](#_page_761_0)  
chờ tròn

This is also referred to as the deadly embrace. Imagine two threads, T1 and T2, and two resources, R1 and R2. T1 has R1, T2 has R2. T1 also requires R2, and T2 also requires R1. This gives something like Figure A-3:  
Đây còn được gọi là cái ôm chết người. Hãy tưởng tượng hai luồng, T1 và T2, và hai tài nguyên, R1 và R2. T1 có R1, T2 có R2. T1 cũng yêu cầu R2 và T2 cũng yêu cầu R1. Điều này cho kết quả giống như Hình A-3:

**Figure A-3  
Hình A-3**

All four of these conditions must hold for deadlock to be possible. Break any one of these conditions and deadlock is not possible.  
Tất cả bốn điều kiện này phải giữ cho bế tắc có thể xảy ra. Phá vỡ bất kỳ một trong những điều kiện này và bế tắc là không thể.

[**Breaking Mutual Exclusion**](#_page_761_0)  
Phá vỡ loại trừ lẫn nhau

One strategy for avoiding deadlock is to sidestep the mutual exclusion condition. You might be able to do this by  
Một chiến lược để tránh bế tắc là vượt qua điều kiện loại trừ lẫn nhau. Bạn có thể làm điều này bằng cách

**•** Using resources that allow simultaneous use, for example, AtomicInteger.  
• Sử dụng các tài nguyên cho phép sử dụng đồng thời, ví dụ AtomicInteger.

**•** Increasing the number of resources such that it equals or exceeds the number of com-peting threads.  
• Tăng số lượng tài nguyên sao cho bằng hoặc vượt quá số lượng luồng cạnh tranh.

**•** Checking that all your resources are free before seizing any.  
• Kiểm tra xem tất cả tài nguyên của bạn có còn miễn phí hay không trước khi thu giữ bất kỳ tài nguyên nào.

Unfortunately, most resources are limited in number and don’t allow simultaneous use. And it’s not uncommon for the identity of the second resource to be predicated on the results of operating on the ﬁrst. But don’t be discouraged; there are three conditions left.  
Thật không may, hầu hết các tài nguyên đều bị giới hạn về số lượng và không cho phép sử dụng đồng thời. Và không có gì lạ khi danh tính của tài nguyên thứ hai được xác định dựa trên kết quả hoạt động của tài nguyên thứ nhất. Nhưng đừng nản lòng; còn ba điều kiện.

<https://www.facebook.com/groups/phattrienlaptrinh>

338 **Appendix A: Concurrency II**  
338 Phụ lục A: Tương tranh II

[**Breaking Lock & Wait**](#_page_761_0)  
Phá Khóa & Đợi

You can also eliminate deadlock if you refuse to wait. Check each resource before you seize it, and release all resources and start over if you run into one that’s busy.  
Bạn cũng có thể loại bỏ bế tắc nếu bạn từ chối chờ đợi. Kiểm tra từng tài nguyên trước khi bạn chiếm lấy nó, đồng thời giải phóng tất cả tài nguyên và bắt đầu lại nếu bạn gặp một tài nguyên đang bận.

This approach introduces several potential problems:  
Cách tiếp cận này giới thiệu một số vấn đề tiềm năng:

**•** Starvation—One thread keeps being unable to acquire the resources it needs (maybe it has a unique combination of resources that seldom all become available).  
• Chết đói—Một luồng liên tục không thể có được các tài nguyên mà nó cần (có thể nó có một tổ hợp tài nguyên duy nhất hiếm khi có sẵn tất cả).

**•** Livelock—Several threads might get into lockstep and all acquire one resource and then release one resource, over and over again. This is especially likely with simplistic CPU scheduling algorithms (think embedded devices or simplistic hand-written thread balancing algorithms).  
• Livelock—Một số luồng có thể đi vào lockstep và tất cả đều có được một tài nguyên và sau đó giải phóng một tài nguyên nhiều lần. Điều này đặc biệt có thể xảy ra với các thuật toán lập lịch CPU đơn giản (nghĩ rằng các thiết bị nhúng hoặc thuật toán cân bằng luồng viết tay đơn giản).

Both of these can cause poor throughput. The ﬁrst results in low CPU utilization, whereas the second results in high and useless CPU utilization.  
Cả hai điều này có thể gây ra thông lượng kém. Kết quả thứ nhất dẫn đến mức sử dụng CPU thấp, trong khi kết quả thứ hai dẫn đến mức sử dụng CPU cao và vô dụng.

As inefﬁcient as this strategy sounds, it’s better than nothing. It has the beneﬁt that it can almost always be implemented if all else fails.  
Dù chiến lược này nghe có vẻ không hiệu quả, nhưng có còn hơn không. Nó có lợi ích là nó hầu như luôn có thể được thực hiện nếu tất cả những cách khác đều thất bại.

[**Breaking Preemption**](#_page_761_0)  
Phá vỡ ưu tiên

Another strategy for avoiding deadlock is to allow threads to take resources away from other threads. This is usually done through a simple request mechanism. When a thread discovers that a resource is busy, it asks the owner to release it. If the owner is also waiting for some other resource, it releases them all and starts over.  
Một chiến lược khác để tránh bế tắc là cho phép các luồng lấy tài nguyên từ các luồng khác. Điều này thường được thực hiện thông qua một cơ chế yêu cầu đơn giản. Khi một luồng phát hiện ra rằng một tài nguyên đang bận, nó sẽ yêu cầu chủ sở hữu giải phóng nó. Nếu chủ sở hữu cũng đang đợi một số tài nguyên khác, nó sẽ giải phóng tất cả chúng và bắt đầu lại.

This is similar to the previous approach but has the beneﬁt that a thread is allowed to wait for a resource. This decreases the number of startovers. Be warned, however, that managing all those requests can be tricky.  
Điều này tương tự như cách tiếp cận trước đó nhưng có lợi ích là một luồng được phép chờ tài nguyên. Điều này làm giảm số lần khởi động. Tuy nhiên, được cảnh báo rằng việc quản lý tất cả các yêu cầu đó có thể phức tạp.

[**Breaking Circular Wait**](#_page_761_0)  
Phá vỡ vòng tròn chờ đợi

This is the most common approach to preventing deadlock. For most systems it requires no more than a simple convention agreed to by all parties.  
Đây là cách tiếp cận phổ biến nhất để ngăn chặn bế tắc. Đối với hầu hết các hệ thống, nó không yêu cầu nhiều hơn một quy ước đơn giản được tất cả các bên đồng ý.

In the example above with Thread 1 wanting both Resource 1 and Resource 2 and Thread 2 wanting both Resource 2 and then Resource 1, simply forcing both Thread 1 and Thread 2 to allocate resources in the same order makes circular wait impossible.  
Trong ví dụ trên với Chủ đề 1 muốn cả Tài nguyên 1 và Tài nguyên 2 và Chủ đề 2 muốn cả Tài nguyên 2 và sau đó là Tài nguyên 1, chỉ cần buộc cả Chủ đề 1 và Chủ đề 2 phân bổ tài nguyên theo cùng một thứ tự khiến cho việc chờ vòng tròn là không thể.

More generally, if all threads can agree on a global ordering of resources and if they all allocate resources in that order, then deadlock is impossible. Like all the other strate-gies, this can cause problems:  
Tổng quát hơn, nếu tất cả các luồng có thể đồng ý về thứ tự toàn cầu của tài nguyên và nếu tất cả chúng phân bổ tài nguyên theo thứ tự đó, thì bế tắc là không thể. Giống như tất cả các chiến lược khác, điều này có thể gây ra vấn đề:

**•** The order of acquisition might not correspond to the order of use; thus a resource acquired at the start might not be used until the end. This can cause resources to be locked longer than strictly necessary.  
• Thứ tự mua có thể không tương ứng với thứ tự sử dụng; do đó, tài nguyên có được khi bắt đầu có thể không được sử dụng cho đến khi kết thúc. Điều này có thể khiến tài nguyên bị khóa lâu hơn mức cần thiết.

<https://www.facebook.com/groups/phattrienlaptrinh>

**Testing Multithreaded Code** 339  
Kiểm tra mã đa luồng 339

**•** Sometimes you cannot impose an order on the acquisition of resources. If the ID of the second resource comes from an operation performed on the ﬁrst, then ordering is not feasible.  
• Đôi khi bạn không thể áp đặt lệnh thu hồi tài nguyên. Nếu ID của tài nguyên thứ hai đến từ thao tác được thực hiện trên tài nguyên đầu tiên, thì việc đặt hàng là không khả thi.

So there are many ways to avoid deadlock. Some lead to starvation, whereas others make heavy use of the CPU and reduce responsiveness. TANSTAAFL!5  
Vì vậy, có nhiều cách để tránh bế tắc. Một số dẫn đến chết đói, trong khi một số khác sử dụng nhiều CPU và giảm khả năng phản hồi. TANSTAAFL!5

Isolating the thread-related part of your solution to allow for tuning and experimenta-tion is a powerful way to gain the insights needed to determine the best strategies.  
Cô lập phần liên quan đến luồng trong giải pháp của bạn để cho phép điều chỉnh và thử nghiệm là một cách hiệu quả để thu được thông tin chi tiết cần thiết nhằm xác định các chiến lược tốt nhất.

[**Testing Multithreaded Code**](#_page_761_0)  
Kiểm tra mã đa luồng

How can we write a test to demonstrate the following code is broken?  
Làm thế nào chúng ta có thể viết một bài kiểm tra để chứng minh đoạn mã sau bị hỏng?

01: public class ClassWithThreadingProblem { 02: int nextId;

03:

04: public int takeNextId() { 05: return nextId++;

06: } 07:}

Here’s a description of a test that will prove the code is broken:  
Dưới đây là mô tả về một thử nghiệm sẽ chứng minh mã bị hỏng:

**•** Remember the current value of nextId.  
• Ghi nhớ giá trị hiện tại của nextId.

**•** Create two threads, both of which call takeNextId() once. **•** Verify that nextId is two more than what we started with.  
• Tạo hai luồng, cả hai đều gọi hàm takeNextId() một lần. • Xác minh rằng nextId nhiều hơn hai so với những gì chúng ta đã bắt đầu.

**•** Run this until we demonstrate that nextId was only incremented by one instead of two.  
• Chạy chương trình này cho đến khi chúng tôi chứng minh rằng nextId chỉ được tăng thêm một thay vì hai.

Listing A-2 shows such a test:  
Liệt kê A-2 cho thấy một thử nghiệm như vậy:

**Listing A-2 ClassWithThreadingProblemTest.java**  
Liệt kê A-2 ClassWithThreadingProblemTest.java

01: package example; 02:

03: import static org.junit.Assert.fail; 04:

05: import org.junit.Test; 06:

07: public class ClassWithThreadingProblemTest { 08: @Test

09: public void twoThreadsShouldFailEventually() throws Exception { 10: final ClassWithThreadingProblem classWithThreadingProblem

= new ClassWithThreadingProblem(); 11:

5. There ain’t no such thing as a free lunch.  
5. Không có gì gọi là bữa trưa miễn phí.

<https://www.facebook.com/groups/phattrienlaptrinh>

340 **Appendix A: Concurrency II**  
340 Phụ lục A: Tương tranh II

**Listing A-2 (continued) ClassWithThreadingProblemTest.java  
Liệt kê A-2 (tiếp theo) ClassWithThreadingProblemTest.java**

12: Runnable runnable = new Runnable() { 13: public void run() {

14: classWithThreadingProblem.takeNextId(); 15: }

16: }; 17:

18: for (int i = 0; i < 50000; ++i) {

19: int startingId = classWithThreadingProblem.lastId; 20: int expectedResult = 2 + startingId;

21:

22: Thread t1 = new Thread(runnable); 23: Thread t2 = new Thread(runnable); 24: t1.start();

25: t2.start(); 26: t1.join(); 27: t2.join(); 28:

29: int endingId = classWithThreadingProblem.lastId; 30:

31: if (endingId != expectedResult) 32: return;

33: } 34:

35: fail("Should have exposed a threading issue but it did not."); 36: }

37: }

**Line** **Description**  
Mô tả dòng

10 Create a single instance of ClassWithThreadingProblem. Note, we must use the ﬁnal keyword because we use it below in an anonymous inner class.  
10 Tạo một phiên bản duy nhất của ClassWithThreadingProblem. Lưu ý, chúng tôi phải sử dụng từ khóa cuối cùng vì chúng tôi sử dụng nó bên dưới trong một lớp bên trong ẩn danh.

12–16 Create an anonymous inner class that uses the single instance of ClassWithThreadingProblem.  
12–16 Tạo một lớp bên trong ẩn danh sử dụng một phiên bản duy nhất của ClassWithThreadingProblem.

18 Run this code “enough” times to demonstrate that the code failed, but not so much that the test “takes too long.” This is a balancing act; we don’t want to wait too long to demonstrate failure. Picking this number is hard— although later we’ll see that we can greatly reduce this number.  
18 Chạy đoạn mã này “đủ” lần để chứng minh rằng đoạn mã đó bị lỗi, nhưng không nhiều đến mức quá trình kiểm tra “mất quá nhiều thời gian”. Đây là một hành động cân bằng; chúng tôi không muốn đợi quá lâu để chứng minh sự thất bại. Chọn con số này thật khó—mặc dù sau này chúng ta sẽ thấy rằng chúng ta có thể giảm con số này đi rất nhiều.

19 Remember the starting value. This test is trying to prove that the code in ClassWithThreadingProblem is broken. If this test passes, it proved that the code was broken. If this test fails, the test was unable to prove that the code is broken.  
19 Ghi nhớ giá trị bắt đầu. Thử nghiệm này nhằm chứng minh rằng mã trong ClassWithThreadingProblem bị hỏng. Nếu thử nghiệm này vượt qua, nó chứng tỏ rằng mã đã bị hỏng. Nếu thử nghiệm này không thành công, thử nghiệm không thể chứng minh rằng mã bị hỏng.

20 We expect the ﬁnal value to be two more than the current value.  
20 Chúng tôi hy vọng giá trị cuối cùng sẽ nhiều hơn hai giá trị hiện tại.

22–23 Create two threads, both of which use the object we created in lines 12–16. This gives us the potential of two threads trying to use our single instance of ClassWithThreadingProblem and interfering with each other.  
22–23 Tạo hai chủ đề, cả hai đều sử dụng đối tượng mà chúng ta đã tạo trong các dòng 12–16. Điều này cho chúng tôi khả năng hai luồng đang cố gắng sử dụng phiên bản ClassWithThreadingProblem duy nhất của chúng tôi và can thiệp lẫn nhau.

<https://www.facebook.com/groups/phattrienlaptrinh>

**Testing Multithreaded Code** 341  
Kiểm tra mã đa luồng 341

**Line** **Description**  
Mô tả dòng

24–25 Make our two threads eligible to run.  
24–25 Làm cho hai chủ đề của chúng tôi đủ điều kiện để chạy.

26–27 Wait for both threads to ﬁnish before we check the results. 29 Record the actual ﬁnal value.  
26–27 Đợi cả hai luồng kết thúc trước khi chúng tôi kiểm tra kết quả. 29 Ghi lại giá trị cuối cùng thực tế.

31–32 Did our endingId differ from what we expected? If so, return end the test— we’ve proven that the code is broken. If not, try again.  
31–32 Phần kết của chúng tôi có khác với những gì chúng tôi mong đợi không? Nếu đúng như vậy, hãy quay lại phần cuối bài kiểm tra— chúng tôi đã chứng minh rằng mã bị hỏng. Nếu không, hãy thử lại.

35 If we got to here, our test was unable to prove the production code was bro-ken in a “reasonable” amount of time; our code has failed. Either the code is not broken or we didn’t run enough iterations to get the failure condition to occur.  
35 Nếu chúng tôi đến đây, thử nghiệm của chúng tôi không thể chứng minh mã sản xuất đã bị hỏng trong một khoảng thời gian “hợp lý”; mã của chúng tôi đã thất bại. Mã không bị hỏng hoặc chúng tôi không chạy đủ số lần lặp để xảy ra tình trạng lỗi.

This test certainly sets up the conditions for a concurrent update problem. However, the problem occurs so infrequently that the vast majority of times this test won’t detect it.  
Thử nghiệm này chắc chắn thiết lập các điều kiện cho sự cố cập nhật đồng thời. Tuy nhiên, sự cố xảy ra không thường xuyên nên phần lớn các lần kiểm tra này sẽ không phát hiện ra sự cố.

Indeed, to truly detect the problem we need to set the number of iterations to over one million. Even then, in ten executions with a loop count of 1,000,000, the problem occurred only once. That means we probably ought to set the iteration count to well over one hun-dred million to get reliable failures. How long are we prepared to wait?  
Thật vậy, để thực sự phát hiện ra vấn đề, chúng ta cần đặt số lần lặp lại trên một triệu. Thậm chí sau đó, trong mười lần thực hiện với số vòng lặp là 1.000.000, sự cố chỉ xảy ra một lần. Điều đó có nghĩa là chúng ta có thể phải đặt số lần lặp lại trên một trăm triệu để nhận được các lỗi đáng tin cậy. Chúng ta sẵn sàng chờ đợi bao lâu?

Even if we tuned the test to get reliable failures on one machine, we’ll probably have to retune the test with different values to demonstrate the failure on another machine, operating system, or version of the JVM.  
Ngay cả khi chúng tôi điều chỉnh thử nghiệm để nhận được các lỗi đáng tin cậy trên một máy, chúng tôi có thể sẽ phải thực hiện lại thử nghiệm với các giá trị khác nhau để chứng minh lỗi trên một máy, hệ điều hành hoặc phiên bản JVM khác.

And this is a *simple* problem. If we cannot demonstrate broken code easily with this problem, how will we ever detect truly complex problems?  
Và đây là một vấn đề đơn giản. Nếu chúng ta không thể chứng minh mã bị hỏng một cách dễ dàng với vấn đề này, làm thế nào chúng ta có thể phát hiện ra các vấn đề thực sự phức tạp?

So what approaches can we take to demonstrate this simple failure? And, more impor-tantly, how can we write tests that will demonstrate failures in more complex code? How will we be able to discover if our code has failures when we do not know where to look?  
Vậy chúng ta có thể thực hiện những cách tiếp cận nào để chứng minh sự thất bại đơn giản này? Và, quan trọng hơn, làm thế nào chúng ta có thể viết các bài kiểm tra để chứng minh các lỗi trong mã phức tạp hơn? Làm cách nào chúng tôi có thể khám phá xem mã của chúng tôi có lỗi hay không khi chúng tôi không biết tìm ở đâu?

Here are a few ideas:  
Dưới đây là một vài ý tưởng:

**• Monte Carlo Testing.** Make tests ﬂexible, so they can be tuned. Then run the test over and over—say on a test server—randomly changing the tuning values. If the tests ever fail, the code is broken. Make sure to start writing those tests early so a continuous integration server starts running them soon. By the way, make sure you carefully log the conditions under which the test failed.  
• Thử nghiệm Monte Carlo. Làm cho các bài kiểm tra trở nên linh hoạt để chúng có thể được điều chỉnh. Sau đó chạy đi chạy lại bài kiểm tra—giả sử trên máy chủ thử nghiệm—thay đổi ngẫu nhiên các giá trị điều chỉnh. Nếu các bài kiểm tra thất bại, mã bị hỏng. Đảm bảo bắt đầu viết các bài kiểm tra đó sớm để máy chủ tích hợp liên tục sớm bắt đầu chạy chúng. Nhân tiện, hãy đảm bảo bạn ghi lại cẩn thận các điều kiện khiến thử nghiệm không thành công.

**•** Run the test on every one of the target deployment platforms. Repeatedly. Continu-ously. The longer the tests run without failure, the more likely that  
• Chạy thử nghiệm trên mọi nền tảng triển khai mục tiêu. Nhiều lần. Liên tục. Các thử nghiệm chạy mà không có lỗi càng lâu thì càng có nhiều khả năng

– The production code is correct or  
– Mã sản xuất đúng hoặc

– The tests aren’t adequate to expose problems.  
– Các bài kiểm tra không đủ để phơi bày các vấn đề.

**•** Run the tests on a machine with varying loads. If you can simulate loads close to a production environment, do so.  
• Chạy thử nghiệm trên máy có tải khác nhau. Nếu bạn có thể mô phỏng tải gần với môi trường sản xuất, hãy làm như vậy.

<https://www.facebook.com/groups/phattrienlaptrinh>

342 **Appendix A: Concurrency II**  
342 Phụ lục A: Tương tranh II

Yet, even if you do all of these things, you still don’t stand a very good chance of ﬁnd-ing threading problems with your code. The most insidious problems are the ones that have such a small cross section that they only occur once in a billion opportunities. Such problems are the terror of complex systems.  
Tuy nhiên, ngay cả khi bạn làm tất cả những điều này, bạn vẫn không có cơ hội tốt để tìm ra các vấn đề về luồng với mã của mình. Những vấn đề ngấm ngầm nhất là những vấn đề có tiết diện nhỏ đến mức chúng chỉ xảy ra một lần trong một tỷ cơ hội. Những vấn đề như vậy là nỗi kinh hoàng của các hệ thống phức tạp.

[**Tool Support for Testing Thread-Based Code**](#_page_761_0)  
Hỗ trợ công cụ để kiểm tra mã dựa trên luồng

IBM has created a tool called ConTest.6 It instruments classes to make it more likely that non-thread-safe code fails.  
IBM đã tạo ra một công cụ gọi là ConTest.6 Công cụ này cung cấp các lớp để làm cho mã không an toàn theo luồng dễ bị lỗi hơn.

We do not have any direct relationship with IBM or the team that developed ConTest. A colleague of ours pointed us to it. We noticed vast improvement in our ability to ﬁnd threading issues after a few minutes of using it.  
Chúng tôi không có bất kỳ mối quan hệ trực tiếp nào với IBM hoặc nhóm phát triển Cuộc thi. Một đồng nghiệp của chúng tôi đã chỉ cho chúng tôi điều đó. Chúng tôi nhận thấy sự cải thiện đáng kể về khả năng tìm các vấn đề về luồng sau vài phút sử dụng.

Here’s an outline of how to use ConTest:  
Dưới đây là phác thảo về cách sử dụng ConTest:

**•** Write tests and production code, making sure there are tests speciﬁcally designed to simulate multiple users under varying loads, as mentioned above.  
• Viết các bài kiểm tra và mã sản xuất, đảm bảo có các bài kiểm tra được thiết kế đặc biệt để mô phỏng nhiều người dùng dưới các mức tải khác nhau, như đã đề cập ở trên.

**•** Instrument test and production code with ConTest. **•** Run the tests.  
• Kiểm tra thiết bị và mã sản xuất với ConTest. • Chạy thử nghiệm.

When we instrumented code with ConTest, our success rate went from roughly one fail-ure in ten million iterations to roughly one failure in *thirty* iterations. Here are the loop values for several runs of the test after instrumentation: 13, 23, 0, 54, 16, 14, 6, 69, 107, 49, 2. So clearly the instrumented classes failed much earlier and with much greater reliability.  
Khi chúng tôi thiết lập mã bằng ConTest, tỷ lệ thành công của chúng tôi đã tăng từ khoảng một lần thất bại trong mười triệu lần lặp lại thành khoảng một lần thất bại trong ba mươi lần lặp lại. Dưới đây là các giá trị vòng lặp cho một số lần chạy thử nghiệm sau khi thiết bị đo: 13, 23, 0, 54, 16, 14, 6, 69, 107, 49, 2. Vì vậy, rõ ràng các lớp thiết bị đã thất bại sớm hơn nhiều và có độ tin cậy cao hơn nhiều.

[**Conclusion**](#_page_761_0)  
Phần kết luận

This chapter has been a very brief sojourn through the large and treacherous territory of concurrent programming. We barely scratched the surface. Our emphasis here was on dis-ciplines to help keep concurrent code clean, but there is much more you should learn if you are going to be writing concurrent systems. We recommend you start with Doug Lea’s wonderful book *Concurrent Programming in Java: Design Principles and Patterns.*7  
Chương này là một cuộc dạo chơi rất ngắn qua lãnh thổ rộng lớn và nguy hiểm của lập trình đồng thời. Chúng tôi hầu như không trầy xước bề mặt. Sự nhấn mạnh của chúng tôi ở đây là về các nguyên tắc để giúp giữ cho mã đồng thời sạch sẽ, nhưng bạn nên tìm hiểu thêm nhiều điều nữa nếu bạn định viết các hệ thống đồng thời. Chúng tôi khuyên bạn nên bắt đầu với cuốn sách tuyệt vời của Doug Lea Lập trình đồng thời trong Java: Nguyên tắc và mẫu thiết kế.7

In this chapter we talked about concurrent update, and the disciplines of clean syn-chronization and locking that can prevent it. We talked about how threads can enhance the throughput of an I/O-bound system and showed the clean techniques for achieving such improvements. We talked about deadlock and the disciplines for preventing it in a clean  
Trong chương này, chúng ta đã nói về cập nhật đồng thời và các nguyên tắc khóa và đồng bộ hóa sạch có thể ngăn chặn điều đó. Chúng tôi đã nói về cách các luồng có thể nâng cao thông lượng của hệ thống liên kết I/O và chỉ ra các kỹ thuật rõ ràng để đạt được những cải tiến như vậy. Chúng tôi đã nói về bế tắc và các nguyên tắc để ngăn chặn nó một cách rõ ràng.

6. [http://www.haifa.ibm.com/projects/veriﬁcation/contest/index.html](http://www.haifa.ibm.com/projects/verification/contest/index.html) 7. See [Lea99] p. 191.

<https://www.facebook.com/groups/phattrienlaptrinh>

**Tutorial: Full Code Examples** 343  
Hướng dẫn: Ví dụ đầy đủ về mã 343

way. Finally, we talked about strategies for exposing concurrent problems by instrumenting your code.  
đường. Cuối cùng, chúng tôi đã nói về các chiến lược để phơi bày các vấn đề đồng thời bằng cách cung cấp mã của bạn.

[**Tutorial: Full Code Examples**](#_page_761_0)  
Hướng dẫn: Ví dụ về mã đầy đủ

[**Client/Server Nonthreaded**](#_page_761_0)  
Máy khách/Máy chủ không phân luồng

**Listing A-3 Server.java**  
Liệt kê A-3 Server.java

package com.objectmentor.clientserver.nonthreaded;

import java.io.IOException; import java.net.ServerSocket; import java.net.Socket;

import java.net.SocketException;

import common.MessageUtils;

public class Server implements Runnable { ServerSocket serverSocket;

volatile boolean keepProcessing = true;

public Server(int port, int millisecondsTimeout) throws IOException { serverSocket = new ServerSocket(port); serverSocket.setSoTimeout(millisecondsTimeout);

}

public void run() { System.out.printf("Server Starting\n");

while (keepProcessing) { try {

System.out.printf("accepting client\n"); Socket socket = serverSocket.accept(); System.out.printf("got client\n"); process(socket);

} catch (Exception e) { handle(e);

} }

}

private void handle(Exception e) {

if (!(e instanceof SocketException)) { e.printStackTrace();

} }

public void stopProcessing() { keepProcessing = false; closeIgnoringException(serverSocket);

}

<https://www.facebook.com/groups/phattrienlaptrinh>

344 **Appendix A: Concurrency II**  
344 Phụ lục A: Tương tranh II

**Listing A-3 (continued) Server.java  
Liệt kê A-3 (tiếp theo) Server.java**

void process(Socket socket) { if (socket == null)

return;

try {

System.out.printf("Server: getting message\n"); String message = MessageUtils.getMessage(socket);

System.out.printf("Server: got message: %s\n", message); Thread.sleep(1000);

System.out.printf("Server: sending reply: %s\n", message); MessageUtils.sendMessage(socket, "Processed: " + message); System.out.printf("Server: sent\n"); closeIgnoringException(socket);

} catch (Exception e) { e.printStackTrace();

}

}

private void closeIgnoringException(Socket socket) { if (socket != null)

try { socket.close();

} catch (IOException ignore) { }

}

private void closeIgnoringException(ServerSocket serverSocket) { if (serverSocket != null)

try { serverSocket.close();

} catch (IOException ignore) { }

} }

**Listing A-4 ClientTest.java**  
Liệt kê A-4 ClientTest.java

package com.objectmentor.clientserver.nonthreaded;

import java.io.IOException; import java.net.ServerSocket; import java.net.Socket;

import java.net.SocketException;

import common.MessageUtils;

public class Server implements Runnable { ServerSocket serverSocket;

volatile boolean keepProcessing = true;

<https://www.facebook.com/groups/phattrienlaptrinh>

**Tutorial: Full Code Examples** 345  
Hướng dẫn: Ví dụ đầy đủ về mã 345

**Listing A-4 (continued) ClientTest.java  
Liệt kê A-4 (tiếp theo) ClientTest.java**

public Server(int port, int millisecondsTimeout) throws IOException { serverSocket = new ServerSocket(port); serverSocket.setSoTimeout(millisecondsTimeout);

}

public void run() { System.out.printf("Server Starting\n");

while (keepProcessing) { try {

System.out.printf("accepting client\n"); Socket socket = serverSocket.accept(); System.out.printf("got client\n"); process(socket);

} catch (Exception e) { handle(e);

} }

}

private void handle(Exception e) {

if (!(e instanceof SocketException)) { e.printStackTrace();

} }

public void stopProcessing() { keepProcessing = false; closeIgnoringException(serverSocket);

}

void process(Socket socket) { if (socket == null)

return;

try {

System.out.printf("Server: getting message\n"); String message = MessageUtils.getMessage(socket);

System.out.printf("Server: got message: %s\n", message); Thread.sleep(1000);

System.out.printf("Server: sending reply: %s\n", message); MessageUtils.sendMessage(socket, "Processed: " + message); System.out.printf("Server: sent\n"); closeIgnoringException(socket);

} catch (Exception e) { e.printStackTrace();

}

}

private void closeIgnoringException(Socket socket) { if (socket != null)

try { socket.close();

<https://www.facebook.com/groups/phattrienlaptrinh>

346 **Appendix A: Concurrency II**  
346 Phụ lục A: Tương tranh II

**Listing A-4 (continued) ClientTest.java**  
Liệt kê A-4 (tiếp theo) ClientTest.java

} catch (IOException ignore) { }

}

private void closeIgnoringException(ServerSocket serverSocket) { if (serverSocket != null)

try { serverSocket.close();

} catch (IOException ignore) { }

} }

**Listing A-5 MessageUtils.java** package common;  
Liệt kê gói A-5 MessageUtils.java phổ biến;

import java.io.IOException; import java.io.InputStream; import java.io.ObjectInputStream; import java.io.ObjectOutputStream; import java.io.OutputStream; import java.net.Socket;

public class MessageUtils {

public static void sendMessage(Socket socket, String message) throws IOException {

OutputStream stream = socket.getOutputStream(); ObjectOutputStream oos = new ObjectOutputStream(stream); oos.writeUTF(message);

oos.flush(); }

public static String getMessage(Socket socket) throws IOException { InputStream stream = socket.getInputStream(); ObjectInputStream ois = new ObjectInputStream(stream);

return ois.readUTF(); }

}

[**Client/Server Using Threads**](#_page_761_0)  
Máy khách/Máy chủ Sử dụng Chủ đề

Changing the server to use threads simply requires a change to the process message (new lines are emphasized to stand out):  
Thay đổi máy chủ để sử dụng các luồng chỉ cần thay đổi thông báo quy trình (các dòng mới được nhấn mạnh để nổi bật):

void process(final Socket socket) { if (socket == null)

return;

**Runnable clientHandler = new Runnable() {** public void run() {

<https://www.facebook.com/groups/phattrienlaptrinh>

**Tutorial: Full Code Examples** 347  
Hướng dẫn: Ví dụ đầy đủ về mã 347

try {

System.out.printf("Server: getting message\n"); String message = MessageUtils.getMessage(socket);

System.out.printf("Server: got message: %s\n", message); Thread.sleep(1000);

System.out.printf("Server: sending reply: %s\n", message); MessageUtils.sendMessage(socket, "Processed: " + message); System.out.printf("Server: sent\n"); closeIgnoringException(socket);

} catch (Exception e) { e.printStackTrace();

} }

};

**Thread clientConnection = new Thread(clientHandler); clientConnection.start();**

}

<https://www.facebook.com/groups/phattrienlaptrinh>

*This page intentionally left blank*

<https://www.facebook.com/groups/phattrienlaptrinh>

[**Appendix B**](#_page_761_0)  
Phụ lục B

[**org.jfree.date.SerialDate**](#_page_761_0)  
org.jfree.date.SerialDate

**Listing B-1 SerialDate.Java**  
Liệt kê B-1 SerialDate.Java

1 /\* ======================================================================== 2 \* JCommon : a free general purpose class library for the Java(tm) platform 3 \* ======================================================================== 4 \*

5 \* (C) Copyright 2000-2005, by Object Refinery Limited and Contributors. 6 \*

7 \* Project Info: http://www.jfree.org/jcommon/index.html 8 \*

9 \* This library is free software; you can redistribute it and/or modify it 10 \* under the terms of the GNU Lesser General Public License as published by 11 \* the Free Software Foundation; either version 2.1 of the License, or

12 \* (at your option) any later version. 13 \*

14 \* This library is distributed in the hope that it will be useful, but

15 \* WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY 16 \* or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public

17 \* License for more details. 18 \*

19 \* You should have received a copy of the GNU Lesser General Public 20 \* License along with this library; if not, write to the Free Software

21 \* Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, 22 \* USA.

23 \*

24 \* [Java is a trademark or registered trademark of Sun Microsystems, Inc. 25 \* in the United States and other countries.]

26 \*

27 \* ---------------28 \* SerialDate.java 29 \* ---------------

30 \* (C) Copyright 2001-2005, by Object Refinery Limited. 31 \*

32 \* Original Author: David Gilbert (for Object Refinery Limited); 33 \* Contributor(s): -;

34 \*

35 \* $Id: SerialDate.java,v 1.7 2005/11/03 09:25:17 mungady Exp $ 36 \*

37 \* Changes (from 11-Oct-2001)

349

<https://www.facebook.com/groups/phattrienlaptrinh>

350 **Appendix B: org.jfree.date.SerialDate**  
350 Phụ lục B: org.jfree.date.SerialDate

**Listing B-1 (continued) SerialDate.Java  
Liệt kê B-1 (tiếp theo) SerialDate.Java**

38 \* --------------------------

39 \* 11-Oct-2001 : Re-organised the class and moved it to new package 40 \* com.jrefinery.date (DG);

41 \* 05-Nov-2001 : Added a getDescription() method, and eliminated NotableDate 42 \* class (DG);

43 \* 12-Nov-2001 : IBD requires setDescription() method, now that NotableDate 44 \* class is gone (DG); Changed getPreviousDayOfWeek(),

45 \* getFollowingDayOfWeek() and getNearestDayOfWeek() to correct 46 \* bugs (DG);

47 \* 05-Dec-2001 : Fixed bug in SpreadsheetDate class (DG);

48 \* 29-May-2002 : Moved the month constants into a separate interface 49 \* (MonthConstants) (DG);

50 \* 27-Aug-2002 : Fixed bug in addMonths() method, thanks to N???levka Petr (DG); 51 \* 03-Oct-2002 : Fixed errors reported by Checkstyle (DG);

52 \* 13-Mar-2003 : Implemented Serializable (DG);

53 \* 29-May-2003 : Fixed bug in addMonths method (DG);

54 \* 04-Sep-2003 : Implemented Comparable. Updated the isInRange javadocs (DG); 55 \* 05-Jan-2005 : Fixed bug in addYears() method (1096282) (DG);

56 \* 57 \*/ 58

59 package org.jfree.date; 60

61 import java.io.Serializable;

62 import java.text.DateFormatSymbols; 63 import java.text.SimpleDateFormat; 64 import java.util.Calendar;

65 import java.util.GregorianCalendar; 66

67 /\*\*

68 \* An abstract class that defines our requirements for manipulating dates, 69 \* without tying down a particular implementation.

70 \* <P>

71 \* Requirement 1 : match at least what Excel does for dates; 72 \* Requirement 2 : class is immutable;

73 \* <P>

74 \* Why not just use java.util.Date? We will, when it makes sense. At times, 75 \* java.util.Date can be \*too\* precise - it represents an instant in time, 76 \* accurate to 1/1000th of a second (with the date itself depending on the 77 \* time-zone). Sometimes we just want to represent a particular day (e.g. 21 78 \* January 2015) without concerning ourselves about the time of day, or the 79 \* time-zone, or anything else. That's what we've defined SerialDate for. 80 \* <P>

81 \* You can call getInstance() to get a concrete subclass of SerialDate, 82 \* without worrying about the exact implementation.

83 \*

84 \* @author David Gilbert 85 \*/

86 public abstract class SerialDate implements Comparable, 87 Serializable,

88 MonthConstants { 89

90 /\*\* For serialization. \*/

91 private static final long serialVersionUID = -293716040467423637L; 92

93 /\*\* Date format symbols. \*/

94 public static final DateFormatSymbols

95 DATE\_FORMAT\_SYMBOLS = new SimpleDateFormat().getDateFormatSymbols(); 96

97 /\*\* The serial number for 1 January 1900. \*/ 98 public static final int SERIAL\_LOWER\_BOUND = 2; 99

100 /\*\* The serial number for 31 December 9999. \*/

101 public static final int SERIAL\_UPPER\_BOUND = 2958465; 102

<https://www.facebook.com/groups/phattrienlaptrinh>

**Appendix B: org.jfree.date.SerialDate** 351  
Phụ lục B: org.jfree.date.SerialDate 351

**Listing B-1 (continued) SerialDate.Java**  
Liệt kê B-1 (tiếp theo) SerialDate.Java

103 /\*\* The lowest year value supported by this date format. \*/ 104 public static final int MINIMUM\_YEAR\_SUPPORTED = 1900;

105

106 /\*\* The highest year value supported by this date format. \*/ 107 public static final int MAXIMUM\_YEAR\_SUPPORTED = 9999;

108

109 /\*\* Useful constant for Monday. Equivalent to java.util.Calendar.MONDAY. \*/ 110 public static final int MONDAY = Calendar.MONDAY;

111

112 /\*\*

113 \* Useful constant for Tuesday. Equivalent to java.util.Calendar.TUESDAY. 114 \*/

115 public static final int TUESDAY = Calendar.TUESDAY; 116

117 /\*\*

118 \* Useful constant for Wednesday. Equivalent to 119 \* java.util.Calendar.WEDNESDAY.

120 \*/

121 public static final int WEDNESDAY = Calendar.WEDNESDAY; 122

123 /\*\*

124 \* Useful constant for Thrusday. Equivalent to java.util.Calendar.THURSDAY. 125 \*/

126 public static final int THURSDAY = Calendar.THURSDAY; 127

128 /\*\* Useful constant for Friday. Equivalent to java.util.Calendar.FRIDAY. \*/ 129 public static final int FRIDAY = Calendar.FRIDAY;

130

131 /\*\*

132 \* Useful constant for Saturday. Equivalent to java.util.Calendar.SATURDAY. 133 \*/

134 public static final int SATURDAY = Calendar.SATURDAY; 135

136 /\*\* Useful constant for Sunday. Equivalent to java.util.Calendar.SUNDAY. \*/ 137 public static final int SUNDAY = Calendar.SUNDAY;

138

139 /\*\* The number of days in each month in non leap years. \*/ 140 static final int[] LAST\_DAY\_OF\_MONTH =

141 {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}; 142

143 /\*\* The number of days in a (non-leap) year up to the end of each month. \*/ 144 static final int[] AGGREGATE\_DAYS\_TO\_END\_OF\_MONTH =

145 {0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334, 365}; 146

147 /\*\* The number of days in a year up to the end of the preceding month. \*/ 148 static final int[] AGGREGATE\_DAYS\_TO\_END\_OF\_PRECEDING\_MONTH =

149 {0, 0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334, 365}; 150

151 /\*\* The number of days in a leap year up to the end of each month. \*/ 152 static final int[] LEAP\_YEAR\_AGGREGATE\_DAYS\_TO\_END\_OF\_MONTH =

153 {0, 31, 60, 91, 121, 152, 182, 213, 244, 274, 305, 335, 366}; 154

155 /\*\*

156 \* The number of days in a leap year up to the end of the preceding month. 157 \*/

158 static final int[]

159 LEAP\_YEAR\_AGGREGATE\_DAYS\_TO\_END\_OF\_PRECEDING\_MONTH =

160 {0, 0, 31, 60, 91, 121, 152, 182, 213, 244, 274, 305, 335, 366}; 161

162 /\*\* A useful constant for referring to the first week in a month. \*/ 163 public static final int FIRST\_WEEK\_IN\_MONTH = 1;

164

<https://www.facebook.com/groups/phattrienlaptrinh>

352 **Appendix B: org.jfree.date.SerialDate**  
352 Phụ lục B: org.jfree.date.SerialDate

**Listing B-1 (continued) SerialDate.Java  
Liệt kê B-1 (tiếp theo) SerialDate.Java**

165 /\*\* A useful constant for referring to the second week in a month. \*/ 166 public static final int SECOND\_WEEK\_IN\_MONTH = 2;

167

168 /\*\* A useful constant for referring to the third week in a month. \*/ 169 public static final int THIRD\_WEEK\_IN\_MONTH = 3;

170

171 /\*\* A useful constant for referring to the fourth week in a month. \*/ 172 public static final int FOURTH\_WEEK\_IN\_MONTH = 4;

173

174 /\*\* A useful constant for referring to the last week in a month. \*/ 175 public static final int LAST\_WEEK\_IN\_MONTH = 0;

176

177 /\*\* Useful range constant. \*/

178 public static final int INCLUDE\_NONE = 0; 179

180 /\*\* Useful range constant. \*/

181 public static final int INCLUDE\_FIRST = 1; 182

183 /\*\* Useful range constant. \*/

184 public static final int INCLUDE\_SECOND = 2; 185

186 /\*\* Useful range constant. \*/

187 public static final int INCLUDE\_BOTH = 3; 188

189 /\*\*

190 \* Useful constant for specifying a day of the week relative to a fixed 191 \* date.

192 \*/

193 public static final int PRECEDING = -1; 194

195 /\*\*

196 \* Useful constant for specifying a day of the week relative to a fixed 197 \* date.

198 \*/

199 public static final int NEAREST = 0; 200

201 /\*\*

202 \* Useful constant for specifying a day of the week relative to a fixed 203 \* date.

204 \*/

205 public static final int FOLLOWING = 1; 206

207 /\*\* A description for the date. \*/ 208 private String description;

209

210 /\*\*

211 \* Default constructor. 212 \*/

213 protected SerialDate() { 214 }

215

216 /\*\*

217 \* Returns <code>true</code> if the supplied integer code represents a 218 \* valid day-of-the-week, and <code>false</code> otherwise.

219 \*

220 \* @param code the code being checked for validity. 221 \*

222 \* @return <code>true</code> if the supplied integer code represents a 223 \* valid day-of-the-week, and <code>false</code> otherwise. 224 \*/

225 public static boolean isValidWeekdayCode(final int code) { 226

<https://www.facebook.com/groups/phattrienlaptrinh>

**Appendix B: org.jfree.date.SerialDate** 353  
Phụ lục B: org.jfree.date.SerialDate 353

**Listing B-1 (continued) SerialDate.Java**  
Liệt kê B-1 (tiếp theo) SerialDate.Java

227 switch(code) { 228 case SUNDAY: 229 case MONDAY: 230 case TUESDAY:

231 case WEDNESDAY: 232 case THURSDAY: 233 case FRIDAY: 234 case SATURDAY: 235 return true; 236 default:

237 return false; 238 }

239

240 } 241

242 /\*\*

243 \* Converts the supplied string to a day of the week. 244 \*

245 \* @param s a string representing the day of the week. 246 \*

247 \* @return <code>-1</code> if the string is not convertable, the day of 248 \* the week otherwise.

249 \*/

250 public static int stringToWeekdayCode(String s) { 251

252 final String[] shortWeekdayNames

253 = DATE\_FORMAT\_SYMBOLS.getShortWeekdays();

254 final String[] weekDayNames = DATE\_FORMAT\_SYMBOLS.getWeekdays(); 255

256 int result = -1; 257 s = s.trim();

258 for (int i = 0; i < weekDayNames.length; i++) { 259 if (s.equals(shortWeekdayNames[i])) {

260 result = i; 261 break;

262 }

263 if (s.equals(weekDayNames[i])) { 264 result = i;

265 break; 266 }

267 }

268 return result; 269

270 } 271

272 /\*\*

273 \* Returns a string representing the supplied day-of-the-week. 274 \* <P>

275 \* Need to find a better approach. 276 \*

277 \* @param weekday the day of the week. 278 \*

279 \* @return a string representing the supplied day-of-the-week. 280 \*/

281 public static String weekdayCodeToString(final int weekday) { 282

283 final String[] weekdays = DATE\_FORMAT\_SYMBOLS.getWeekdays(); 284 return weekdays[weekday];

285

286 } 287

288 /\*\*

<https://www.facebook.com/groups/phattrienlaptrinh>

354 **Appendix B: org.jfree.date.SerialDate**  
354 Phụ lục B: org.jfree.date.SerialDate

**Listing B-1 (continued) SerialDate.Java  
Liệt kê B-1 (tiếp theo) SerialDate.Java**

289 \* Returns an array of month names. 290 \*

291 \* @return an array of month names. 292 \*/

293 public static String[] getMonths() { 294

295 return getMonths(false); 296

297 } 298

299 /\*\*

300 \* Returns an array of month names. 301 \*

302 \* @param shortened a flag indicating that shortened month names should 303 \* be returned.

304 \*

305 \* @return an array of month names. 306 \*/

307 public static String[] getMonths(final boolean shortened) { 308

309 if (shortened) {

310 return DATE\_FORMAT\_SYMBOLS.getShortMonths(); 311 }

312 else {

313 return DATE\_FORMAT\_SYMBOLS.getMonths(); 314 }

315

316 } 317

318 /\*\*

319 \* Returns true if the supplied integer code represents a valid month. 320 \*

321 \* @param code the code being checked for validity. 322 \*

323 \* @return <code>true</code> if the supplied integer code represents a 324 \* valid month.

325 \*/

326 public static boolean isValidMonthCode(final int code) { 327

328 switch(code) { 329 case JANUARY: 330 case FEBRUARY: 331 case MARCH: 332 case APRIL: 333 case MAY:

334 case JUNE: 335 case JULY: 336 case AUGUST:

337 case SEPTEMBER: 338 case OCTOBER: 339 case NOVEMBER: 340 case DECEMBER: 341 return true; 342 default:

343 return false; 344 }

345

346 } 347

348 /\*\*

349 \* Returns the quarter for the specified month. 350 \*

<https://www.facebook.com/groups/phattrienlaptrinh>

**Appendix B: org.jfree.date.SerialDate** 355

**Listing B-1 (continued) SerialDate.Java**

351 \* @param code the month code (1-12). 352 \*

353 \* @return the quarter that the month belongs to. 354 \* @throws java.lang.IllegalArgumentException 355 \*/

356 public static int monthCodeToQuarter(final int code) { 357

358 switch(code) { 359 case JANUARY: 360 case FEBRUARY:

361 case MARCH: return 1; 362 case APRIL:

363 case MAY:

364 case JUNE: return 2; 365 case JULY:

366 case AUGUST:

367 case SEPTEMBER: return 3; 368 case OCTOBER:

369 case NOVEMBER:

370 case DECEMBER: return 4;

371 default: throw new IllegalArgumentException(

372 "SerialDate.monthCodeToQuarter: invalid month code."); 373 }

374

375 } 376

377 /\*\*

378 \* Returns a string representing the supplied month. 379 \* <P>

380 \* The string returned is the long form of the month name taken from the 381 \* default locale.

382 \*

383 \* @param month the month. 384 \*

385 \* @return a string representing the supplied month. 386 \*/

387 public static String monthCodeToString(final int month) { 388

389 return monthCodeToString(month, false); 390

391 } 392

393 /\*\*

394 \* Returns a string representing the supplied month. 395 \* <P>

396 \* The string returned is the long or short form of the month name taken 397 \* from the default locale.

398 \*

399 \* @param month the month.

400 \* @param shortened if <code>true</code> return the abbreviation of the 401 \* month.

402 \*

403 \* @return a string representing the supplied month. 404 \* @throws java.lang.IllegalArgumentException

405 \*/

406 public static String monthCodeToString(final int month,

407 final boolean shortened) { 408

409 // check arguments...

410 if (!isValidMonthCode(month)) {

411 throw new IllegalArgumentException(

412 "SerialDate.monthCodeToString: month outside valid range.");

<https://www.facebook.com/groups/phattrienlaptrinh>

356 **Appendix B: org.jfree.date.SerialDate**

**Listing B-1 (continued) SerialDate.Java**

413 } 414

415 final String[] months; 416

417 if (shortened) {

418 months = DATE\_FORMAT\_SYMBOLS.getShortMonths(); 419 }

420 else {

421 months = DATE\_FORMAT\_SYMBOLS.getMonths(); 422 }

423

424 return months[month - 1]; 425

426 } 427

428 /\*\*

429 \* Converts a string to a month code. 430 \* <P>

431 \* This method will return one of the constants JANUARY, FEBRUARY, ..., 432 \* DECEMBER that corresponds to the string. If the string is not

433 \* recognised, this method returns -1. 434 \*

435 \* @param s the string to parse. 436 \*

437 \* @return <code>-1</code> if the string is not parseable, the month of the 438 \* year otherwise.

439 \*/

440 public static int stringToMonthCode(String s) { 441

442 final String[] shortMonthNames = DATE\_FORMAT\_SYMBOLS.getShortMonths(); 443 final String[] monthNames = DATE\_FORMAT\_SYMBOLS.getMonths();

444

445 int result = -1; 446 s = s.trim(); 447

448 // first try parsing the string as an integer (1-12)... 449 try {

450 result = Integer.parseInt(s); 451 }

452 catch (NumberFormatException e) { 453 // suppress

454 } 455

456 // now search through the month names... 457 if ((result < 1) || (result > 12)) {

458 for (int i = 0; i < monthNames.length; i++) { 459 if (s.equals(shortMonthNames[i])) {

460 result = i + 1; 461 break;

462 }

463 if (s.equals(monthNames[i])) { 464 result = i + 1;

465 break; 466 }

467 } 468 } 469

470 return result; 471

472 } 473

474 /\*\*

<https://www.facebook.com/groups/phattrienlaptrinh>

**Appendix B: org.jfree.date.SerialDate** 357

**Listing B-1 (continued) SerialDate.Java**

475 \* Returns true if the supplied integer code represents a valid 476 \* week-in-the-month, and false otherwise.

477 \*

478 \* @param code the code being checked for validity.

479 \* @return <code>true</code> if the supplied integer code represents a 480 \* valid week-in-the-month.

481 \*/

482 public static boolean isValidWeekInMonthCode(final int code) { 483

484 switch(code) {

485 case FIRST\_WEEK\_IN\_MONTH: 486 case SECOND\_WEEK\_IN\_MONTH: 487 case THIRD\_WEEK\_IN\_MONTH: 488 case FOURTH\_WEEK\_IN\_MONTH:

489 case LAST\_WEEK\_IN\_MONTH: return true; 490 default: return false;

491 } 492

493 } 494

495 /\*\*

496 \* Determines whether or not the specified year is a leap year. 497 \*

498 \* @param yyyy the year (in the range 1900 to 9999). 499 \*

500 \* @return <code>true</code> if the specified year is a leap year. 501 \*/

502 public static boolean isLeapYear(final int yyyy) { 503

504 if ((yyyy % 4) != 0) { 505 return false;

506 }

507 else if ((yyyy % 400) == 0) { 508 return true;

509 }

510 else if ((yyyy % 100) == 0) { 511 return false;

512 }

513 else {

514 return true; 515 }

516

517 } 518

519 /\*\*

520 \* Returns the number of leap years from 1900 to the specified year 521 \* INCLUSIVE.

522 \* <P>

523 \* Note that 1900 is not a leap year. 524 \*

525 \* @param yyyy the year (in the range 1900 to 9999). 526 \*

527 \* @return the number of leap years from 1900 to the specified year. 528 \*/

529 public static int leapYearCount(final int yyyy) { 530

531 final int leap4 = (yyyy - 1896) / 4; 532 final int leap100 = (yyyy - 1800) / 100; 533 final int leap400 = (yyyy - 1600) / 400; 534 return leap4 - leap100 + leap400;

535

536 }

<https://www.facebook.com/groups/phattrienlaptrinh>

358 **Appendix B: org.jfree.date.SerialDate**

**Listing B-1 (continued) SerialDate.Java**

537

538 /\*\*

539 \* Returns the number of the last day of the month, taking into account 540 \* leap years.

541 \*

542 \* @param month the month.

543 \* @param yyyy the year (in the range 1900 to 9999). 544 \*

545 \* @return the number of the last day of the month. 546 \*/

547 public static int lastDayOfMonth(final int month, final int yyyy) { 548

549 final int result = LAST\_DAY\_OF\_MONTH[month]; 550 if (month != FEBRUARY) {

551 return result; 552 }

553 else if (isLeapYear(yyyy)) { 554 return result + 1;

555 }

556 else {

557 return result; 558 }

559

560 } 561

562 /\*\*

563 \* Creates a new date by adding the specified number of days to the base 564 \* date.

565 \*

566 \* @param days the number of days to add (can be negative). 567 \* @param base the base date.

568 \*

569 \* @return a new date. 570 \*/

571 public static SerialDate addDays(final int days, final SerialDate base) { 572

573 final int serialDayNumber = base.toSerial() + days; 574 return SerialDate.createInstance(serialDayNumber); 575

576 } 577

578 /\*\*

579 \* Creates a new date by adding the specified number of months to the base 580 \* date.

581 \* <P>

582 \* If the base date is close to the end of the month, the day on the result 583 \* may be adjusted slightly: 31 May + 1 month = 30 June.

584 \*

585 \* @param months the number of months to add (can be negative). 586 \* @param base the base date.

587 \*

588 \* @return a new date. 589 \*/

590 public static SerialDate addMonths(final int months,

591 final SerialDate base) { 592

593 final int yy = (12 \* base.getYYYY() + base.getMonth() + months - 1) 594 / 12;

595 final int mm = (12 \* base.getYYYY() + base.getMonth() + months - 1) 596 % 12 + 1;

597 final int dd = Math.min(

598 base.getDayOfMonth(), SerialDate.lastDayOfMonth(mm, yy)

<https://www.facebook.com/groups/phattrienlaptrinh>

**Appendix B: org.jfree.date.SerialDate** 359

**Listing B-1 (continued) SerialDate.Java**

599 );

600 return SerialDate.createInstance(dd, mm, yy); 601

602 } 603

604 /\*\*

605 \* Creates a new date by adding the specified number of years to the base 606 \* date.

607 \*

608 \* @param years the number of years to add (can be negative). 609 \* @param base the base date.

610 \*

611 \* @return A new date. 612 \*/

613 public static SerialDate addYears(final int years, final SerialDate base) { 614

615 final int baseY = base.getYYYY(); 616 final int baseM = base.getMonth();

617 final int baseD = base.getDayOfMonth(); 618

619 final int targetY = baseY + years; 620 final int targetD = Math.min(

621 baseD, SerialDate.lastDayOfMonth(baseM, targetY) 622 );

623

624 return SerialDate.createInstance(targetD, baseM, targetY); 625

626 } 627

628 /\*\*

629 \* Returns the latest date that falls on the specified day-of-the-week and 630 \* is BEFORE the base date.

631 \*

632 \* @param targetWeekday a code for the target day-of-the-week. 633 \* @param base the base date.

634 \*

635 \* @return the latest date that falls on the specified day-of-the-week and 636 \* is BEFORE the base date.

637 \*/

638 public static SerialDate getPreviousDayOfWeek(final int targetWeekday, 639 final SerialDate base) { 640

641 // check arguments...

642 if (!SerialDate.isValidWeekdayCode(targetWeekday)) { 643 throw new IllegalArgumentException(

644 "Invalid day-of-the-week code." 645 );

646 } 647

648 // find the date... 649 final int adjust;

650 final int baseDOW = base.getDayOfWeek(); 651 if (baseDOW > targetWeekday) {

652 adjust = Math.min(0, targetWeekday - baseDOW); 653 }

654 else {

655 adjust = -7 + Math.max(0, targetWeekday - baseDOW); 656 }

657

658 return SerialDate.addDays(adjust, base); 659

660 }

<https://www.facebook.com/groups/phattrienlaptrinh>

360 **Appendix B: org.jfree.date.SerialDate**

**Listing B-1 (continued) SerialDate.Java**

661

662 /\*\*

663 \* Returns the earliest date that falls on the specified day-of-the-week 664 \* and is AFTER the base date.

665 \*

666 \* @param targetWeekday a code for the target day-of-the-week. 667 \* @param base the base date.

668 \*

669 \* @return the earliest date that falls on the specified day-of-the-week 670 \* and is AFTER the base date.

671 \*/

672 public static SerialDate getFollowingDayOfWeek(final int targetWeekday, 673 final SerialDate base) { 674

675 // check arguments...

676 if (!SerialDate.isValidWeekdayCode(targetWeekday)) { 677 throw new IllegalArgumentException(

678 "Invalid day-of-the-week code." 679 );

680 } 681

682 // find the date... 683 final int adjust;

684 final int baseDOW = base.getDayOfWeek(); 685 if (baseDOW > targetWeekday) {

686 adjust = 7 + Math.min(0, targetWeekday - baseDOW); 687 }

688 else {

689 adjust = Math.max(0, targetWeekday - baseDOW); 690 }

691

692 return SerialDate.addDays(adjust, base); 693 }

694

695 /\*\*

696 \* Returns the date that falls on the specified day-of-the-week and is 697 \* CLOSEST to the base date.

698 \*

699 \* @param targetDOW a code for the target day-of-the-week. 700 \* @param base the base date.

701 \*

702 \* @return the date that falls on the specified day-of-the-week and is 703 \* CLOSEST to the base date.

704 \*/

705 public static SerialDate getNearestDayOfWeek(final int targetDOW, 706 final SerialDate base) { 707

708 // check arguments...

709 if (!SerialDate.isValidWeekdayCode(targetDOW)) { 710 throw new IllegalArgumentException(

711 "Invalid day-of-the-week code." 712 );

713 } 714

715 // find the date...

716 final int baseDOW = base.getDayOfWeek(); 717 int adjust = -Math.abs(targetDOW - baseDOW); 718 if (adjust >= 4) {

719 adjust = 7 - adjust; 720 }

721 if (adjust <= -4) {

722 adjust = 7 + adjust;

<https://www.facebook.com/groups/phattrienlaptrinh>

**Appendix B: org.jfree.date.SerialDate** 361

**Listing B-1 (continued) SerialDate.Java**

723 }

724 return SerialDate.addDays(adjust, base); 725

726 } 727

728 /\*\*

729 \* Rolls the date forward to the last day of the month. 730 \*

731 \* @param base the base date. 732 \*

733 \* @return a new serial date. 734 \*/

735 public SerialDate getEndOfCurrentMonth(final SerialDate base) { 736 final int last = SerialDate.lastDayOfMonth(

737 base.getMonth(), base.getYYYY() 738 );

739 return SerialDate.createInstance(last, base.getMonth(), base.getYYYY()); 740 }

741

742 /\*\*

743 \* Returns a string corresponding to the week-in-the-month code. 744 \* <P>

745 \* Need to find a better approach. 746 \*

747 \* @param count an integer code representing the week-in-the-month. 748 \*

749 \* @return a string corresponding to the week-in-the-month code. 750 \*/

751 public static String weekInMonthToString(final int count) { 752

753 switch (count) {

754 case SerialDate.FIRST\_WEEK\_IN\_MONTH : return "First"; 755 case SerialDate.SECOND\_WEEK\_IN\_MONTH : return "Second"; 756 case SerialDate.THIRD\_WEEK\_IN\_MONTH : return "Third"; 757 case SerialDate.FOURTH\_WEEK\_IN\_MONTH : return "Fourth"; 758 case SerialDate.LAST\_WEEK\_IN\_MONTH : return "Last"; 759 default :

760 return "SerialDate.weekInMonthToString(): invalid code."; 761 }

762

763 } 764

765 /\*\*

766 \* Returns a string representing the supplied 'relative'. 767 \* <P>

768 \* Need to find a better approach. 769 \*

770 \* @param relative a constant representing the 'relative'. 771 \*

772 \* @return a string representing the supplied 'relative'. 773 \*/

774 public static String relativeToString(final int relative) { 775

776 switch (relative) {

777 case SerialDate.PRECEDING : return "Preceding"; 778 case SerialDate.NEAREST : return "Nearest"; 779 case SerialDate.FOLLOWING : return "Following"; 780 default : return "ERROR : Relative To String"; 781 }

782

783 } 784

<https://www.facebook.com/groups/phattrienlaptrinh>

362 **Appendix B: org.jfree.date.SerialDate**

**Listing B-1 (continued) SerialDate.Java**

785 /\*\*

786 \* Factory method that returns an instance of some concrete subclass of 787 \* {@link SerialDate}.

788 \*

789 \* @param day the day (1-31). 790 \* @param month the month (1-12).

791 \* @param yyyy the year (in the range 1900 to 9999). 792 \*

793 \* @return An instance of {@link SerialDate}. 794 \*/

795 public static SerialDate createInstance(final int day, final int month, 796 final int yyyy) {

797 return new SpreadsheetDate(day, month, yyyy); 798 }

799

800 /\*\*

801 \* Factory method that returns an instance of some concrete subclass of 802 \* {@link SerialDate}.

803 \*

804 \* @param serial the serial number for the day (1 January 1900 = 2). 805 \*

806 \* @return a instance of SerialDate. 807 \*/

808 public static SerialDate createInstance(final int serial) { 809 return new SpreadsheetDate(serial);

810 } 811

812 /\*\*

813 \* Factory method that returns an instance of a subclass of SerialDate. 814 \*

815 \* @param date A Java date object. 816 \*

817 \* @return a instance of SerialDate. 818 \*/

819 public static SerialDate createInstance(final java.util.Date date) { 820

821 final GregorianCalendar calendar = new GregorianCalendar(); 822 calendar.setTime(date);

823 return new SpreadsheetDate(calendar.get(Calendar.DATE),

824 calendar.get(Calendar.MONTH) + 1, 825 calendar.get(Calendar.YEAR)); 826

827 } 828

829 /\*\*

830 \* Returns the serial number for the date, where 1 January 1900 = 2 (this 831 \* corresponds, almost, to the numbering system used in Microsoft Excel for 832 \* Windows and Lotus 1-2-3).

833 \*

834 \* @return the serial number for the date. 835 \*/

836 public abstract int toSerial(); 837

838 /\*\*

839 \* Returns a java.util.Date. Since java.util.Date has more precision than 840 \* SerialDate, we need to define a convention for the 'time of day'.

841 \*

842 \* @return this as <code>java.util.Date</code>. 843 \*/

844 public abstract java.util.Date toDate(); 845

846 /\*\*

<https://www.facebook.com/groups/phattrienlaptrinh>

**Appendix B: org.jfree.date.SerialDate** 363

**Listing B-1 (continued) SerialDate.Java**

847 \* Returns a description of the date. 848 \*

849 \* @return a description of the date. 850 \*/

851 public String getDescription() { 852 return this.description; 853 }

854

855 /\*\*

856 \* Sets the description for the date. 857 \*

858 \* @param description the new description for the date. 859 \*/

860 public void setDescription(final String description) { 861 this.description = description;

862 } 863

864 /\*\*

865 \* Converts the date to a string. 866 \*

867 \* @return a string representation of the date. 868 \*/

869 public String toString() {

870 return getDayOfMonth() + "-" + SerialDate.monthCodeToString(getMonth()) 871 + "-" + getYYYY();

872 } 873

874 /\*\*

875 \* Returns the year (assume a valid range of 1900 to 9999). 876 \*

877 \* @return the year. 878 \*/

879 public abstract int getYYYY(); 880

881 /\*\*

882 \* Returns the month (January = 1, February = 2, March = 3). 883 \*

884 \* @return the month of the year. 885 \*/

886 public abstract int getMonth(); 887

888 /\*\*

889 \* Returns the day of the month. 890 \*

891 \* @return the day of the month. 892 \*/

893 public abstract int getDayOfMonth(); 894

895 /\*\*

896 \* Returns the day of the week. 897 \*

898 \* @return the day of the week. 899 \*/

900 public abstract int getDayOfWeek(); 901

902 /\*\*

903 \* Returns the difference (in days) between this date and the specified 904 \* 'other' date.

905 \* <P>

906 \* The result is positive if this date is after the 'other' date and 907 \* negative if it is before the 'other' date.

908 \*

<https://www.facebook.com/groups/phattrienlaptrinh>

364 **Appendix B: org.jfree.date.SerialDate**

**Listing B-1 (continued) SerialDate.Java**

909 \* @param other the date being compared to. 910 \*

911 \* @return the difference between this and the other date. 912 \*/

913 public abstract int compare(SerialDate other); 914

915 /\*\*

916 \* Returns true if this SerialDate represents the same date as the 917 \* specified SerialDate.

918 \*

919 \* @param other the date being compared to. 920 \*

921 \* @return <code>true</code> if this SerialDate represents the same date as 922 \* the specified SerialDate.

923 \*/

924 public abstract boolean isOn(SerialDate other); 925

926 /\*\*

927 \* Returns true if this SerialDate represents an earlier date compared to 928 \* the specified SerialDate.

929 \*

930 \* @param other The date being compared to. 931 \*

932 \* @return <code>true</code> if this SerialDate represents an earlier date 933 \* compared to the specified SerialDate.

934 \*/

935 public abstract boolean isBefore(SerialDate other); 936

937 /\*\*

938 \* Returns true if this SerialDate represents the same date as the 939 \* specified SerialDate.

940 \*

941 \* @param other the date being compared to. 942 \*

943 \* @return <code>true<code> if this SerialDate represents the same date 944 \* as the specified SerialDate.

945 \*/

946 public abstract boolean isOnOrBefore(SerialDate other); 947

948 /\*\*

949 \* Returns true if this SerialDate represents the same date as the 950 \* specified SerialDate.

951 \*

952 \* @param other the date being compared to. 953 \*

954 \* @return <code>true</code> if this SerialDate represents the same date 955 \* as the specified SerialDate.

956 \*/

957 public abstract boolean isAfter(SerialDate other); 958

959 /\*\*

960 \* Returns true if this SerialDate represents the same date as the 961 \* specified SerialDate.

962 \*

963 \* @param other the date being compared to. 964 \*

965 \* @return <code>true</code> if this SerialDate represents the same date 966 \* as the specified SerialDate.

967 \*/

968 public abstract boolean isOnOrAfter(SerialDate other); 969

970 /\*\*

971 \* Returns <code>true</code> if this {@link SerialDate} is within the

<https://www.facebook.com/groups/phattrienlaptrinh>

**Appendix B: org.jfree.date.SerialDate** 365

**Listing B-1 (continued) SerialDate.Java**

972 \* specified range (INCLUSIVE). The date order of d1 and d2 is not 973 \* important.

974 \*

975 \* @param d1 a boundary date for the range.

976 \* @param d2 the other boundary date for the range. 977 \*

978 \* @return A boolean. 979 \*/

980 public abstract boolean isInRange(SerialDate d1, SerialDate d2); 981

982 /\*\*

983 \* Returns <code>true</code> if this {@link SerialDate} is within the 984 \* specified range (caller specifies whether or not the end-points are 985 \* included). The date order of d1 and d2 is not important.

986 \*

987 \* @param d1 a boundary date for the range.

988 \* @param d2 the other boundary date for the range.

989 \* @param include a code that controls whether or not the start and end 990 \* dates are included in the range.

991 \*

992 \* @return A boolean. 993 \*/

994 public abstract boolean isInRange(SerialDate d1, SerialDate d2, 995 int include);

996

997 /\*\*

998 \* Returns the latest date that falls on the specified day-of-the-week and 999 \* is BEFORE this date.

1000 \*

1001 \* @param targetDOW a code for the target day-of-the-week. 1002 \*

1003 \* @return the latest date that falls on the specified day-of-the-week and 1004 \* is BEFORE this date.

1005 \*/

1006 public SerialDate getPreviousDayOfWeek(final int targetDOW) { 1007 return getPreviousDayOfWeek(targetDOW, this);

1008 } 1009

1010 /\*\*

1011 \* Returns the earliest date that falls on the specified day-of-the-week 1012 \* and is AFTER this date.

1013 \*

1014 \* @param targetDOW a code for the target day-of-the-week. 1015 \*

1016 \* @return the earliest date that falls on the specified day-of-the-week 1017 \* and is AFTER this date.

1018 \*/

1019 public SerialDate getFollowingDayOfWeek(final int targetDOW) { 1020 return getFollowingDayOfWeek(targetDOW, this);

1021 } 1022

1023 /\*\*

1024 \* Returns the nearest date that falls on the specified day-of-the-week. 1025 \*

1026 \* @param targetDOW a code for the target day-of-the-week. 1027 \*

1028 \* @return the nearest date that falls on the specified day-of-the-week. 1029 \*/

1030 public SerialDate getNearestDayOfWeek(final int targetDOW) { 1031 return getNearestDayOfWeek(targetDOW, this);

1032 } 1033

1034 }

<https://www.facebook.com/groups/phattrienlaptrinh>

366 **Appendix B: org.jfree.date.SerialDate**

**Listing B-2 SerialDateTest.java**

1 /\* ======================================================================== 2 \* JCommon : a free general purpose class library for the Java(tm) platform 3 \* ======================================================================== 4 \*

5 \* (C) Copyright 2000-2005, by Object Refinery Limited and Contributors. 6 \*

7 \* Project Info: http://www.jfree.org/jcommon/index.html 8 \*

9 \* This library is free software; you can redistribute it and/or modify it 10 \* under the terms of the GNU Lesser General Public License as published by 11 \* the Free Software Foundation; either version 2.1 of the License, or

12 \* (at your option) any later version. 13 \*

14 \* This library is distributed in the hope that it will be useful, but

15 \* WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY 16 \* or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public

17 \* License for more details. 18 \*

19 \* You should have received a copy of the GNU Lesser General Public 20 \* License along with this library; if not, write to the Free Software

21 \* Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, 22 \* USA.

23 \*

24 \* [Java is a trademark or registered trademark of Sun Microsystems, Inc. 25 \* in the United States and other countries.]

26 \*

27 \* --------------------28 \* SerialDateTests.java 29 \* --------------------

30 \* (C) Copyright 2001-2005, by Object Refinery Limited. 31 \*

32 \* Original Author: David Gilbert (for Object Refinery Limited); 33 \* Contributor(s): -;

34 \*

35 \* $Id: SerialDateTests.java,v 1.6 2005/11/16 15:58:40 taqua Exp $ 36 \*

37 \* Changes 38 \* -------

39 \* 15-Nov-2001 : Version 1 (DG);

40 \* 25-Jun-2002 : Removed unnecessary import (DG);

41 \* 24-Oct-2002 : Fixed errors reported by Checkstyle (DG); 42 \* 13-Mar-2003 : Added serialization test (DG);

43 \* 05-Jan-2005 : Added test for bug report 1096282 (DG); 44 \*

45 \*/ 46

47 package org.jfree.date.junit; 48

49 import java.io.ByteArrayInputStream; 50 import java.io.ByteArrayOutputStream; 51 import java.io.ObjectInput;

52 import java.io.ObjectInputStream; 53 import java.io.ObjectOutput;

54 import java.io.ObjectOutputStream; 55

56 import junit.framework.Test;

57 import junit.framework.TestCase; 58 import junit.framework.TestSuite; 59

60 import org.jfree.date.MonthConstants; 61 import org.jfree.date.SerialDate;

62

<https://www.facebook.com/groups/phattrienlaptrinh>

**Appendix B: org.jfree.date.SerialDate** 367

**Listing B-2 (continued) SerialDateTest.java**

63 /\*\*

64 \* Some JUnit tests for the {@link SerialDate} class. 65 \*/

66 public class SerialDateTests extends TestCase { 67

68 /\*\* Date representing November 9. \*/ 69 private SerialDate nov9Y2001;

70

71 /\*\*

72 \* Creates a new test case. 73 \*

74 \* @param name the name. 75 \*/

76 public SerialDateTests(final String name) { 77 super(name);

78 } 79

80 /\*\*

81 \* Returns a test suite for the JUnit test runner. 82 \*

83 \* @return The test suite. 84 \*/

85 public static Test suite() {

86 return new TestSuite(SerialDateTests.class); 87 }

88

89 /\*\*

90 \* Problem set up. 91 \*/

92 protected void setUp() {

93 this.nov9Y2001 = SerialDate.createInstance(9, MonthConstants.NOVEMBER, 2001); 94 }

95

96 /\*\*

97 \* 9 Nov 2001 plus two months should be 9 Jan 2002. 98 \*/

99 public void testAddMonthsTo9Nov2001() {

100 final SerialDate jan9Y2002 = SerialDate.addMonths(2, this.nov9Y2001); 101 final SerialDate answer = SerialDate.createInstance(9, 1, 2002);

102 assertEquals(answer, jan9Y2002); 103 }

104

105 /\*\*

106 \* A test case for a reported bug, now fixed. 107 \*/

108 public void testAddMonthsTo5Oct2003() {

109 final SerialDate d1 = SerialDate.createInstance(5, MonthConstants.OCTOBER, 2003); 110 final SerialDate d2 = SerialDate.addMonths(2, d1);

111 assertEquals(d2, SerialDate.createInstance(5, MonthConstants.DECEMBER, 2003)); 112 }

113

114 /\*\*

115 \* A test case for a reported bug, now fixed. 116 \*/

117 public void testAddMonthsTo1Jan2003() {

118 final SerialDate d1 = SerialDate.createInstance(1, MonthConstants.JANUARY, 2003); 119 final SerialDate d2 = SerialDate.addMonths(0, d1);

120 assertEquals(d2, d1); 121 }

122

123 /\*\*

124 \* Monday preceding Friday 9 November 2001 should be 5 November.

<https://www.facebook.com/groups/phattrienlaptrinh>

368 **Appendix B: org.jfree.date.SerialDate**

**Listing B-2 (continued) SerialDateTest.java**

125 \*/

126 public void testMondayPrecedingFriday9Nov2001() {

127 SerialDate mondayBefore = SerialDate.getPreviousDayOfWeek( 128 SerialDate.MONDAY, this.nov9Y2001

129 );

130 assertEquals(5, mondayBefore.getDayOfMonth()); 131 }

132

133 /\*\*

134 \* Monday following Friday 9 November 2001 should be 12 November. 135 \*/

136 public void testMondayFollowingFriday9Nov2001() {

137 SerialDate mondayAfter = SerialDate.getFollowingDayOfWeek( 138 SerialDate.MONDAY, this.nov9Y2001

139 );

140 assertEquals(12, mondayAfter.getDayOfMonth()); 141 }

142

143 /\*\*

144 \* Monday nearest Friday 9 November 2001 should be 12 November. 145 \*/

146 public void testMondayNearestFriday9Nov2001() {

147 SerialDate mondayNearest = SerialDate.getNearestDayOfWeek( 148 SerialDate.MONDAY, this.nov9Y2001

149 );

150 assertEquals(12, mondayNearest.getDayOfMonth()); 151 }

152

153 /\*\*

154 \* The Monday nearest to 22nd January 1970 falls on the 19th. 155 \*/

156 public void testMondayNearest22Jan1970() {

157 SerialDate jan22Y1970 = SerialDate.createInstance(22, MonthConstants.JANUARY, 1970); 158 SerialDate mondayNearest=SerialDate.getNearestDayOfWeek(SerialDate.MONDAY, jan22Y1970); 159 assertEquals(19, mondayNearest.getDayOfMonth());

160 } 161

162 /\*\*

163 \* Problem that the conversion of days to strings returns the right result. Actually, this 164 \* result depends on the Locale so this test needs to be modified.

165 \*/

166 public void testWeekdayCodeToString() { 167

168 final String test = SerialDate.weekdayCodeToString(SerialDate.SATURDAY); 169 assertEquals("Saturday", test);

170

171 } 172

173 /\*\*

174 \* Test the conversion of a string to a weekday. Note that this test will fail if the 175 \* default locale doesn't use English weekday names...devise a better test!

176 \*/

177 public void testStringToWeekday() { 178

179 int weekday = SerialDate.stringToWeekdayCode("Wednesday"); 180 assertEquals(SerialDate.WEDNESDAY, weekday);

181

182 weekday = SerialDate.stringToWeekdayCode(" Wednesday "); 183 assertEquals(SerialDate.WEDNESDAY, weekday);

184

<https://www.facebook.com/groups/phattrienlaptrinh>

**Appendix B: org.jfree.date.SerialDate** 369

**Listing B-2 (continued) SerialDateTest.java**

185 weekday = SerialDate.stringToWeekdayCode("Wed"); 186 assertEquals(SerialDate.WEDNESDAY, weekday); 187

188 } 189

190 /\*\*

191 \* Test the conversion of a string to a month. Note that this test will fail if the 192 \* default locale doesn't use English month names...devise a better test!

193 \*/

194 public void testStringToMonthCode() { 195

196 int m = SerialDate.stringToMonthCode("January"); 197 assertEquals(MonthConstants.JANUARY, m);

198

199 m = SerialDate.stringToMonthCode(" January "); 200 assertEquals(MonthConstants.JANUARY, m);

201

202 m = SerialDate.stringToMonthCode("Jan"); 203 assertEquals(MonthConstants.JANUARY, m); 204

205 } 206

207 /\*\*

208 \* Tests the conversion of a month code to a string. 209 \*/

210 public void testMonthCodeToStringCode() { 211

212 final String test = SerialDate.monthCodeToString(MonthConstants.DECEMBER); 213 assertEquals("December", test);

214

215 } 216

217 /\*\*

218 \* 1900 is not a leap year. 219 \*/

220 public void testIsNotLeapYear1900() {

221 assertTrue(!SerialDate.isLeapYear(1900)); 222 }

223

224 /\*\*

225 \* 2000 is a leap year. 226 \*/

227 public void testIsLeapYear2000() {

228 assertTrue(SerialDate.isLeapYear(2000)); 229 }

230

231 /\*\*

232 \* The number of leap years from 1900 up-to-and-including 1899 is 0. 233 \*/

234 public void testLeapYearCount1899() {

235 assertEquals(SerialDate.leapYearCount(1899), 0); 236 }

237

238 /\*\*

239 \* The number of leap years from 1900 up-to-and-including 1903 is 0. 240 \*/

241 public void testLeapYearCount1903() {

242 assertEquals(SerialDate.leapYearCount(1903), 0); 243 }

244

245 /\*\*

246 \* The number of leap years from 1900 up-to-and-including 1904 is 1. 247 \*/

<https://www.facebook.com/groups/phattrienlaptrinh>

370 **Appendix B: org.jfree.date.SerialDate**

**Listing B-2 (continued) SerialDateTest.java**

248 public void testLeapYearCount1904() {

249 assertEquals(SerialDate.leapYearCount(1904), 1); 250 }

251

252 /\*\*

253 \* The number of leap years from 1900 up-to-and-including 1999 is 24. 254 \*/

255 public void testLeapYearCount1999() {

256 assertEquals(SerialDate.leapYearCount(1999), 24); 257 }

258

259 /\*\*

260 \* The number of leap years from 1900 up-to-and-including 2000 is 25. 261 \*/

262 public void testLeapYearCount2000() {

263 assertEquals(SerialDate.leapYearCount(2000), 25); 264 }

265

266 /\*\*

267 \* Serialize an instance, restore it, and check for equality. 268 \*/

269 public void testSerialization() { 270

271 SerialDate d1 = SerialDate.createInstance(15, 4, 2000); 272 SerialDate d2 = null;

273

274 try {

275 ByteArrayOutputStream buffer = new ByteArrayOutputStream(); 276 ObjectOutput out = new ObjectOutputStream(buffer);

277 out.writeObject(d1); 278 out.close();

279

280 ObjectInput in = new ObjectInputStream(

new ByteArrayInputStream(buffer.toByteArray())); 281 d2 = (SerialDate) in.readObject();

282 in.close(); 283 }

284 catch (Exception e) {

285 System.out.println(e.toString()); 286 }

287 assertEquals(d1, d2); 288

289 } 290

291 /\*\*

292 \* A test for bug report 1096282 (now fixed). 293 \*/

294 public void test1096282() {

295 SerialDate d = SerialDate.createInstance(29, 2, 2004); 296 d = SerialDate.addYears(1, d);

297 SerialDate expected = SerialDate.createInstance(28, 2, 2005); 298 assertTrue(d.isOn(expected));

299 } 300

301 /\*\*

302 \* Miscellaneous tests for the addMonths() method. 303 \*/

304 public void testAddMonths() {

305 SerialDate d1 = SerialDate.createInstance(31, 5, 2004); 306

<https://www.facebook.com/groups/phattrienlaptrinh>

**Appendix B: org.jfree.date.SerialDate** 371

**Listing B-2 (continued) SerialDateTest.java**

307 SerialDate d2 = SerialDate.addMonths(1, d1); 308 assertEquals(30, d2.getDayOfMonth());

309 assertEquals(6, d2.getMonth()); 310 assertEquals(2004, d2.getYYYY()); 311

312 SerialDate d3 = SerialDate.addMonths(2, d1); 313 assertEquals(31, d3.getDayOfMonth());

314 assertEquals(7, d3.getMonth()); 315 assertEquals(2004, d3.getYYYY()); 316

317 SerialDate d4 = SerialDate.addMonths(1, SerialDate.addMonths(1, d1)); 318 assertEquals(30, d4.getDayOfMonth());

319 assertEquals(7, d4.getMonth()); 320 assertEquals(2004, d4.getYYYY()); 321 }

322 }

<https://www.facebook.com/groups/phattrienlaptrinh>

372 **Appendix B: org.jfree.date.SerialDate**

**Listing B-3 MonthConstants.java**

1 /\* ======================================================================== 2 \* JCommon : a free general purpose class library for the Java(tm) platform 3 \* ======================================================================== 4 \*

5 \* (C) Copyright 2000-2005, by Object Refinery Limited and Contributors. 6 \*

7 \* Project Info: http://www.jfree.org/jcommon/index.html 8 \*

9 \* This library is free software; you can redistribute it and/or modify it 10 \* under the terms of the GNU Lesser General Public License as published by 11 \* the Free Software Foundation; either version 2.1 of the License, or

12 \* (at your option) any later version. 13 \*

14 \* This library is distributed in the hope that it will be useful, but

15 \* WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY 16 \* or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public

17 \* License for more details. 18 \*

19 \* You should have received a copy of the GNU Lesser General Public 20 \* License along with this library; if not, write to the Free Software

21 \* Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, 22 \* USA.

23 \*

24 \* [Java is a trademark or registered trademark of Sun Microsystems, Inc. 25 \* in the United States and other countries.]

26 \*

27 \* -------------------28 \* MonthConstants.java 29 \* -------------------

30 \* (C) Copyright 2002, 2003, by Object Refinery Limited. 31 \*

32 \* Original Author: David Gilbert (for Object Refinery Limited); 33 \* Contributor(s): -;

34 \*

35 \* $Id: MonthConstants.java,v 1.4 2005/11/16 15:58:40 taqua Exp $ 36 \*

37 \* Changes 38 \* -------

39 \* 29-May-2002 : Version 1 (code moved from SerialDate class) (DG); 40 \*

41 \*/ 42

43 package org.jfree.date; 44

45 /\*\*

46 \* Useful constants for months. Note that these are NOT equivalent to the 47 \* constants defined by java.util.Calendar (where JANUARY=0 and DECEMBER=11). 48 \* <P>

49 \* Used by the SerialDate and RegularTimePeriod classes. 50 \*

51 \* @author David Gilbert 52 \*/

53 public interface MonthConstants { 54

55 /\*\* Constant for January. \*/

56 public static final int JANUARY = 1; 57

58 /\*\* Constant for February. \*/

59 public static final int FEBRUARY = 2; 60

<https://www.facebook.com/groups/phattrienlaptrinh>

**Appendix B: org.jfree.date.SerialDate** 373

**Listing B-3 (continued) MonthConstants.java**

61 /\*\* Constant for March. \*/

62 public static final int MARCH = 3; 63

64 /\*\* Constant for April. \*/

65 public static final int APRIL = 4; 66

67 /\*\* Constant for May. \*/

68 public static final int MAY = 5; 69

70 /\*\* Constant for June. \*/

71 public static final int JUNE = 6; 72

73 /\*\* Constant for July. \*/

74 public static final int JULY = 7; 75

76 /\*\* Constant for August. \*/

77 public static final int AUGUST = 8; 78

79 /\*\* Constant for September. \*/

80 public static final int SEPTEMBER = 9; 81

82 /\*\* Constant for October. \*/

83 public static final int OCTOBER = 10; 84

85 /\*\* Constant for November. \*/

86 public static final int NOVEMBER = 11; 87

88 /\*\* Constant for December. \*/

89 public static final int DECEMBER = 12; 90

91 }

<https://www.facebook.com/groups/phattrienlaptrinh>

374 **Appendix B: org.jfree.date.SerialDate**

**Listing B-4 BobsSerialDateTest.java**

1 package org.jfree.date.junit; 2

3 import junit.framework.TestCase; 4 import org.jfree.date.\*;

5 import static org.jfree.date.SerialDate.\*; 6

7 import java.util.\*; 8

9 public class BobsSerialDateTest extends TestCase { 10

11 public void testIsValidWeekdayCode() throws Exception { 12 for (int day = 1; day <= 7; day++)

13 assertTrue(isValidWeekdayCode(day)); 14 assertFalse(isValidWeekdayCode(0)); 15 assertFalse(isValidWeekdayCode(8)); 16 }

17

18 public void testStringToWeekdayCode() throws Exception { 19

20 assertEquals(-1, stringToWeekdayCode("Hello"));

21 assertEquals(MONDAY, stringToWeekdayCode("Monday")); 22 assertEquals(MONDAY, stringToWeekdayCode("Mon"));

23 //todo assertEquals(MONDAY,stringToWeekdayCode("monday")); 24 // assertEquals(MONDAY,stringToWeekdayCode("MONDAY"));

25 // assertEquals(MONDAY, stringToWeekdayCode("mon")); 26

27 assertEquals(TUESDAY, stringToWeekdayCode("Tuesday")); 28 assertEquals(TUESDAY, stringToWeekdayCode("Tue"));

29 // assertEquals(TUESDAY,stringToWeekdayCode("tuesday")); 30 // assertEquals(TUESDAY,stringToWeekdayCode("TUESDAY")); 31 // assertEquals(TUESDAY, stringToWeekdayCode("tue")); 32 // assertEquals(TUESDAY, stringToWeekdayCode("tues")); 33

34 assertEquals(WEDNESDAY, stringToWeekdayCode("Wednesday")); 35 assertEquals(WEDNESDAY, stringToWeekdayCode("Wed"));

36 // assertEquals(WEDNESDAY,stringToWeekdayCode("wednesday")); 37 // assertEquals(WEDNESDAY,stringToWeekdayCode("WEDNESDAY")); 38 // assertEquals(WEDNESDAY, stringToWeekdayCode("wed"));

39

40 assertEquals(THURSDAY, stringToWeekdayCode("Thursday")); 41 assertEquals(THURSDAY, stringToWeekdayCode("Thu"));

42 // assertEquals(THURSDAY,stringToWeekdayCode("thursday")); 43 // assertEquals(THURSDAY,stringToWeekdayCode("THURSDAY")); 44 // assertEquals(THURSDAY, stringToWeekdayCode("thu"));

45 // assertEquals(THURSDAY, stringToWeekdayCode("thurs")); 46

47 assertEquals(FRIDAY, stringToWeekdayCode("Friday")); 48 assertEquals(FRIDAY, stringToWeekdayCode("Fri"));

49 // assertEquals(FRIDAY,stringToWeekdayCode("friday")); 50 // assertEquals(FRIDAY,stringToWeekdayCode("FRIDAY")); 51 // assertEquals(FRIDAY, stringToWeekdayCode("fri")); 52

53 assertEquals(SATURDAY, stringToWeekdayCode("Saturday")); 54 assertEquals(SATURDAY, stringToWeekdayCode("Sat"));

55 // assertEquals(SATURDAY,stringToWeekdayCode("saturday")); 56 // assertEquals(SATURDAY,stringToWeekdayCode("SATURDAY")); 57 // assertEquals(SATURDAY, stringToWeekdayCode("sat"));

58

59 assertEquals(SUNDAY, stringToWeekdayCode("Sunday")); 60 assertEquals(SUNDAY, stringToWeekdayCode("Sun"));

61 // assertEquals(SUNDAY,stringToWeekdayCode("sunday")); 62 // assertEquals(SUNDAY,stringToWeekdayCode("SUNDAY")); 63 // assertEquals(SUNDAY, stringToWeekdayCode("sun")); 64 }

65

<https://www.facebook.com/groups/phattrienlaptrinh>

**Appendix B: org.jfree.date.SerialDate** 375

**Listing B-4 (continued) BobsSerialDateTest.java**

66 public void testWeekdayCodeToString() throws Exception { 67 assertEquals("Sunday", weekdayCodeToString(SUNDAY)); 68 assertEquals("Monday", weekdayCodeToString(MONDAY)); 69 assertEquals("Tuesday", weekdayCodeToString(TUESDAY));

70 assertEquals("Wednesday", weekdayCodeToString(WEDNESDAY)); 71 assertEquals("Thursday", weekdayCodeToString(THURSDAY)); 72 assertEquals("Friday", weekdayCodeToString(FRIDAY));

73 assertEquals("Saturday", weekdayCodeToString(SATURDAY)); 74 }

75

76 public void testIsValidMonthCode() throws Exception { 77 for (int i = 1; i <= 12; i++)

78 assertTrue(isValidMonthCode(i)); 79 assertFalse(isValidMonthCode(0)); 80 assertFalse(isValidMonthCode(13)); 81 }

82

83 public void testMonthToQuarter() throws Exception { 84 assertEquals(1, monthCodeToQuarter(JANUARY));

85 assertEquals(1, monthCodeToQuarter(FEBRUARY)); 86 assertEquals(1, monthCodeToQuarter(MARCH)); 87 assertEquals(2, monthCodeToQuarter(APRIL)); 88 assertEquals(2, monthCodeToQuarter(MAY));

89 assertEquals(2, monthCodeToQuarter(JUNE)); 90 assertEquals(3, monthCodeToQuarter(JULY)); 91 assertEquals(3, monthCodeToQuarter(AUGUST));

92 assertEquals(3, monthCodeToQuarter(SEPTEMBER)); 93 assertEquals(4, monthCodeToQuarter(OCTOBER)); 94 assertEquals(4, monthCodeToQuarter(NOVEMBER)); 95 assertEquals(4, monthCodeToQuarter(DECEMBER)); 96

97 try {

98 monthCodeToQuarter(-1);

99 fail("Invalid Month Code should throw exception"); 100 } catch (IllegalArgumentException e) {

101 } 102 } 103

104 public void testMonthCodeToString() throws Exception { 105 assertEquals("January", monthCodeToString(JANUARY)); 106 assertEquals("February", monthCodeToString(FEBRUARY)); 107 assertEquals("March", monthCodeToString(MARCH));

108 assertEquals("April", monthCodeToString(APRIL)); 109 assertEquals("May", monthCodeToString(MAY)); 110 assertEquals("June", monthCodeToString(JUNE)); 111 assertEquals("July", monthCodeToString(JULY));

112 assertEquals("August", monthCodeToString(AUGUST));

113 assertEquals("September", monthCodeToString(SEPTEMBER)); 114 assertEquals("October", monthCodeToString(OCTOBER)); 115 assertEquals("November", monthCodeToString(NOVEMBER)); 116 assertEquals("December", monthCodeToString(DECEMBER)); 117

118 assertEquals("Jan", monthCodeToString(JANUARY, true)); 119 assertEquals("Feb", monthCodeToString(FEBRUARY, true)); 120 assertEquals("Mar", monthCodeToString(MARCH, true)); 121 assertEquals("Apr", monthCodeToString(APRIL, true)); 122 assertEquals("May", monthCodeToString(MAY, true));

123 assertEquals("Jun", monthCodeToString(JUNE, true)); 124 assertEquals("Jul", monthCodeToString(JULY, true)); 125 assertEquals("Aug", monthCodeToString(AUGUST, true));

126 assertEquals("Sep", monthCodeToString(SEPTEMBER, true)); 127 assertEquals("Oct", monthCodeToString(OCTOBER, true));

<https://www.facebook.com/groups/phattrienlaptrinh>

376 **Appendix B: org.jfree.date.SerialDate**

**Listing B-4 (continued) BobsSerialDateTest.java**

128 assertEquals("Nov", monthCodeToString(NOVEMBER, true)); 129 assertEquals("Dec", monthCodeToString(DECEMBER, true)); 130

131 try {

132 monthCodeToString(-1);

133 fail("Invalid month code should throw exception"); 134 } catch (IllegalArgumentException e) {

135 } 136

137 } 138

139 public void testStringToMonthCode() throws Exception { 140 assertEquals(JANUARY,stringToMonthCode("1"));

141 assertEquals(FEBRUARY,stringToMonthCode("2")); 142 assertEquals(MARCH,stringToMonthCode("3")); 143 assertEquals(APRIL,stringToMonthCode("4")); 144 assertEquals(MAY,stringToMonthCode("5"));

145 assertEquals(JUNE,stringToMonthCode("6")); 146 assertEquals(JULY,stringToMonthCode("7")); 147 assertEquals(AUGUST,stringToMonthCode("8"));

148 assertEquals(SEPTEMBER,stringToMonthCode("9")); 149 assertEquals(OCTOBER,stringToMonthCode("10")); 150 assertEquals(NOVEMBER, stringToMonthCode("11")); 151 assertEquals(DECEMBER,stringToMonthCode("12")); 152

153 //todo assertEquals(-1, stringToMonthCode("0")); 154 // assertEquals(-1, stringToMonthCode("13")); 155

156 assertEquals(-1,stringToMonthCode("Hello")); 157

158 for (int m = 1; m <= 12; m++) {

159 assertEquals(m, stringToMonthCode(monthCodeToString(m, false))); 160 assertEquals(m, stringToMonthCode(monthCodeToString(m, true))); 161 }

162

163 // assertEquals(1,stringToMonthCode("jan")); 164 // assertEquals(2,stringToMonthCode("feb")); 165 // assertEquals(3,stringToMonthCode("mar")); 166 // assertEquals(4,stringToMonthCode("apr")); 167 // assertEquals(5,stringToMonthCode("may")); 168 // assertEquals(6,stringToMonthCode("jun")); 169 // assertEquals(7,stringToMonthCode("jul")); 170 // assertEquals(8,stringToMonthCode("aug")); 171 // assertEquals(9,stringToMonthCode("sep")); 172 // assertEquals(10,stringToMonthCode("oct")); 173 // assertEquals(11,stringToMonthCode("nov")); 174 // assertEquals(12,stringToMonthCode("dec")); 175

176 // assertEquals(1,stringToMonthCode("JAN")); 177 // assertEquals(2,stringToMonthCode("FEB")); 178 // assertEquals(3,stringToMonthCode("MAR")); 179 // assertEquals(4,stringToMonthCode("APR")); 180 // assertEquals(5,stringToMonthCode("MAY")); 181 // assertEquals(6,stringToMonthCode("JUN")); 182 // assertEquals(7,stringToMonthCode("JUL")); 183 // assertEquals(8,stringToMonthCode("AUG")); 184 // assertEquals(9,stringToMonthCode("SEP")); 185 // assertEquals(10,stringToMonthCode("OCT")); 186 // assertEquals(11,stringToMonthCode("NOV")); 187 // assertEquals(12,stringToMonthCode("DEC")); 188

189 // assertEquals(1,stringToMonthCode("january")); 190 // assertEquals(2,stringToMonthCode("february"));

<https://www.facebook.com/groups/phattrienlaptrinh>

**Appendix B: org.jfree.date.SerialDate** 377

**Listing B-4 (continued) BobsSerialDateTest.java**

191 // assertEquals(3,stringToMonthCode("march")); 192 // assertEquals(4,stringToMonthCode("april")); 193 // assertEquals(5,stringToMonthCode("may")); 194 // assertEquals(6,stringToMonthCode("june")); 195 // assertEquals(7,stringToMonthCode("july")); 196 // assertEquals(8,stringToMonthCode("august"));

197 // assertEquals(9,stringToMonthCode("september")); 198 // assertEquals(10,stringToMonthCode("october")); 199 // assertEquals(11,stringToMonthCode("november")); 200 // assertEquals(12,stringToMonthCode("december")); 201

202 // assertEquals(1,stringToMonthCode("JANUARY")); 203 // assertEquals(2,stringToMonthCode("FEBRUARY")); 204 // assertEquals(3,stringToMonthCode("MAR"));

205 // assertEquals(4,stringToMonthCode("APRIL")); 206 // assertEquals(5,stringToMonthCode("MAY")); 207 // assertEquals(6,stringToMonthCode("JUNE")); 208 // assertEquals(7,stringToMonthCode("JULY")); 209 // assertEquals(8,stringToMonthCode("AUGUST"));

210 // assertEquals(9,stringToMonthCode("SEPTEMBER")); 211 // assertEquals(10,stringToMonthCode("OCTOBER")); 212 // assertEquals(11,stringToMonthCode("NOVEMBER")); 213 // assertEquals(12,stringToMonthCode("DECEMBER")); 214 }

215

216 public void testIsValidWeekInMonthCode() throws Exception { 217 for (int w = 0; w <= 4; w++) {

218 assertTrue(isValidWeekInMonthCode(w)); 219 }

220 assertFalse(isValidWeekInMonthCode(5)); 221 }

222

223 public void testIsLeapYear() throws Exception { 224 assertFalse(isLeapYear(1900));

225 assertFalse(isLeapYear(1901)); 226 assertFalse(isLeapYear(1902)); 227 assertFalse(isLeapYear(1903)); 228 assertTrue(isLeapYear(1904)); 229 assertTrue(isLeapYear(1908)); 230 assertFalse(isLeapYear(1955)); 231 assertTrue(isLeapYear(1964)); 232 assertTrue(isLeapYear(1980)); 233 assertTrue(isLeapYear(2000)); 234 assertFalse(isLeapYear(2001)); 235 assertFalse(isLeapYear(2100)); 236 }

237

238 public void testLeapYearCount() throws Exception { 239 assertEquals(0, leapYearCount(1900));

240 assertEquals(0, leapYearCount(1901)); 241 assertEquals(0, leapYearCount(1902)); 242 assertEquals(0, leapYearCount(1903)); 243 assertEquals(1, leapYearCount(1904)); 244 assertEquals(1, leapYearCount(1905)); 245 assertEquals(1, leapYearCount(1906)); 246 assertEquals(1, leapYearCount(1907)); 247 assertEquals(2, leapYearCount(1908)); 248 assertEquals(24, leapYearCount(1999)); 249 assertEquals(25, leapYearCount(2001)); 250 assertEquals(49, leapYearCount(2101)); 251 assertEquals(73, leapYearCount(2201));

<https://www.facebook.com/groups/phattrienlaptrinh>

378 **Appendix B: org.jfree.date.SerialDate**

**Listing B-4 (continued) BobsSerialDateTest.java**

252 assertEquals(97, leapYearCount(2301)); 253 assertEquals(122, leapYearCount(2401)); 254 }

255

256 public void testLastDayOfMonth() throws Exception { 257 assertEquals(31, lastDayOfMonth(JANUARY, 1901)); 258 assertEquals(28, lastDayOfMonth(FEBRUARY, 1901)); 259 assertEquals(31, lastDayOfMonth(MARCH, 1901)); 260 assertEquals(30, lastDayOfMonth(APRIL, 1901)); 261 assertEquals(31, lastDayOfMonth(MAY, 1901));

262 assertEquals(30, lastDayOfMonth(JUNE, 1901)); 263 assertEquals(31, lastDayOfMonth(JULY, 1901)); 264 assertEquals(31, lastDayOfMonth(AUGUST, 1901));

265 assertEquals(30, lastDayOfMonth(SEPTEMBER, 1901)); 266 assertEquals(31, lastDayOfMonth(OCTOBER, 1901)); 267 assertEquals(30, lastDayOfMonth(NOVEMBER, 1901)); 268 assertEquals(31, lastDayOfMonth(DECEMBER, 1901)); 269 assertEquals(29, lastDayOfMonth(FEBRUARY, 1904)); 270 }

271

272 public void testAddDays() throws Exception { 273 SerialDate newYears = d(1, JANUARY, 1900);

274 assertEquals(d(2, JANUARY, 1900), addDays(1, newYears)); 275 assertEquals(d(1, FEBRUARY, 1900), addDays(31, newYears)); 276 assertEquals(d(1, JANUARY, 1901), addDays(365, newYears));

277 assertEquals(d(31, DECEMBER, 1904), addDays(5 \* 365, newYears)); 278 }

279

280 private static SpreadsheetDate d(int day, int month, int year) {return new SpreadsheetDate(day, month, year);}

281

282 public void testAddMonths() throws Exception {

283 assertEquals(d(1, FEBRUARY, 1900), addMonths(1, d(1, JANUARY, 1900))); 284 assertEquals(d(28, FEBRUARY, 1900), addMonths(1, d(31, JANUARY, 1900))); 285 assertEquals(d(28, FEBRUARY, 1900), addMonths(1, d(30, JANUARY, 1900))); 286 assertEquals(d(28, FEBRUARY, 1900), addMonths(1, d(29, JANUARY, 1900))); 287 assertEquals(d(28, FEBRUARY, 1900), addMonths(1, d(28, JANUARY, 1900))); 288 assertEquals(d(27, FEBRUARY, 1900), addMonths(1, d(27, JANUARY, 1900))); 289

290 assertEquals(d(30, JUNE, 1900), addMonths(5, d(31, JANUARY, 1900))); 291 assertEquals(d(30, JUNE, 1901), addMonths(17, d(31, JANUARY, 1900))); 292

293 assertEquals(d(29, FEBRUARY, 1904), addMonths(49, d(31, JANUARY, 1900))); 294

295 } 296

297 public void testAddYears() throws Exception {

298 assertEquals(d(1, JANUARY, 1901), addYears(1, d(1, JANUARY, 1900))); 299 assertEquals(d(28, FEBRUARY, 1905), addYears(1, d(29, FEBRUARY, 1904))); 300 assertEquals(d(28, FEBRUARY, 1905), addYears(1, d(28, FEBRUARY, 1904))); 301 assertEquals(d(28, FEBRUARY, 1904), addYears(1, d(28, FEBRUARY, 1903))); 302 }

303

304 public void testGetPreviousDayOfWeek() throws Exception {

305 assertEquals(d(24, FEBRUARY, 2006), getPreviousDayOfWeek(FRIDAY, d(1, MARCH, 2006))); 306 assertEquals(d(22, FEBRUARY, 2006), getPreviousDayOfWeek(WEDNESDAY, d(1, MARCH, 2006))); 307 assertEquals(d(29, FEBRUARY, 2004), getPreviousDayOfWeek(SUNDAY, d(3, MARCH, 2004)));

308 assertEquals(d(29, DECEMBER, 2004), getPreviousDayOfWeek(WEDNESDAY, d(5, JANUARY, 2005))); 309

310 try {

311 getPreviousDayOfWeek(-1, d(1, JANUARY, 2006));

312 fail("Invalid day of week code should throw exception");

<https://www.facebook.com/groups/phattrienlaptrinh>

**Appendix B: org.jfree.date.SerialDate** 379

**Listing B-4 (continued) BobsSerialDateTest.java**

313 } catch (IllegalArgumentException e) { 314 }

315 } 316

317 public void testGetFollowingDayOfWeek() throws Exception {

318 // assertEquals(d(1, JANUARY, 2005),getFollowingDayOfWeek(SATURDAY, d(25, DECEMBER, 2004))); 319 assertEquals(d(1, JANUARY, 2005), getFollowingDayOfWeek(SATURDAY, d(26, DECEMBER, 2004))); 320 assertEquals(d(3, MARCH, 2004), getFollowingDayOfWeek(WEDNESDAY, d(28, FEBRUARY, 2004))); 321

322 try {

323 getFollowingDayOfWeek(-1, d(1, JANUARY, 2006));

324 fail("Invalid day of week code should throw exception"); 325 } catch (IllegalArgumentException e) {

326 } 327 } 328

329 public void testGetNearestDayOfWeek() throws Exception {

330 assertEquals(d(16, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(16, APRIL, 2006))); 331 assertEquals(d(16, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(17, APRIL, 2006))); 332 assertEquals(d(16, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(18, APRIL, 2006))); 333 assertEquals(d(16, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(19, APRIL, 2006))); 334 assertEquals(d(23, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(20, APRIL, 2006))); 335 assertEquals(d(23, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(21, APRIL, 2006))); 336 assertEquals(d(23, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(22, APRIL, 2006))); 337

338 //todo assertEquals(d(17, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(16, APRIL, 2006))); 339 assertEquals(d(17, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(17, APRIL, 2006)));

340 assertEquals(d(17, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(18, APRIL, 2006))); 341 assertEquals(d(17, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(19, APRIL, 2006))); 342 assertEquals(d(17, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(20, APRIL, 2006))); 343 assertEquals(d(24, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(21, APRIL, 2006))); 344 assertEquals(d(24, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(22, APRIL, 2006))); 345

346 // assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(16, APRIL, 2006))); 347 // assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(17, APRIL, 2006))); 348 assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(18, APRIL, 2006))); 349 assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(19, APRIL, 2006))); 350 assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(20, APRIL, 2006))); 351 assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(21, APRIL, 2006))); 352 assertEquals(d(25, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(22, APRIL, 2006))); 353

354 // assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(16, APRIL, 2006))); 355 // assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(17, APRIL, 2006))); 356 // assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(18, APRIL, 2006))); 357 assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(19, APRIL, 2006))); 358 assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(20, APRIL, 2006))); 359 assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(21, APRIL, 2006))); 360 assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(22, APRIL, 2006))); 361

362 // assertEquals(d(13, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(16, APRIL, 2006))); 363 // assertEquals(d(20, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(17, APRIL, 2006))); 364 // assertEquals(d(20, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(18, APRIL, 2006))); 365 // assertEquals(d(20, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(19, APRIL, 2006))); 366 assertEquals(d(20, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(20, APRIL, 2006))); 367 assertEquals(d(20, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(21, APRIL, 2006))); 368 assertEquals(d(20, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(22, APRIL, 2006))); 369

370 // assertEquals(d(14, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(16, APRIL, 2006))); 371 // assertEquals(d(14, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(17, APRIL, 2006))); 372 // assertEquals(d(21, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(18, APRIL, 2006))); 373 // assertEquals(d(21, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(19, APRIL, 2006))); 374 // assertEquals(d(21, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(20, APRIL, 2006)));

<https://www.facebook.com/groups/phattrienlaptrinh>

380 **Appendix B: org.jfree.date.SerialDate**

**Listing B-4 (continued) BobsSerialDateTest.java**

375 assertEquals(d(21, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(21, APRIL, 2006))); 376 assertEquals(d(21, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(22, APRIL, 2006))); 377

378 // assertEquals(d(15, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(16, APRIL, 2006))); 379 // assertEquals(d(15, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(17, APRIL, 2006))); 380 // assertEquals(d(15, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(18, APRIL, 2006))); 381 // assertEquals(d(22, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(19, APRIL, 2006))); 382 // assertEquals(d(22, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(20, APRIL, 2006))); 383 // assertEquals(d(22, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(21, APRIL, 2006))); 384 assertEquals(d(22, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(22, APRIL, 2006))); 385

386 try {

387 getNearestDayOfWeek(-1, d(1, JANUARY, 2006));

388 fail("Invalid day of week code should throw exception"); 389 } catch (IllegalArgumentException e) {

390 } 391 } 392

393 public void testEndOfCurrentMonth() throws Exception { 394 SerialDate d = SerialDate.createInstance(2);

395 assertEquals(d(31, JANUARY, 2006), d.getEndOfCurrentMonth(d(1, JANUARY, 2006))); 396 assertEquals(d(28, FEBRUARY, 2006), d.getEndOfCurrentMonth(d(1, FEBRUARY, 2006))); 397 assertEquals(d(31, MARCH, 2006), d.getEndOfCurrentMonth(d(1, MARCH, 2006)));

398 assertEquals(d(30, APRIL, 2006), d.getEndOfCurrentMonth(d(1, APRIL, 2006))); 399 assertEquals(d(31, MAY, 2006), d.getEndOfCurrentMonth(d(1, MAY, 2006))); 400 assertEquals(d(30, JUNE, 2006), d.getEndOfCurrentMonth(d(1, JUNE, 2006))); 401 assertEquals(d(31, JULY, 2006), d.getEndOfCurrentMonth(d(1, JULY, 2006)));

402 assertEquals(d(31, AUGUST, 2006), d.getEndOfCurrentMonth(d(1, AUGUST, 2006)));

403 assertEquals(d(30, SEPTEMBER, 2006), d.getEndOfCurrentMonth(d(1, SEPTEMBER, 2006))); 404 assertEquals(d(31, OCTOBER, 2006), d.getEndOfCurrentMonth(d(1, OCTOBER, 2006))); 405 assertEquals(d(30, NOVEMBER, 2006), d.getEndOfCurrentMonth(d(1, NOVEMBER, 2006))); 406 assertEquals(d(31, DECEMBER, 2006), d.getEndOfCurrentMonth(d(1, DECEMBER, 2006))); 407 assertEquals(d(29, FEBRUARY, 2008), d.getEndOfCurrentMonth(d(1, FEBRUARY, 2008))); 408 }

409

410 public void testWeekInMonthToString() throws Exception {

411 assertEquals("First",weekInMonthToString(FIRST\_WEEK\_IN\_MONTH)); 412 assertEquals("Second",weekInMonthToString(SECOND\_WEEK\_IN\_MONTH)); 413 assertEquals("Third",weekInMonthToString(THIRD\_WEEK\_IN\_MONTH)); 414 assertEquals("Fourth",weekInMonthToString(FOURTH\_WEEK\_IN\_MONTH)); 415 assertEquals("Last",weekInMonthToString(LAST\_WEEK\_IN\_MONTH)); 416

417 //todo try {

418 // weekInMonthToString(-1);

419 // fail("Invalid week code should throw exception"); 420 // } catch (IllegalArgumentException e) {

421 // } 422 } 423

424 public void testRelativeToString() throws Exception { 425 assertEquals("Preceding",relativeToString(PRECEDING)); 426 assertEquals("Nearest",relativeToString(NEAREST)); 427 assertEquals("Following",relativeToString(FOLLOWING)); 428

429 //todo try {

430 // relativeToString(-1000);

431 // fail("Invalid relative code should throw exception"); 432 // } catch (IllegalArgumentException e) {

433 // } 434 } 435

<https://www.facebook.com/groups/phattrienlaptrinh>

**Appendix B: org.jfree.date.SerialDate** 381

**Listing B-4 (continued) BobsSerialDateTest.java**

436 public void testCreateInstanceFromDDMMYYY() throws Exception { 437 SerialDate date = createInstance(1, JANUARY, 1900);

438 assertEquals(1,date.getDayOfMonth()); 439 assertEquals(JANUARY,date.getMonth()); 440 assertEquals(1900,date.getYYYY()); 441 assertEquals(2,date.toSerial());

442 } 443

444 public void testCreateInstanceFromSerial() throws Exception { 445 assertEquals(d(1, JANUARY, 1900),createInstance(2));

446 assertEquals(d(1, JANUARY, 1901), createInstance(367)); 447 }

448

449 public void testCreateInstanceFromJavaDate() throws Exception { 450 assertEquals(d(1, JANUARY, 1900),

createInstance(new GregorianCalendar(1900,0,1).getTime())); 451 assertEquals(d(1, JANUARY, 2006),

createInstance(new GregorianCalendar(2006,0,1).getTime())); 452 }

453

454 public static void main(String[] args) {

455 junit.textui.TestRunner.run(BobsSerialDateTest.class); 456 }

457 }

<https://www.facebook.com/groups/phattrienlaptrinh>

382 **Appendix B: org.jfree.date.SerialDate**

**Listing B-5 SpreadsheetDate.java**

1 /\* ======================================================================== 2 \* JCommon : a free general purpose class library for the Java(tm) platform 3 \* ======================================================================== 4 \*

5 \* (C) Copyright 2000-2005, by Object Refinery Limited and Contributors. 6 \*

7 \* Project Info: http://www.jfree.org/jcommon/index.html 8 \*

9 \* This library is free software; you can redistribute it and/or modify it 10 \* under the terms of the GNU Lesser General Public License as published by 11 \* the Free Software Foundation; either version 2.1 of the License, or

12 \* (at your option) any later version. 13 \*

14 \* This library is distributed in the hope that it will be useful, but

15 \* WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY 16 \* or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public

17 \* License for more details. 18 \*

19 \* You should have received a copy of the GNU Lesser General Public 20 \* License along with this library; if not, write to the Free Software

21 \* Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, 22 \* USA.

23 \*

24 \* [Java is a trademark or registered trademark of Sun Microsystems, Inc. 25 \* in the United States and other countries.]

26 \*

27 \* --------------------28 \* SpreadsheetDate.java 29 \* --------------------

30 \* (C) Copyright 2000-2005, by Object Refinery Limited and Contributors. 31 \*

32 \* Original Author: David Gilbert (for Object Refinery Limited); 33 \* Contributor(s): -;

34 \*

35 \* $Id: SpreadsheetDate.java,v 1.8 2005/11/03 09:25:39 mungady Exp $ 36 \*

37 \* Changes 38 \* -------

39 \* 11-Oct-2001 : Version 1 (DG);

40 \* 05-Nov-2001 : Added getDescription() and setDescription() methods (DG);

41 \* 12-Nov-2001 : Changed name from ExcelDate.java to SpreadsheetDate.java (DG); 42 \* Fixed a bug in calculating day, month and year from serial

43 \* number (DG);

44 \* 24-Jan-2002 : Fixed a bug in calculating the serial number from the day, 45 \* month and year. Thanks to Trevor Hills for the report (DG); 46 \* 29-May-2002 : Added equals(Object) method (SourceForge ID 558850) (DG); 47 \* 03-Oct-2002 : Fixed errors reported by Checkstyle (DG);

48 \* 13-Mar-2003 : Implemented Serializable (DG);

49 \* 04-Sep-2003 : Completed isInRange() methods (DG); 50 \* 05-Sep-2003 : Implemented Comparable (DG);

51 \* 21-Oct-2003 : Added hashCode() method (DG); 52 \*

53 \*/ 54

55 package org.jfree.date; 56

57 import java.util.Calendar; 58 import java.util.Date;

59

60 /\*\*

61 \* Represents a date using an integer, in a similar fashion to the

62 \* implementation in Microsoft Excel. The range of dates supported is

<https://www.facebook.com/groups/phattrienlaptrinh>

**Appendix B: org.jfree.date.SerialDate** 383

**Listing B-5 (continued) SpreadsheetDate.java**

63 \* 1-Jan-1900 to 31-Dec-9999. 64 \* <P>

65 \* Be aware that there is a deliberate bug in Excel that recognises the year 66 \* 1900 as a leap year when in fact it is not a leap year. You can find more 67 \* information on the Microsoft website in article Q181370:

68 \* <P>

69 \* http://support.microsoft.com/support/kb/articles/Q181/3/70.asp 70 \* <P>

71 \* Excel uses the convention that 1-Jan-1900 = 1. This class uses the 72 \* convention 1-Jan-1900 = 2.

73 \* The result is that the day number in this class will be different to the

74 \* Excel figure for January and February 1900...but then Excel adds in an extra 75 \* day (29-Feb-1900 which does not actually exist!) and from that point forward 76 \* the day numbers will match.

77 \*

78 \* @author David Gilbert 79 \*/

80 public class SpreadsheetDate extends SerialDate { 81

82 /\*\* For serialization. \*/

83 private static final long serialVersionUID = -2039586705374454461L; 84

85 /\*\*

86 \* The day number (1-Jan-1900 = 2, 2-Jan-1900 = 3, ..., 31-Dec-9999 = 87 \* 2958465).

88 \*/

89 private int serial; 90

91 /\*\* The day of the month (1 to 28, 29, 30 or 31 depending on the month). \*/ 92 private int day;

93

94 /\*\* The month of the year (1 to 12). \*/ 95 private int month;

96

97 /\*\* The year (1900 to 9999). \*/ 98 private int year;

99

100 /\*\* An optional description for the date. \*/ 101 private String description;

102

103 /\*\*

104 \* Creates a new date instance. 105 \*

106 \* @param day the day (in the range 1 to 28/29/30/31). 107 \* @param month the month (in the range 1 to 12).

108 \* @param year the year (in the range 1900 to 9999). 109 \*/

110 public SpreadsheetDate(final int day, final int month, final int year) { 111

112 if ((year >= 1900) && (year <= 9999)) { 113 this.year = year;

114 }

115 else {

116 throw new IllegalArgumentException(

117 "The 'year' argument must be in range 1900 to 9999." 118 );

119 } 120

121 if ((month >= MonthConstants.JANUARY)

122 && (month <= MonthConstants.DECEMBER)) { 123 this.month = month;

124 }

<https://www.facebook.com/groups/phattrienlaptrinh>

384 **Appendix B: org.jfree.date.SerialDate**

**Listing B-5 (continued) SpreadsheetDate.java**

125 else {

126 throw new IllegalArgumentException(

127 "The 'month' argument must be in the range 1 to 12." 128 );

129 } 130

131 if ((day >= 1) && (day <= SerialDate.lastDayOfMonth(month, year))) { 132 this.day = day;

133 }

134 else {

135 throw new IllegalArgumentException("Invalid 'day' argument."); 136 }

137

138 // the serial number needs to be synchronised with the day-month-year... 139 this.serial = calcSerial(day, month, year);

140

141 this.description = null; 142

143 } 144

145 /\*\*

146 \* Standard constructor - creates a new date object representing the 147 \* specified day number (which should be in the range 2 to 2958465. 148 \*

149 \* @param serial the serial number for the day (range: 2 to 2958465). 150 \*/

151 public SpreadsheetDate(final int serial) { 152

153 if ((serial >= SERIAL\_LOWER\_BOUND) && (serial <= SERIAL\_UPPER\_BOUND)) { 154 this.serial = serial;

155 }

156 else {

157 throw new IllegalArgumentException(

158 "SpreadsheetDate: Serial must be in range 2 to 2958465."); 159 }

160

161 // the day-month-year needs to be synchronised with the serial number... 162 calcDayMonthYear();

163

164 } 165

166 /\*\*

167 \* Returns the description that is attached to the date. It is not

168 \* required that a date have a description, but for some applications it 169 \* is useful.

170 \*

171 \* @return The description that is attached to the date. 172 \*/

173 public String getDescription() { 174 return this.description; 175 }

176

177 /\*\*

178 \* Sets the description for the date. 179 \*

180 \* @param description the description for this date (<code>null</code> 181 \* permitted).

182 \*/

183 public void setDescription(final String description) { 184 this.description = description;

185 } 186

<https://www.facebook.com/groups/phattrienlaptrinh>

**Appendix B: org.jfree.date.SerialDate** 385

**Listing B-5 (continued) SpreadsheetDate.java**

187 /\*\*

188 \* Returns the serial number for the date, where 1 January 1900 = 2 189 \* (this corresponds, almost, to the numbering system used in Microsoft 190 \* Excel for Windows and Lotus 1-2-3).

191 \*

192 \* @return The serial number of this date. 193 \*/

194 public int toSerial() { 195 return this.serial; 196 }

197

198 /\*\*

199 \* Returns a <code>java.util.Date</code> equivalent to this date. 200 \*

201 \* @return The date. 202 \*/

203 public Date toDate() {

204 final Calendar calendar = Calendar.getInstance();

205 calendar.set(getYYYY(), getMonth() - 1, getDayOfMonth(), 0, 0, 0); 206 return calendar.getTime();

207 } 208

209 /\*\*

210 \* Returns the year (assume a valid range of 1900 to 9999). 211 \*

212 \* @return The year. 213 \*/

214 public int getYYYY() { 215 return this.year; 216 }

217

218 /\*\*

219 \* Returns the month (January = 1, February = 2, March = 3). 220 \*

221 \* @return The month of the year. 222 \*/

223 public int getMonth() { 224 return this.month; 225 }

226

227 /\*\*

228 \* Returns the day of the month. 229 \*

230 \* @return The day of the month. 231 \*/

232 public int getDayOfMonth() { 233 return this.day;

234 } 235

236 /\*\*

237 \* Returns a code representing the day of the week. 238 \* <P>

239 \* The codes are defined in the {@link SerialDate} class as:

240 \* <code>SUNDAY</code>, <code>MONDAY</code>, <code>TUESDAY</code>,

241 \* <code>WEDNESDAY</code>, <code>THURSDAY</code>, <code>FRIDAY</code>, and 242 \* <code>SATURDAY</code>.

243 \*

244 \* @return A code representing the day of the week. 245 \*/

246 public int getDayOfWeek() {

247 return (this.serial + 6) % 7 + 1; 248 }

<https://www.facebook.com/groups/phattrienlaptrinh>

386 **Appendix B: org.jfree.date.SerialDate**

**Listing B-5 (continued) SpreadsheetDate.java**

249

250 /\*\*

251 \* Tests the equality of this date with an arbitrary object. 252 \* <P>

253 \* This method will return true ONLY if the object is an instance of the 254 \* {@link SerialDate} base class, and it represents the same day as this 255 \* {@link SpreadsheetDate}.

256 \*

257 \* @param object the object to compare (<code>null</code> permitted). 258 \*

259 \* @return A boolean. 260 \*/

261 public boolean equals(final Object object) { 262

263 if (object instanceof SerialDate) {

264 final SerialDate s = (SerialDate) object; 265 return (s.toSerial() == this.toSerial()); 266 }

267 else {

268 return false; 269 }

270

271 } 272

273 /\*\*

274 \* Returns a hash code for this object instance. 275 \*

276 \* @return A hash code. 277 \*/

278 public int hashCode() { 279 return toSerial(); 280 }

281

282 /\*\*

283 \* Returns the difference (in days) between this date and the specified 284 \* 'other' date.

285 \*

286 \* @param other the date being compared to. 287 \*

288 \* @return The difference (in days) between this date and the specified 289 \* 'other' date.

290 \*/

291 public int compare(final SerialDate other) { 292 return this.serial - other.toSerial(); 293 }

294

295 /\*\*

296 \* Implements the method required by the Comparable interface. 297 \*

298 \* @param other the other object (usually another SerialDate). 299 \*

300 \* @return A negative integer, zero, or a positive integer as this object 301 \* is less than, equal to, or greater than the specified object. 302 \*/

303 public int compareTo(final Object other) { 304 return compare((SerialDate) other); 305 }

306

307 /\*\*

308 \* Returns true if this SerialDate represents the same date as the 309 \* specified SerialDate.

310 \*

<https://www.facebook.com/groups/phattrienlaptrinh>

**Appendix B: org.jfree.date.SerialDate** 387

**Listing B-5 (continued) SpreadsheetDate.java**

311 \* @param other the date being compared to. 312 \*

313 \* @return <code>true</code> if this SerialDate represents the same date as 314 \* the specified SerialDate.

315 \*/

316 public boolean isOn(final SerialDate other) { 317 return (this.serial == other.toSerial()); 318 }

319

320 /\*\*

321 \* Returns true if this SerialDate represents an earlier date compared to 322 \* the specified SerialDate.

323 \*

324 \* @param other the date being compared to. 325 \*

326 \* @return <code>true</code> if this SerialDate represents an earlier date 327 \* compared to the specified SerialDate.

328 \*/

329 public boolean isBefore(final SerialDate other) { 330 return (this.serial < other.toSerial());

331 } 332

333 /\*\*

334 \* Returns true if this SerialDate represents the same date as the 335 \* specified SerialDate.

336 \*

337 \* @param other the date being compared to. 338 \*

339 \* @return <code>true</code> if this SerialDate represents the same date 340 \* as the specified SerialDate.

341 \*/

342 public boolean isOnOrBefore(final SerialDate other) { 343 return (this.serial <= other.toSerial());

344 } 345

346 /\*\*

347 \* Returns true if this SerialDate represents the same date as the 348 \* specified SerialDate.

349 \*

350 \* @param other the date being compared to. 351 \*

352 \* @return <code>true</code> if this SerialDate represents the same date 353 \* as the specified SerialDate.

354 \*/

355 public boolean isAfter(final SerialDate other) { 356 return (this.serial > other.toSerial()); 357 }

358

359 /\*\*

360 \* Returns true if this SerialDate represents the same date as the 361 \* specified SerialDate.

362 \*

363 \* @param other the date being compared to. 364 \*

365 \* @return <code>true</code> if this SerialDate represents the same date as 366 \* the specified SerialDate.

367 \*/

368 public boolean isOnOrAfter(final SerialDate other) { 369 return (this.serial >= other.toSerial());

370 } 371

372 /\*\*

373 \* Returns <code>true</code> if this {@link SerialDate} is within the

<https://www.facebook.com/groups/phattrienlaptrinh>

388 **Appendix B: org.jfree.date.SerialDate**

**Listing B-5 (continued) SpreadsheetDate.java**

374 \* specified range (INCLUSIVE). The date order of d1 and d2 is not 375 \* important.

376 \*

377 \* @param d1 a boundary date for the range.

378 \* @param d2 the other boundary date for the range. 379 \*

380 \* @return A boolean. 381 \*/

382 public boolean isInRange(final SerialDate d1, final SerialDate d2) { 383 return isInRange(d1, d2, SerialDate.INCLUDE\_BOTH);

384 } 385

386 /\*\*

387 \* Returns true if this SerialDate is within the specified range (caller 388 \* specifies whether or not the end-points are included). The order of d1 389 \* and d2 is not important.

390 \*

391 \* @param d1 one boundary date for the range.

392 \* @param d2 a second boundary date for the range.

393 \* @param include a code that controls whether or not the start and end 394 \* dates are included in the range.

395 \*

396 \* @return <code>true</code> if this SerialDate is within the specified 397 \* range.

398 \*/

399 public boolean isInRange(final SerialDate d1, final SerialDate d2, 400 final int include) {

401 final int s1 = d1.toSerial(); 402 final int s2 = d2.toSerial();

403 final int start = Math.min(s1, s2); 404 final int end = Math.max(s1, s2); 405

406 final int s = toSerial();

407 if (include == SerialDate.INCLUDE\_BOTH) { 408 return (s >= start && s <= end);

409 }

410 else if (include == SerialDate.INCLUDE\_FIRST) { 411 return (s >= start && s < end);

412 }

413 else if (include == SerialDate.INCLUDE\_SECOND) { 414 return (s > start && s <= end);

415 }

416 else {

417 return (s > start && s < end); 418 }

419 } 420

421 /\*\*

422 \* Calculate the serial number from the day, month and year. 423 \* <P>

424 \* 1-Jan-1900 = 2. 425 \*

426 \* @param d the day. 427 \* @param m the month. 428 \* @param y the year. 429 \*

430 \* @return the serial number from the day, month and year. 431 \*/

432 private int calcSerial(final int d, final int m, final int y) {

433 final int yy = ((y - 1900) \* 365) + SerialDate.leapYearCount(y - 1); 434 int mm = SerialDate.AGGREGATE\_DAYS\_TO\_END\_OF\_PRECEDING\_MONTH[m]; 435 if (m > MonthConstants.FEBRUARY) {

<https://www.facebook.com/groups/phattrienlaptrinh>

**Appendix B: org.jfree.date.SerialDate** 389

**Listing B-5 (continued) SpreadsheetDate.java**

436 if (SerialDate.isLeapYear(y)) { 437 mm = mm + 1;

438 } 439 }

440 final int dd = d;

441 return yy + mm + dd + 1; 442 }

443

444 /\*\*

445 \* Calculate the day, month and year from the serial number. 446 \*/

447 private void calcDayMonthYear() { 448

449 // get the year from the serial date

450 final int days = this.serial - SERIAL\_LOWER\_BOUND; 451 // overestimated because we ignored leap days

452 final int overestimatedYYYY = 1900 + (days / 365);

453 final int leaps = SerialDate.leapYearCount(overestimatedYYYY); 454 final int nonleapdays = days - leaps;

455 // underestimated because we overestimated years 456 int underestimatedYYYY = 1900 + (nonleapdays / 365); 457

458 if (underestimatedYYYY == overestimatedYYYY) { 459 this.year = underestimatedYYYY;

460 }

461 else {

462 int ss1 = calcSerial(1, 1, underestimatedYYYY); 463 while (ss1 <= this.serial) {

464 underestimatedYYYY = underestimatedYYYY + 1; 465 ss1 = calcSerial(1, 1, underestimatedYYYY); 466 }

467 this.year = underestimatedYYYY - 1; 468 }

469

470 final int ss2 = calcSerial(1, 1, this.year); 471

472 int[] daysToEndOfPrecedingMonth

473 = AGGREGATE\_DAYS\_TO\_END\_OF\_PRECEDING\_MONTH; 474

475 if (isLeapYear(this.year)) { 476 daysToEndOfPrecedingMonth

477 = LEAP\_YEAR\_AGGREGATE\_DAYS\_TO\_END\_OF\_PRECEDING\_MONTH; 478 }

479

480 // get the month from the serial date 481 int mm = 1;

482 int sss = ss2 + daysToEndOfPrecedingMonth[mm] - 1; 483 while (sss < this.serial) {

484 mm = mm + 1;

485 sss = ss2 + daysToEndOfPrecedingMonth[mm] - 1; 486 }

487 this.month = mm - 1; 488

489 // what's left is d(+1); 490 this.day = this.serial - ss2

491 - daysToEndOfPrecedingMonth[this.month] + 1; 492

493 } 494

495 }

<https://www.facebook.com/groups/phattrienlaptrinh>

390 **Appendix B: org.jfree.date.SerialDate**

**Listing B-6 RelativeDayOfWeekRule.java**

1 /\* ======================================================================== 2 \* JCommon : a free general purpose class library for the Java(tm) platform 3 \* ======================================================================== 4 \*

5 \* (C) Copyright 2000-2005, by Object Refinery Limited and Contributors. 6 \*

7 \* Project Info: http://www.jfree.org/jcommon/index.html 8 \*

9 \* This library is free software; you can redistribute it and/or modify it 10 \* under the terms of the GNU Lesser General Public License as published by 11 \* the Free Software Foundation; either version 2.1 of the License, or

12 \* (at your option) any later version. 13 \*

14 \* This library is distributed in the hope that it will be useful, but

15 \* WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY 16 \* or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public

17 \* License for more details. 18 \*

19 \* You should have received a copy of the GNU Lesser General Public 20 \* License along with this library; if not, write to the Free Software

21 \* Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, 22 \* USA.

23 \*

24 \* [Java is a trademark or registered trademark of Sun Microsystems, Inc. 25 \* in the United States and other countries.]

26 \*

27 \* --------------------------28 \* RelativeDayOfWeekRule.java 29 \* --------------------------

30 \* (C) Copyright 2000-2003, by Object Refinery Limited and Contributors. 31 \*

32 \* Original Author: David Gilbert (for Object Refinery Limited); 33 \* Contributor(s): -;

34 \*

35 \* $Id: RelativeDayOfWeekRule.java,v 1.6 2005/11/16 15:58:40 taqua Exp $ 36 \*

37 \* Changes (from 26-Oct-2001) 38 \* --------------------------

39 \* 26-Oct-2001 : Changed package to com.jrefinery.date.\*; 40 \* 03-Oct-2002 : Fixed errors reported by Checkstyle (DG); 41 \*

42 \*/ 43

44 package org.jfree.date; 45

46 /\*\*

47 \* An annual date rule that returns a date for each year based on (a) a 48 \* reference rule; (b) a day of the week; and (c) a selection parameter 49 \* (SerialDate.PRECEDING, SerialDate.NEAREST, SerialDate.FOLLOWING). 50 \* <P>

51 \* For example, Good Friday can be specified as 'the Friday PRECEDING Easter 52 \* Sunday'.

53 \*

54 \* @author David Gilbert 55 \*/

56 public class RelativeDayOfWeekRule extends AnnualDateRule { 57

58 /\*\* A reference to the annual date rule on which this rule is based. \*/ 59 private AnnualDateRule subrule;

60

61 /\*\*

62 \* The day of the week (SerialDate.MONDAY, SerialDate.TUESDAY, and so on).

<https://www.facebook.com/groups/phattrienlaptrinh>

**Appendix B: org.jfree.date.SerialDate** 391

**Listing B-6 (continued) RelativeDayOfWeekRule.java**

63 \*/

64 private int dayOfWeek; 65

66 /\*\* Specifies which day of the week (PRECEDING, NEAREST or FOLLOWING). \*/ 67 private int relative;

68

69 /\*\*

70 \* Default constructor - builds a rule for the Monday following 1 January. 71 \*/

72 public RelativeDayOfWeekRule() {

73 this(new DayAndMonthRule(), SerialDate.MONDAY, SerialDate.FOLLOWING); 74 }

75

76 /\*\*

77 \* Standard constructor - builds rule based on the supplied sub-rule. 78 \*

79 \* @param subrule the rule that determines the reference date.

80 \* @param dayOfWeek the day-of-the-week relative to the reference date. 81 \* @param relative indicates \*which\* day-of-the-week (preceding, nearest 82 \* or following).

83 \*/

84 public RelativeDayOfWeekRule(final AnnualDateRule subrule, 85 final int dayOfWeek, final int relative) {

86 this.subrule = subrule;

87 this.dayOfWeek = dayOfWeek; 88 this.relative = relative; 89 }

90

91 /\*\*

92 \* Returns the sub-rule (also called the reference rule). 93 \*

94 \* @return The annual date rule that determines the reference date for this 95 \* rule.

96 \*/

97 public AnnualDateRule getSubrule() { 98 return this.subrule;

99 } 100

101 /\*\*

102 \* Sets the sub-rule. 103 \*

104 \* @param subrule the annual date rule that determines the reference date 105 \* for this rule.

106 \*/

107 public void setSubrule(final AnnualDateRule subrule) { 108 this.subrule = subrule;

109 } 110

111 /\*\*

112 \* Returns the day-of-the-week for this rule. 113 \*

114 \* @return the day-of-the-week for this rule. 115 \*/

116 public int getDayOfWeek() { 117 return this.dayOfWeek; 118 }

119

120 /\*\*

121 \* Sets the day-of-the-week for this rule. 122 \*

123 \* @param dayOfWeek the day-of-the-week (SerialDate.MONDAY, 124 \* SerialDate.TUESDAY, and so on).

<https://www.facebook.com/groups/phattrienlaptrinh>

392 **Appendix B: org.jfree.date.SerialDate**

**Listing B-6 (continued) RelativeDayOfWeekRule.java**

125 \*/

126 public void setDayOfWeek(final int dayOfWeek) { 127 this.dayOfWeek = dayOfWeek;

128 } 129

130 /\*\*

131 \* Returns the 'relative' attribute, that determines \*which\* 132 \* day-of-the-week we are interested in (SerialDate.PRECEDING, 133 \* SerialDate.NEAREST or SerialDate.FOLLOWING).

134 \*

135 \* @return The 'relative' attribute. 136 \*/

137 public int getRelative() { 138 return this.relative; 139 }

140

141 /\*\*

142 \* Sets the 'relative' attribute (SerialDate.PRECEDING, SerialDate.NEAREST, 143 \* SerialDate.FOLLOWING).

144 \*

145 \* @param relative determines \*which\* day-of-the-week is selected by this 146 \* rule.

147 \*/

148 public void setRelative(final int relative) { 149 this.relative = relative;

150 } 151

152 /\*\*

153 \* Creates a clone of this rule. 154 \*

155 \* @return a clone of this rule. 156 \*

157 \* @throws CloneNotSupportedException this should never happen. 158 \*/

159 public Object clone() throws CloneNotSupportedException { 160 final RelativeDayOfWeekRule duplicate

161 = (RelativeDayOfWeekRule) super.clone();

162 duplicate.subrule = (AnnualDateRule) duplicate.getSubrule().clone(); 163 return duplicate;

164 } 165

166 /\*\*

167 \* Returns the date generated by this rule, for the specified year. 168 \*

169 \* @param year the year (1900 &lt;= year &lt;= 9999). 170 \*

171 \* @return The date generated by the rule for the given year (possibly 172 \* <code>null</code>).

173 \*/

174 public SerialDate getDate(final int year) { 175

176 // check argument...

177 if ((year < SerialDate.MINIMUM\_YEAR\_SUPPORTED)

178 || (year > SerialDate.MAXIMUM\_YEAR\_SUPPORTED)) { 179 throw new IllegalArgumentException(

180 "RelativeDayOfWeekRule.getDate(): year outside valid range."); 181 }

182

183 // calculate the date... 184 SerialDate result = null;

185 final SerialDate base = this.subrule.getDate(year); 186

<https://www.facebook.com/groups/phattrienlaptrinh>

**Appendix B: org.jfree.date.SerialDate** 393

**Listing B-6 (continued) RelativeDayOfWeekRule.java**

187 if (base != null) {

188 switch (this.relative) {

189 case(SerialDate.PRECEDING):

190 result = SerialDate.getPreviousDayOfWeek(this.dayOfWeek, 191 base);

192 break;

193 case(SerialDate.NEAREST):

194 result = SerialDate.getNearestDayOfWeek(this.dayOfWeek, 195 base);

196 break;

197 case(SerialDate.FOLLOWING):

198 result = SerialDate.getFollowingDayOfWeek(this.dayOfWeek, 199 base);

200 break; 201 default: 202 break; 203 }

204 }

205 return result; 206

207 } 208

209 }

<https://www.facebook.com/groups/phattrienlaptrinh>

394 **Appendix B: org.jfree.date.SerialDate**

**Listing B-7 DayDate.java (Final)**

1 /\* ======================================================================== 2 \* JCommon : a free general purpose class library for the Java(tm) platform 3 \* ======================================================================== 4 \*

5 \* (C) Copyright 2000-2005, by Object Refinery Limited and Contributors. ...

36 \*/

37 package org.jfree.date; 38

39 import java.io.Serializable; 40 import java.util.\*;

41

42 /\*\*

43 \* An abstract class that represents immutable dates with a precision of 44 \* one day. The implementation will map each date to an integer that 45 \* represents an ordinal number of days from some fixed origin.

46 \*

47 \* Why not just use java.util.Date? We will, when it makes sense. At times, 48 \* java.util.Date can be \*too\* precise - it represents an instant in time, 49 \* accurate to 1/1000th of a second (with the date itself depending on the 50 \* time-zone). Sometimes we just want to represent a particular day (e.g. 21 51 \* January 2015) without concerning ourselves about the time of day, or the 52 \* time-zone, or anything else. That's what we've defined DayDate for.

53 \*

54 \* Use DayDateFactory.makeDate to create an instance. 55 \*

56 \* @author David Gilbert

57 \* @author Robert C. Martin did a lot of refactoring. 58 \*/

59

60 public abstract class DayDate implements Comparable, Serializable { 61 public abstract int getOrdinalDay();

62 public abstract int getYear(); 63 public abstract Month getMonth();

64 public abstract int getDayOfMonth(); 65

66 protected abstract Day getDayOfWeekForOrdinalZero(); 67

68 public DayDate plusDays(int days) {

69 return DayDateFactory.makeDate(getOrdinalDay() + days); 70 }

71

72 public DayDate plusMonths(int months) {

73 int thisMonthAsOrdinal = getMonth().toInt() - Month.JANUARY.toInt(); 74 int thisMonthAndYearAsOrdinal = 12 \* getYear() + thisMonthAsOrdinal; 75 int resultMonthAndYearAsOrdinal = thisMonthAndYearAsOrdinal + months; 76 int resultYear = resultMonthAndYearAsOrdinal / 12;

77 int resultMonthAsOrdinal = resultMonthAndYearAsOrdinal % 12 + Month.JANUARY.toInt(); 78 Month resultMonth = Month.fromInt(resultMonthAsOrdinal);

79 int resultDay = correctLastDayOfMonth(getDayOfMonth(), resultMonth, resultYear); 80 return DayDateFactory.makeDate(resultDay, resultMonth, resultYear);

81 } 82

83 public DayDate plusYears(int years) { 84 int resultYear = getYear() + years;

85 int resultDay = correctLastDayOfMonth(getDayOfMonth(), getMonth(), resultYear); 86 return DayDateFactory.makeDate(resultDay, getMonth(), resultYear);

87 } 88

89 private int correctLastDayOfMonth(int day, Month month, int year) { 90 int lastDayOfMonth = DateUtil.lastDayOfMonth(month, year);

91 if (day > lastDayOfMonth)

<https://www.facebook.com/groups/phattrienlaptrinh>

**Appendix B: org.jfree.date.SerialDate** 395

**Listing B-7 (continued) DayDate.java (Final)**

92 day = lastDayOfMonth; 93 return day;

94 } 95

96 public DayDate getPreviousDayOfWeek(Day targetDayOfWeek) {

97 int offsetToTarget = targetDayOfWeek.toInt() - getDayOfWeek().toInt(); 98 if (offsetToTarget >= 0)

99 offsetToTarget -= 7;

100 return plusDays(offsetToTarget); 101 }

102

103 public DayDate getFollowingDayOfWeek(Day targetDayOfWeek) {

104 int offsetToTarget = targetDayOfWeek.toInt() - getDayOfWeek().toInt(); 105 if (offsetToTarget <= 0)

106 offsetToTarget += 7;

107 return plusDays(offsetToTarget); 108 }

109

110 public DayDate getNearestDayOfWeek(Day targetDayOfWeek) {

111 int offsetToThisWeeksTarget = targetDayOfWeek.toInt() - getDayOfWeek().toInt(); 112 int offsetToFutureTarget = (offsetToThisWeeksTarget + 7) % 7;

113 int offsetToPreviousTarget = offsetToFutureTarget - 7; 114

115 if (offsetToFutureTarget > 3)

116 return plusDays(offsetToPreviousTarget); 117 else

118 return plusDays(offsetToFutureTarget); 119 }

120

121 public DayDate getEndOfMonth() { 122 Month month = getMonth();

123 int year = getYear();

124 int lastDay = DateUtil.lastDayOfMonth(month, year); 125 return DayDateFactory.makeDate(lastDay, month, year); 126 }

127

128 public Date toDate() {

129 final Calendar calendar = Calendar.getInstance();

130 int ordinalMonth = getMonth().toInt() - Month.JANUARY.toInt(); 131 calendar.set(getYear(), ordinalMonth, getDayOfMonth(), 0, 0, 0); 132 return calendar.getTime();

133 } 134

135 public String toString() {

136 return String.format("%02d-%s-%d", getDayOfMonth(), getMonth(), getYear()); 137 }

138

139 public Day getDayOfWeek() {

140 Day startingDay = getDayOfWeekForOrdinalZero();

141 int startingOffset = startingDay.toInt() - Day.SUNDAY.toInt(); 142 int ordinalOfDayOfWeek = (getOrdinalDay() + startingOffset) % 7; 143 return Day.fromInt(ordinalOfDayOfWeek + Day.SUNDAY.toInt()); 144 }

145

146 public int daysSince(DayDate date) {

147 return getOrdinalDay() - date.getOrdinalDay(); 148 }

149

150 public boolean isOn(DayDate other) {

151 return getOrdinalDay() == other.getOrdinalDay(); 152 }

153

<https://www.facebook.com/groups/phattrienlaptrinh>

396 **Appendix B: org.jfree.date.SerialDate**

**Listing B-7 (continued) DayDate.java (Final)**

154 public boolean isBefore(DayDate other) {

155 return getOrdinalDay() < other.getOrdinalDay(); 156 }

157

158 public boolean isOnOrBefore(DayDate other) {

159 return getOrdinalDay() <= other.getOrdinalDay(); 160 }

161

162 public boolean isAfter(DayDate other) {

163 return getOrdinalDay() > other.getOrdinalDay(); 164 }

165

166 public boolean isOnOrAfter(DayDate other) {

167 return getOrdinalDay() >= other.getOrdinalDay(); 168 }

169

170 public boolean isInRange(DayDate d1, DayDate d2) { 171 return isInRange(d1, d2, DateInterval.CLOSED); 172 }

173

174 public boolean isInRange(DayDate d1, DayDate d2, DateInterval interval) { 175 int left = Math.min(d1.getOrdinalDay(), d2.getOrdinalDay());

176 int right = Math.max(d1.getOrdinalDay(), d2.getOrdinalDay()); 177 return interval.isIn(getOrdinalDay(), left, right);

178 } 179 }

<https://www.facebook.com/groups/phattrienlaptrinh>

**Appendix B: org.jfree.date.SerialDate** 397

**Listing B-8 Month.java (Final)**

1 package org.jfree.date; 2

3 import java.text.DateFormatSymbols; 4

5 public enum Month {

6 JANUARY(1), FEBRUARY(2), MARCH(3), 7 APRIL(4), MAY(5), JUNE(6),

8 JULY(7), AUGUST(8), SEPTEMBER(9), 9 OCTOBER(10),NOVEMBER(11),DECEMBER(12);

10 private static DateFormatSymbols dateFormatSymbols = new DateFormatSymbols(); 11 private static final int[] LAST\_DAY\_OF\_MONTH =

12 {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}; 13

14 private int index; 15

16 Month(int index) { 17 this.index = index; 18 }

19

20 public static Month fromInt(int monthIndex) { 21 for (Month m : Month.values()) {

22 if (m.index == monthIndex) 23 return m;

24 }

25 throw new IllegalArgumentException("Invalid month index " + monthIndex); 26 }

27

28 public int lastDay() {

29 return LAST\_DAY\_OF\_MONTH[index]; 30 }

31

32 public int quarter() {

33 return 1 + (index - 1) / 3; 34 }

35

36 public String toString() {

37 return dateFormatSymbols.getMonths()[index - 1]; 38 }

39

40 public String toShortString() {

41 return dateFormatSymbols.getShortMonths()[index - 1]; 42 }

43

44 public static Month parse(String s) { 45 s = s.trim();

46 for (Month m : Month.values()) 47 if (m.matches(s))

48 return m; 49

50 try {

51 return fromInt(Integer.parseInt(s)); 52 }

53 catch (NumberFormatException e) {}

54 throw new IllegalArgumentException("Invalid month " + s); 55 }

56

57 private boolean matches(String s) {

58 return s.equalsIgnoreCase(toString()) || 59 s.equalsIgnoreCase(toShortString()); 60 }

61

62 public int toInt() { 63 return index;

64 } 65 }

<https://www.facebook.com/groups/phattrienlaptrinh>

398 **Appendix B: org.jfree.date.SerialDate**

**Listing B-9 Day.java (Final)**

1 package org.jfree.date; 2

3 import java.util.Calendar;

4 import java.text.DateFormatSymbols; 5

6 public enum Day {

7 MONDAY(Calendar.MONDAY), 8 TUESDAY(Calendar.TUESDAY),

9 WEDNESDAY(Calendar.WEDNESDAY), 10 THURSDAY(Calendar.THURSDAY), 11 FRIDAY(Calendar.FRIDAY),

12 SATURDAY(Calendar.SATURDAY), 13 SUNDAY(Calendar.SUNDAY);

14

15 private final int index;

16 private static DateFormatSymbols dateSymbols = new DateFormatSymbols(); 17

18 Day(int day) { 19 index = day; 20 }

21

22 public static Day fromInt(int index) throws IllegalArgumentException { 23 for (Day d : Day.values())

24 if (d.index == index) 25 return d;

26 throw new IllegalArgumentException(

27 String.format("Illegal day index: %d.", index)); 28 }

29

30 public static Day parse(String s) throws IllegalArgumentException { 31 String[] shortWeekdayNames =

32 dateSymbols.getShortWeekdays(); 33 String[] weekDayNames =

34 dateSymbols.getWeekdays(); 35

36 s = s.trim();

37 for (Day day : Day.values()) {

38 if (s.equalsIgnoreCase(shortWeekdayNames[day.index]) || 39 s.equalsIgnoreCase(weekDayNames[day.index])) {

40 return day; 41 }

42 }

43 throw new IllegalArgumentException(

44 String.format("%s is not a valid weekday string", s)); 45 }

46

47 public String toString() {

48 return dateSymbols.getWeekdays()[index]; 49 }

50

51 public int toInt() { 52 return index;

53 } 54 }

<https://www.facebook.com/groups/phattrienlaptrinh>

**Appendix B: org.jfree.date.SerialDate** 399

**Listing B-10 DateInterval.java (Final)**

1 package org.jfree.date; 2

3 public enum DateInterval { 4 OPEN {

5 public boolean isIn(int d, int left, int right) { 6 return d > left && d < right;

7 } 8 },

9 CLOSED\_LEFT {

10 public boolean isIn(int d, int left, int right) { 11 return d >= left && d < right;

12 } 13 },

14 CLOSED\_RIGHT {

15 public boolean isIn(int d, int left, int right) { 16 return d > left && d <= right;

17 } 18 },

19 CLOSED {

20 public boolean isIn(int d, int left, int right) { 21 return d >= left && d <= right;

22 } 23 }; 24

25 public abstract boolean isIn(int d, int left, int right); 26 }

<https://www.facebook.com/groups/phattrienlaptrinh>

400 **Appendix B: org.jfree.date.SerialDate**

**Listing B-11 WeekInMonth.java (Final)**

1 package org.jfree.date; 2

3 public enum WeekInMonth {

4 FIRST(1), SECOND(2), THIRD(3), FOURTH(4), LAST(0); 5 private final int index;

6

7 WeekInMonth(int index) { 8 this.index = index;

9 } 10

11 public int toInt() { 12 return index;

13 } 14 }

<https://www.facebook.com/groups/phattrienlaptrinh>

**Appendix B: org.jfree.date.SerialDate** 401

**Listing B-12 WeekdayRange.java (Final)**

1 package org.jfree.date; 2

3 public enum WeekdayRange { 4 LAST, NEAREST, NEXT

5 }

<https://www.facebook.com/groups/phattrienlaptrinh>

402 **Appendix B: org.jfree.date.SerialDate**

**Listing B-13 DateUtil.java (Final)**

1 package org.jfree.date; 2

3 import java.text.DateFormatSymbols; 4

5 public class DateUtil {

6 private static DateFormatSymbols dateFormatSymbols = new DateFormatSymbols(); 7

8 public static String[] getMonthNames() { 9 return dateFormatSymbols.getMonths();

10 } 11

12 public static boolean isLeapYear(int year) { 13 boolean fourth = year % 4 == 0;

14 boolean hundredth = year % 100 == 0;

15 boolean fourHundredth = year % 400 == 0;

16 return fourth && (!hundredth || fourHundredth); 17 }

18

19 public static int lastDayOfMonth(Month month, int year) { 20 if (month == Month.FEBRUARY && isLeapYear(year))

21 return month.lastDay() + 1; 22 else

23 return month.lastDay(); 24 }

25

26 public static int leapYearCount(int year) { 27 int leap4 = (year - 1896) / 4;

28 int leap100 = (year - 1800) / 100; 29 int leap400 = (year - 1600) / 400; 30 return leap4 - leap100 + leap400; 31 }

32 }

<https://www.facebook.com/groups/phattrienlaptrinh>

**Appendix B: org.jfree.date.SerialDate** 403

**Listing B-14 DayDateFactory.java (Final)**

1 package org.jfree.date; 2

3 public abstract class DayDateFactory {

4 private static DayDateFactory factory = new SpreadsheetDateFactory(); 5 public static void setInstance(DayDateFactory factory) {

6 DayDateFactory.factory = factory; 7 }

8

9 protected abstract DayDate \_makeDate(int ordinal);

10 protected abstract DayDate \_makeDate(int day, Month month, int year); 11 protected abstract DayDate \_makeDate(int day, int month, int year); 12 protected abstract DayDate \_makeDate(java.util.Date date);

13 protected abstract int \_getMinimumYear(); 14 protected abstract int \_getMaximumYear(); 15

16 public static DayDate makeDate(int ordinal) { 17 return factory.\_makeDate(ordinal);

18 } 19

20 public static DayDate makeDate(int day, Month month, int year) { 21 return factory.\_makeDate(day, month, year);

22 } 23

24 public static DayDate makeDate(int day, int month, int year) { 25 return factory.\_makeDate(day, month, year);

26 } 27

28 public static DayDate makeDate(java.util.Date date) { 29 return factory.\_makeDate(date);

30 } 31

32 public static int getMinimumYear() { 33 return factory.\_getMinimumYear(); 34 }

35

36 public static int getMaximumYear() { 37 return factory.\_getMaximumYear(); 38 }

39 }

<https://www.facebook.com/groups/phattrienlaptrinh>

404 **Appendix B: org.jfree.date.SerialDate**

**Listing B-15 SpreadsheetDateFactory.java (Final)**

1 package org.jfree.date; 2

3 import java.util.\*; 4

5 public class SpreadsheetDateFactory extends DayDateFactory { 6 public DayDate \_makeDate(int ordinal) {

7 return new SpreadsheetDate(ordinal); 8 }

9

10 public DayDate \_makeDate(int day, Month month, int year) { 11 return new SpreadsheetDate(day, month, year);

12 } 13

14 public DayDate \_makeDate(int day, int month, int year) { 15 return new SpreadsheetDate(day, month, year);

16 } 17

18 public DayDate \_makeDate(Date date) {

19 final GregorianCalendar calendar = new GregorianCalendar(); 20 calendar.setTime(date);

21 return new SpreadsheetDate( 22 calendar.get(Calendar.DATE),

23 Month.fromInt(calendar.get(Calendar.MONTH) + 1), 24 calendar.get(Calendar.YEAR));

25 } 26

27 protected int \_getMinimumYear() {

28 return SpreadsheetDate.MINIMUM\_YEAR\_SUPPORTED; 29 }

30

31 protected int \_getMaximumYear() {

32 return SpreadsheetDate.MAXIMUM\_YEAR\_SUPPORTED; 33 }

34 }

<https://www.facebook.com/groups/phattrienlaptrinh>

**Appendix B: org.jfree.date.SerialDate** 405

**Listing B-16 SpreadsheetDate.java (Final)**

1 /\* ======================================================================== 2 \* JCommon : a free general purpose class library for the Java(tm) platform 3 \* ======================================================================== 4 \*

5 \* (C) Copyright 2000-2005, by Object Refinery Limited and Contributors. 6 \*

...

52 \* 53 \*/ 54

55 package org.jfree.date; 56

57 import static org.jfree.date.Month.FEBRUARY; 58

59 import java.util.\*; 60

61 /\*\*

62 \* Represents a date using an integer, in a similar fashion to the

63 \* implementation in Microsoft Excel. The range of dates supported is 64 \* 1-Jan-1900 to 31-Dec-9999.

65 \* <p/>

66 \* Be aware that there is a deliberate bug in Excel that recognises the year 67 \* 1900 as a leap year when in fact it is not a leap year. You can find more 68 \* information on the Microsoft website in article Q181370:

69 \* <p/>

70 \* http://support.microsoft.com/support/kb/articles/Q181/3/70.asp 71 \* <p/>

72 \* Excel uses the convention that 1-Jan-1900 = 1. This class uses the 73 \* convention 1-Jan-1900 = 2.

74 \* The result is that the day number in this class will be different to the

75 \* Excel figure for January and February 1900...but then Excel adds in an extra 76 \* day (29-Feb-1900 which does not actually exist!) and from that point forward 77 \* the day numbers will match.

78 \*

79 \* @author David Gilbert 80 \*/

81 public class SpreadsheetDate extends DayDate {

82 public static final int EARLIEST\_DATE\_ORDINAL = 2; // 1/1/1900 83 public static final int LATEST\_DATE\_ORDINAL = 2958465; // 12/31/9999 84 public static final int MINIMUM\_YEAR\_SUPPORTED = 1900;

85 public static final int MAXIMUM\_YEAR\_SUPPORTED = 9999;

86 static final int[] AGGREGATE\_DAYS\_TO\_END\_OF\_PRECEDING\_MONTH =

87 {0, 0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334, 365};

88 static final int[] LEAP\_YEAR\_AGGREGATE\_DAYS\_TO\_END\_OF\_PRECEDING\_MONTH =

89 {0, 0, 31, 60, 91, 121, 152, 182, 213, 244, 274, 305, 335, 366}; 90

91 private int ordinalDay; 92 private int day;

93 private Month month; 94 private int year; 95

96 public SpreadsheetDate(int day, Month month, int year) {

97 if (year < MINIMUM\_YEAR\_SUPPORTED || year > MAXIMUM\_YEAR\_SUPPORTED) 98 throw new IllegalArgumentException(

99 "The 'year' argument must be in range " +

100 MINIMUM\_YEAR\_SUPPORTED + " to " + MAXIMUM\_YEAR\_SUPPORTED + "."); 101 if (day < 1 || day > DateUtil.lastDayOfMonth(month, year))

102 throw new IllegalArgumentException("Invalid 'day' argument."); 103

104 this.year = year; 105 this.month = month;

<https://www.facebook.com/groups/phattrienlaptrinh>

406 **Appendix B: org.jfree.date.SerialDate**

**Listing B-16 (continued) SpreadsheetDate.java (Final)**

106 this.day = day;

107 ordinalDay = calcOrdinal(day, month, year); 108 }

109

110 public SpreadsheetDate(int day, int month, int year) { 111 this(day, Month.fromInt(month), year);

112 } 113

114 public SpreadsheetDate(int serial) {

115 if (serial < EARLIEST\_DATE\_ORDINAL || serial > LATEST\_DATE\_ORDINAL) 116 throw new IllegalArgumentException(

117 "SpreadsheetDate: Serial must be in range 2 to 2958465."); 118

119 ordinalDay = serial; 120 calcDayMonthYear(); 121 }

122

123 public int getOrdinalDay() { 124 return ordinalDay;

125 } 126

127 public int getYear() { 128 return year;

129 } 130

131 public Month getMonth() { 132 return month;

133 } 134

135 public int getDayOfMonth() { 136 return day;

137 } 138

139 protected Day getDayOfWeekForOrdinalZero() {return Day.SATURDAY;} 140

141 public boolean equals(Object object) { 142 if (!(object instanceof DayDate)) 143 return false;

144

145 DayDate date = (DayDate) object;

146 return date.getOrdinalDay() == getOrdinalDay(); 147 }

148

149 public int hashCode() { 150 return getOrdinalDay(); 151 }

152

153 public int compareTo(Object other) { 154 return daysSince((DayDate) other); 155 }

156

157 private int calcOrdinal(int day, Month month, int year) { 158 int leapDaysForYear = DateUtil.leapYearCount(year - 1);

159 int daysUpToYear = (year - MINIMUM\_YEAR\_SUPPORTED) \* 365 + leapDaysForYear; 160 int daysUpToMonth = AGGREGATE\_DAYS\_TO\_END\_OF\_PRECEDING\_MONTH[month.toInt()]; 161 if (DateUtil.isLeapYear(year) && month.toInt() > FEBRUARY.toInt())

162 daysUpToMonth++;

163 int daysInMonth = day - 1;

164 return daysUpToYear + daysUpToMonth + daysInMonth + EARLIEST\_DATE\_ORDINAL; 165 }

166

<https://www.facebook.com/groups/phattrienlaptrinh>

**Appendix B: org.jfree.date.SerialDate** 407

**Listing B-16 (continued) SpreadsheetDate.java (Final)**

167 private void calcDayMonthYear() {

168 int days = ordinalDay - EARLIEST\_DATE\_ORDINAL;

169 int overestimatedYear = MINIMUM\_YEAR\_SUPPORTED + days / 365;

170 int nonleapdays = days - DateUtil.leapYearCount(overestimatedYear); 171 int underestimatedYear = MINIMUM\_YEAR\_SUPPORTED + nonleapdays / 365; 172

173 year = huntForYearContaining(ordinalDay, underestimatedYear); 174 int firstOrdinalOfYear = firstOrdinalOfYear(year);

175 month = huntForMonthContaining(ordinalDay, firstOrdinalOfYear);

176 day = ordinalDay - firstOrdinalOfYear - daysBeforeThisMonth(month.toInt()); 177 }

178

179 private Month huntForMonthContaining(int anOrdinal, int firstOrdinalOfYear) { 180 int daysIntoThisYear = anOrdinal - firstOrdinalOfYear;

181 int aMonth = 1;

182 while (daysBeforeThisMonth(aMonth) < daysIntoThisYear) 183 aMonth++;

184

185 return Month.fromInt(aMonth - 1); 186 }

187

188 private int daysBeforeThisMonth(int aMonth) { 189 if (DateUtil.isLeapYear(year))

190 return LEAP\_YEAR\_AGGREGATE\_DAYS\_TO\_END\_OF\_PRECEDING\_MONTH[aMonth] - 1; 191 else

192 return AGGREGATE\_DAYS\_TO\_END\_OF\_PRECEDING\_MONTH[aMonth] - 1; 193 }

194

195 private int huntForYearContaining(int anOrdinalDay, int startingYear) { 196 int aYear = startingYear;

197 while (firstOrdinalOfYear(aYear) <= anOrdinalDay) 198 aYear++;

199

200 return aYear - 1; 201 }

202

203 private int firstOrdinalOfYear(int year) { 204 return calcOrdinal(1, Month.JANUARY, year); 205 }

206

207 public static DayDate createInstance(Date date) {

208 GregorianCalendar calendar = new GregorianCalendar(); 209 calendar.setTime(date);

210 return new SpreadsheetDate(calendar.get(Calendar.DATE),

211 Month.fromInt(calendar.get(Calendar.MONTH) + 1), 212 calendar.get(Calendar.YEAR));

213 214 } 215 }

<https://www.facebook.com/groups/phattrienlaptrinh>

*This page intentionally left blank*

<https://www.facebook.com/groups/phattrienlaptrinh>

[**Appendix C**](#_page_761_0)

[**Cross References of Heuristics**](#_page_761_0)

Cross references of Smells and Heuristics. All other cross references can be deleted.

C1 .......................................................................16-276, 16-279, 17-292 C2 .........................................................16-279, 16-285, 16-295, 17-292 C3 .........................................................16-283, 16-285, 16-288, 17-293 C4 .................................................................................................17-293 C5 .................................................................................................17-293 E1 .................................................................................................17-294 E2 .................................................................................................17-294 F1 ....................................................................................14-239, 17-295 F2 .................................................................................................17-295 F3 .................................................................................................17-295 F4 ...............................14-289, 16-273, 16-285, 16-287, 16-288, 17-295 G1 ....................................................................................16-276, 17-295 G2 .......................................................................16-273, 16-274, 17-296 G3 ....................................................................................16-274, 17-296 G4 ................................................9-31, 16-279, 16-286, 16-291, 17-297 G5 ...................................9-31, 16-279, 16-286, 16-291, 16-296, 17-297 G6 ................................6-106, 16-280, 16-283, 16-284, 16-289, 16-293, 16-294, 16-296, 17-299

G7 .......................................................................16-281, 16-283, 17-300 G8 ....................................................................................16-283, 17-301 G9 ............................................16-283, 16-285, 16-286, 16-287, 17-302 G10 ..............................................5-86, 15-264, 16-276, 16-284, 17-302 G11 ..........................................15-264, 16-284, 16-288, 16-292, 17-302 G12 ...............16-284, 16-285, 16-286, 16-287, 16-288, 16-295, 17-303 G13 .....................................................................16-286, 16-288, 17-303 G14 .....................................................................16-288, 16-292, 17-304

409

<https://www.facebook.com/groups/phattrienlaptrinh>

410 **Appendix C: Cross References of Heuristics**

G15 ..................................................................................16-288, 17-305 G16 ..................................................................................16-289, 17-306 G17 .....................................................................16-289, 17-307, 17-312 G18 .......................................................16-289, 16-290, 16-291, 17-308 G19 .......................................................16-290, 16-291, 16-292, 17-309 G20 ..................................................................................16-290, 17-309 G21 ..................................................................................16-291, 17-310 G22 ..................................................................................16-294, 17-322 G23 ..........................................................??-44, 14-239, 16-295, 17-313 G24 ..................................................................................16-296, 17-313 G25 ..................................................................................16-296, 17-314 G26 ...............................................................................................17-316 G27 ...............................................................................................17-316 G28 ..................................................................................15-262, 17-317 G29 ..................................................................................15-262, 17-317 G30 ..................................................................................15-263, 17-317 G31 ..................................................................................15-264, 17-318 G32 ..................................................................................15-265, 17-319 G33 .....................................................................15-265, 15-266, 17-320 G34 ...........................................................................1-40, 6-106, 17-321 G35 ......................................................................................5-90, 17-323 G36 ....................................................................................6-103, 17-324 J1 .....................................................................................16-276, 17-325 J2 ........................................................................16-278, 16-285, 17-326 J3 ........................................................................16-283, 16-285, 17-327 N1 ..............................15-264, 16-277, 16-279, 16-282, 16-287, 16-288, 16-289, 16-290, 16-294, 16-296, 17-328

N2 ....................................................................................16-277, 17-330 N3 .......................................................................16-284, 16-288, 17-331 N4 .......................................................................15-263, 16-291, 17-332 N5 .............................................................2-26, 14-221, 15-262, 17-332 N6 ....................................................................................15-261, 17-333 N7 ....................................................................................15-263, 17-333 T1 .......................................................................16-273, 16-274, 17-334 T2 ....................................................................................16-273, 17-334 T3 ....................................................................................16-274, 17-334 T4 .................................................................................................17-334 T5 .......................................................................16-274, 16-275, 17-335 T6 ....................................................................................16-275, 17-335 T7 ....................................................................................16-275, 17-335 T8 ....................................................................................16-275, 17-335 T9 .................................................................................................17-336

<https://www.facebook.com/groups/phattrienlaptrinh>

[**Epilogue**](#_page_761_0)

In 2005, while attending the Agile conference in Denver, Elisabeth Hedrickson1 handed me a green wrist band similar to the kind that Lance Armstrong made so popular. This one said “Test Obsessed” on it. I gladly put it on and wore it proudly. Since learning TDD from Kent Beck in 1999, I have indeed become obsessed with test-driven development.

But then something strange happened. I found I could not take the band off. Not because it was physically stuck, but because it was *morally* stuck. The band made an overt statement about my professional ethics. It was a visible indication of my committment to writing the best code I could write. Taking it off seemed like a betrayal of those ethics and of that committment.

So it is on my wrist still. When I write code, I see it there in my peripheral vision. It is a constant reminder of the promise I made to myself to write clean code.

1. <http://www.qualitytree.com/>

411

<https://www.facebook.com/groups/phattrienlaptrinh>

*This page intentionally left blank*

<https://www.facebook.com/groups/phattrienlaptrinh>

[**Index**](#_page_761_0)

## detection, 237–238

++ (pre- or post-increment) operator, 325, 326

**A**

aborted computation, 109 abstract classes, 149, 271, 290

ABSTRACT FACTORY pattern, 38, 156, 273, 274

abstract interfaces, 94 abstract methods

adding to ArgumentMarshaler, 234–235 modifying, 282

abstract terms, 95 abstraction

classes depending on, 150

code at wrong level of, 290–291 descending one level at a time, 37 functions descending only one level

of, 304–306 mixing levels of, 36–37

names at the appropriate level of, 311 separating levels of, 305

wrapping an implementation, 11 abstraction levels

raising, 290 separating, 305

accessor functions, Law of Demeter and, 98

accessors, naming, 25 Active Records, 101 adapted server, 185

affinity, 84

*Agile Software Development: Principles, Patterns, Practices (PPP)*, 15

algorithms

correcting, 269–270 repeating, 48 understanding, 297–298

ambiguities

in code, 301 ignored tests as, 313

amplification comments, 59 analysis functions, 265 “annotation form”, of AspectJ, 166 Ant project, 76, 77

AOP (aspect-oriented programming), 160, 163

APIs. *See also* public APIs

calling a null-returning method from, 110

specialized for tests, 127 wrapping third-party, 108

applications

decoupled from Spring, 164 decoupling from construction

details, 156 infrastructure of, 163

keeping concurrency-related code separate, 181

arbitrary structure, 303–304

argsarray, converting into a list, 231–232 Args class

constructing, 194 implementation of, 194–200

rough drafts of, 201–212, 226–231

413

<https://www.facebook.com/groups/phattrienlaptrinh>

414

ArgsException class listing, 198–200

merging exceptions into, 239–242 argument(s)

flag, 41

for a function, 40 in functions, 288

monadic forms of, 41 reducing, 43

argument lists, 43 argument objects, 43 argument types

adding, 200, 237 negative impact of, 208

ArgumentMarshaler class

adding the skeleton of, 213–214 birth of, 212

ArgumentMarshaler interface, 197–198 arrays, moving, 279

art, of clean code, 6–7 artificial coupling, 293 AspectJ language, 166

aspect-oriented programming (AOP), 160, 163

aspects

in AOP, 160–161

“first-class” support for, 166 assert statements, 130–131 assertEquals, 42

assertions, using a set of, 111 assignments, unaligned, 87–88 atomic operation, 323–324 attributes, 68

authors

of JUnit, 252 programmers as, 13–14

authorship statements, 55

automated code instrumentation, 189–190 automated suite, of unit tests, 124

**B**

bad code, 3–4. *See also* dirty code; messy code

degrading effect of, 250

**Index**

example, 71–72

experience of cleaning, 250 not making up for, 55

bad comments, 59–74

banner, gathering functions beneath, 67 base classes, 290, 291

BDUF (Big Design Up Front), 167 beans, private variables manipulated,

100–101

Beck, Kent, 3, 34, 71, 171, 252, 289, 296

behaviors, 288–289

Big Design Up Front (BDUF), 167 blank lines, in code, 78–79

blocks, calling functions within, 35 Booch,Grady, 8–9

boolean, passing into a function, 41 boolean arguments, 194, 288 boolean map, deleting, 224

boolean output, of tests, 132 bound resources, 183, 184 boundaries

clean, 120

exploring and learning, 116 incorrect behavior at, 289 separating known from unknown,

118–119

boundary condition errors, 269 boundary conditions

encapsulating, 304 testing, 314

boundary tests, easing a migration, 118 “Bowling Game”, 312

Boy Scout Rule, 14–15, 257 following, 284 satisfying, 265

broken windows metaphor, 8 bucket brigade, 303

BUILD-OPERATE-CHECK pattern, 127 builds, 287

business logic, separating from error handling, 109

bylines, 68

byte-manipulation libraries, 161, 162–163

<https://www.facebook.com/groups/phattrienlaptrinh>

**Index**

**C**

*The C++ Programming Language*, 7 calculations, breaking into intermediate

values, 296 call stack, 324 Callable interface, 326 caller, cluttering, 104 calling hierarchy, 106

calls, avoiding chains of, 98 caring, for code, 10 Cartesian points, 42

CAS operation, as atomic, 328 change(s)

isolating from, 149–150

large number of very tiny, 213 organizing for, 147–150

tests enabling, 124 change history, deleting, 270 check exceptions, in Java, 106 circular wait, 337, 338–339 clarification, comments as, 57 clarity, 25, 26

class names, 25 classes

cohesion of, 140–141

creating for bigger concepts, 28–29 declaring instance variables, 81 enforcing design and business

rules, 115

exposing internals of, 294 instrumenting into ConTest, 342 keeping small, 136, 175 minimizing the number of, 176 naming, 25, 138

nonthread-safe, 328–329 as nouns of a language, 49 organization of, 136

organizing to reduce risk of change, 147

supporting advanced concurrency design, 183

classification, of errors, 107

415

clean boundaries, 120 clean code

art of, 6–7 described, 7–12 writing, 6–7

clean tests, 124–127 cleanliness

acquired sense of, 6–7 tied to tests, 9

cleanup, of code, 14–15 clever names, 26

client, using two methods, 330

client code, connecting to a server, 318 client-based locking, 185, 329, 330–332 clientScheduler, 320

client/server application, concurrency in, 317–321

Client/Server nonthreaded, code for, 343–346

client-server using threads, code changes, 346–347

ClientTest.java, 318, 344–346 closing braces, comments on, 67–68 Clover, 268, 269

clutter

Javadocs as, 276 keeping free of, 293

code, 2

bad, 3–4

Beck's rules of, 10 commented-out, 68–69, 287 dead, 292

explaining yourself in, 55 expressing yourself in, 54 formatting of, 76 implicity of, 18–19 instrumenting, 188, 342 jiggling, 190

making readable, 311 necessity of, 2

reading from top to bottom, 37 simplicity of, 18, 19

technique for shrouding, 20

<https://www.facebook.com/groups/phattrienlaptrinh>

416

code, *continued*

third-party, 114–115 width of lines in, 85–90

at wrong level of abstraction, 290–291 code bases, dominated by error

handling, 103

code changes, comments not always following, 54

code completion, automatic, 20 code coverage analysis, 254–256 code instrumentation, 188–190 “code sense”, 6, 7

code smells, listing of, 285–314 coding standard, 299

cohesion

of classes, 140–141 maintaining, 141–146

command line arguments, 193–194 commands, separating from queries, 45–46 comment header standard, 55–56 comment headers, replacing, 70 commented-out code, 68–69, 287 commenting style, example of bad, 71–72 comments

amplifying importance of something, 59

bad, 59–74 deleting, 282 as failures, 54 good, 55–59

heuristics on, 286–287 HTML, 69

inaccurate, 54 informative, 56 journal, 63–64 legal, 55–56 mandated, 63 misleading, 63 mumbling, 59–60

as a necessary evil, 53–59 noise, 64–66

not making up for bad code, 55 obsolete, 286

poorly written, 287 proper use of, 54

**Index**

redundant, 60–62, 272, 275, 286–287 restating the obvious, 64

separated from code, 54 TODO, 58–59

too much information in, 70 venting in, 65

writing, 287

“communication gap”, minimizing, 168 Compare and Swap (CAS) operation,

327–328 ComparisonCompactor module, 252–265

defactored, 256–261 final, 263–265 interim, 261–263

original code, 254–256 compiler warnings, turning off, 289

complex code, demonstrating failures in, 341

complexity, managing, 139–140 computer science (CS) terms, using for

names, 27 concepts

keeping close to each other, 80 naming, 19

one word per, 26

separating at different levels, 290 spelling similar similarly, 20 vertical openness between, 78–79

conceptual affinity, of code, 84 concerns

cross-cutting, 160–161 separating, 154, 166, 178, 250

concrete classes, 149 concrete details, 149 concrete terms, 94 concurrency

defense principles, 180–182 issues, 190

motives for adopting, 178–179 myths and misconceptions about,

179–180 concurrency code

compared to nonconcurrency-related code, 181

focusing, 321

<https://www.facebook.com/groups/phattrienlaptrinh>

**Index**

concurrent algorithms, 179 concurrent applications, partition

behavior, 183 concurrent code

breaking, 329–333

defending from problems of, 180 flaws hiding in, 188

concurrent programming, 180

*Concurrent Programming in Java: Design Principles and Patterns*, 182, 342

concurrent programs, 178 concurrent update problems, 341

ConcurrentHashMap implementation, 183 conditionals

avoiding negative, 302 encapsulating, 257–258, 301

configurable data, 306 configuration constants, 306 consequences, warning of, 58 consistency

in code, 292 of enums, 278 in names, 40

consistent conventions, 259 constants

versus enums, 308–309 hiding, 308

inheriting, 271, 307–308

keeping at the appropriate level, 83 leaving as raw numbers, 300

not inheriting, 307–308 passing as symbols, 276 turning into enums, 275–276

construction

moving all to main, 155, 156 separating with factory, 156 of a system, 154

constructor arguments, 157 constructors, overloading, 25 consumer threads, 184 ConTest tool, 190, 342 context

adding meaningful, 27–29 not adding gratuitous, 29–30

providing with exceptions, 107

417

continuous readers, 184

control variables, within loop statements, 80–81

convenient idioms, 155 convention(s)

following standard, 299–300 over configuration, 164 structure over, 301

using consistent, 259 convoluted code, 175 copyright statements, 55 cosmic-rays. *See* one-offs CountDownLatch class, 183

coupling. *See also* decoupling; temporal coupling; tight coupling

artificial, 293

hidden temporal, 302–303 lack of, 150

coverage patterns, testing, 314 coverage tools, 313

“crisp abstraction”, 8–9 cross-cutting concerns, 160 Cunningham, Ward, 11–12 cuteness, in code, 26

**D**

dangling false argument, 294 data

abstraction, 93–95 copies of, 181–182 encapsulation, 181

limiting the scope of, 181 sets processed in parallel, 179 types, 97, 101

data structures. *See also* structure(s) compared to objects, 95, 97 defined, 95

interfaces representing, 94 treating Active Records as, 101

data transfer-objects (DTOs), 100–101, 160

database normal forms, 48 DateInterval enum, 282–283 DAY enumeration, 277

<https://www.facebook.com/groups/phattrienlaptrinh>

418

DayDate class, running SerialDate as, 271

DayDateFactory, 273–274 dead code, 288, 292 dead functions, 288 deadlock, 183, 335–339

deadly embrace. *See* circular wait debugging, finding deadlocks, 336 decision making, optimizing, 167–168 decisions, postponing, 168 declarations, unaligned, 87–88 DECORATOR objects, 164 DECORATOR pattern, 274 decoupled architecture, 167 decoupling, from construction

details, 156

decoupling strategy, concurrency as, 178

default constructor, deleting, 276 degradation, preventing, 14 deletions, as the majority of

changes, 250

density, vertical in code, 79–80 dependencies

finding and breaking, 250 injecting, 157

logical, 282

making logical physical, 298–299 between methods, 329–333 between synchronized

methods, 185 Dependency Injection (DI), 157

Dependency Inversion Principle (DIP), 15, 150

dependency magnet, 47

dependent functions, formatting, 82–83 derivatives

base classes depending on, 291 base classes knowing about, 273 of the exception class, 48 moving set functions into, 232,

233–235

pushing functionality into, 217

**Index**

description

of a class, 138

overloading the structure of code into, 310

descriptive names choosing, 309–310 using, 39–40

design(s)

of concurrent algorithms, 179 minimally coupled, 167 principles of, 15

design patterns, 290

details, paying attention to, 8 DI (Dependency Injection), 157 Dijkstra, Edsger, 48

dining philosophers execution model, 184–185

DIP (Dependency Inversion Principle), 15, 150

dirty code. *See also* bad code; messy code

dirty code, cleaning, 200 dirty tests, 123

disinformation, avoiding, 19–20 distance, vertical in code, 80–84 distinctions, making meaningful, 20–21 domain-specific languages (DSLs),

168–169

domain-specific testing language, 127 DoubleArgumentMarshalerclass, 238 DRY principle (Don't Repeat Yourself),

181, 289

DTOs (data transfer objects), 100–101, 160 dummy scopes, 90

duplicate if statements, 276 duplication

of code, 48

in code, 289–290 eliminating, 173–175 focusing on, 10 forms of, 173, 290 reduction of, 48

strategies for eliminating, 48

<https://www.facebook.com/groups/phattrienlaptrinh>

**Index**

dyadic argument, 40 dyadic functions, 42 dynamic proxies, 161

**E**

e, as a variable name, 22 Eclipse, 26

edit sessions, playing back, 13–14 efficiency, of code, 7

EJB architecture, early as over-engineered, 167

EJB standard, complete overhaul of, 164 EJB2 beans, 160

EJB3, Bank object rewritten in, 165–166 “elegant” code, 7

emergent design, 171–176 encapsulation, 136

of boundary conditions, 304 breaking, 106–107

of conditionals, 301

encodings, avoiding, 23–24, 312–313 entity bean, 158–160

enum(s)

changing MonthConstants to, 272 using, 308–309

enumeration, moving, 277 environment, heuristics on, 287 environment control system, 128–129 envying, the scope of a class, 293 error check, hiding a side effect, 258 Error class, 47–48

error code constants, 198–200 error codes

implying a class or enum, 47–48 preferring exceptions to, 46 returning, 103–104

reusing old, 48

separating from the Args module, 242–250

error detection, pushing to the edges, 109 error flags, 103–104

error handling, 8, 47–48

419

error messages, 107, 250

error processing, testing, 238–239 errorMessage method, 250

errors. *See also* boundary condition errors; spelling errors; string comparison errors

classifying, 107 Evans, Eric, 311

events, 41

exception classification, 107 exception clauses, 107–108 exception management code, 223 exceptions

instead of return codes, 103–105 narrowing the type of, 105–106 preferring to error codes, 46 providing context with, 107 separating from Args, 242–250 throwing, 104–105, 194 unchecked, 106–107

execution, possible paths of, 321–326 execution models, 183–185

Executor framework, 326–327 ExecutorClientScheduler.java, 321 explanation, of intent, 56–57 explanatory variables, 296–297 explicitness, of code, 19 expressive code, 295 expressiveness

in code, 10–11 ensuring, 175–176

Extract Method refactoring, 11 *Extreme Programming Adventures*

*in C#*, 10

*Extreme Programming Installed*, 10 “eye-full”, code fitting into, 79–80

**F**

factories, 155–156 factory classes, 273–275 failure

to express ourselves in code, 54

<https://www.facebook.com/groups/phattrienlaptrinh>

420

failure, *continued* patterns of, 314

tolerating with no harm, 330 false argument, 294

fast tests, 132

fast-running threads, starving longer running, 183

fear, of renaming, 30 Feathers, Michael, 10 feature envy

eliminating, 293–294 smelling of, 278

file size, in Java, 76 final keywords, 276

F.I.R.S.T. acronym, 132–133 First Law, of TDD, 122 FitNesse project

coding style for, 90 file sizes, 76, 77 function in, 32–33 invoking all tests, 224

flag arguments, 41, 288 focussed code, 8

foreign code. *See* third-party code formatting

horizontal, 85–90 purpose of, 76

Uncle Bob’s rules, 90–92 vertical, 76–85

formatting style, for a team of developers, 90

Fortran, forcing encodings, 23 Fowler, Martin, 285, 293 frame, 324

function arguments, 40–45 function call dependencies, 84–85 function headers, 70

function signature, 45

functionality, placement of, 295–296 functions

breaking into smaller, 141–146 calling within a block, 35 dead, 288

defining private, 292

**Index**

descending one level of abstraction, 304–306

doing one thing, 35–36, 302 dyadic, 42

eliminating extraneous if statements, 262

establishing the temporal nature of, 260

formatting dependent, 82–83 gathering beneath a banner, 67 heuristics on, 288

intention-revealing, 19 keeping small, 175 length of, 34–35 moving, 279

naming, 39, 297

number of arguments in, 288

one level of abstraction per, 36–37 in place of comments, 67 renaming for clarity, 258 rewriting for clarity, 258–259 sections within, 36

small as better, 34

structured programming with, 49 understanding, 297–298

as verbs of a language, 49 writing, 49

futures, 326

**G**

Gamma, Eric, 252

general heuristics, 288–307 generated byte-code, 180

generics, improving code readability, 115 get functions, 218

getBoolean function, 224 GETFIELD instruction, 325, 326 getNextIdmethod, 326 getState function, 129 Gilbert, David, 267, 268

given-when-then convention, 130 glitches. *See* one-offs

<https://www.facebook.com/groups/phattrienlaptrinh>

**Index**

global setup strategy, 155 “God class”, 136–137 good comments, 55–59

goto statements, avoiding, 48, 49 grand redesign, 5

gratuitous context, 29–30

**H**

hand-coded instrumentation, 189 HashTable, 328–329

headers. *See* comment headers; function headers

heuristics

cross references of, 286, 409 general, 288–307

listing of, 285–314

hidden temporal coupling, 259, 302–303 hidden things, in a function, 44

hiding

implementation, 94 structures, 99

hierarchy of scopes, 88

HN. *See* Hungarian Notation horizontal alignment, of code, 87–88 horizontal formatting, 85–90 horizontal white space, 86

HTML, in source code, 69

Hungarian Notation (HN), 23–24, 295 Hunt, Andy, 8, 289

hybrid structures, 99

**I**

if statements duplicate, 276 eliminating, 262

if-else chain

appearing again and again, 290 eliminating, 233

ignored tests, 313 implementation

duplication of, 173 encoding, 24

421

exposing, 94 hiding, 94

wrapping an abstraction, 11 *Implementation Patterns*, 3, 296 implicity, of code, 18

import lists

avoiding long, 307 shortening in SerialDate, 270

imports, as hard dependencies, 307 imprecision, in code, 301 inaccurate comments, 54 inappropriate information, in

comments, 286 inappropriate static methods, 296 include method, 48 inconsistency, in code, 292 inconsistent spellings, 20 incrementalism, 212–214

indent level, of a function, 35 indentation, of code, 88–89 indentation rules, 89 independent tests, 132 information

inappropriate, 286

too much, 70, 291–292 informative comments, 56 inheritance hierarchy, 308

inobvious connection, between a comment and code, 70

input arguments, 41 instance variables

in classes, 140 declaring, 81

hiding the declaration of, 81–82 passing as function

arguments, 231 proliferation of, 140

instrumented classes, 342 insufficient tests, 313 integer argument(s)

defining, 194 integrating, 224–225

integer argument functionality,

moving into ArgumentMarshaler, 215–216

<https://www.facebook.com/groups/phattrienlaptrinh>

422

integer argument type, adding to Args, 212

integers, pattern of changes for, 220 IntelliJ, 26

intent

explaining in code, 55 explanation of, 56–57 obscured, 295

intention-revealing function, 19 intention-revealing names, 18–19 interface(s)

defining local or remote, 158–160 encoding, 24

implementing, 149–150 representing abstract concerns, 150 turning ArgumentMarshaler into, 237 well-defined, 291–292

writing, 119

internal structures, objects hiding, 97 intersection, of domains, 160 intuition, not relying on, 289 inventor of C++, 7

Inversion of Control (IoC), 157 InvocationHandler object, 162 I/O bound, 318

isolating, from change, 149–150 isxxxArg methods, 221–222 iterative process, refactoring as, 265

**J**

jar files, deploying derivatives and bases in, 291

Java

aspects or aspect-like mechanisms, 161–166

heuristics on, 307–309

as a wordy language, 200

Java 5, improvements for concurrent development, 182–183

Java 5 Executor framework, 320–321 Java 5 VM, nonblocking solutions in,

327–328

Java AOP frameworks, 163–166

**Index**

Java programmers, encoding not needed, 24

Java proxies, 161–163 Java source files, 76–77 javadocs

as clutter, 276

in nonpublic code, 71 preserving formatting in, 270 in public APIs, 59

requiring for every function, 63 java.util.concurrentpackage, collections

in, 182–183

JBoss AOP, proxies in, 163 JCommon library, 267 JCommon unit tests, 270 JDepend project, 76, 77

JDK proxy, providing persistence support, 161–163

Jeffries, Ron, 10–11, 289 jiggling strategies, 190 JNDI lookups, 157 journal comments, 63–64 JUnit, 34

JUnit framework, 252–265 Junit project, 76, 77

Just-In-Time Compiler, 180

**K**

keyword form, of a function name, 43

**L**

L, lower-case in variable names, 20 language design, art of programming as, 49 languages

appearing to be simple, 12 level of abstraction, 2

multiple in one source file, 288 multiples in a comment, 270

last-in, first-out (LIFO) data structure, operand stack as, 324

Law of Demeter, 97–98, 306

<https://www.facebook.com/groups/phattrienlaptrinh>

**Index**

LAZY INITIALIZATION/ EVALUATION idiom, 154

LAZY-INITIALIZATION, 157 Lea, Doug, 182, 342

learning tests, 116, 118 LeBlanc’s law, 4 legacy code, 307

legal comments, 55–56 level of abstraction, 36–37 levels of detail, 99

lexicon, having a consistent, 26 lines of code

duplicating, 173 width of, 85

list(s)

of arguments, 43

meaning specific to programmers, 19 returning a predefined immutable, 110

literate code, 9

literate programming, 9 *Literate Programming*, 141 livelock, 183, 338

local comments, 69–70 local variables, 324

declaring, 292

at the top of each function, 80 lock & wait, 337, 338

locks, introducing, 185 log4j package, 116–118

logical dependencies, 282, 298–299 LOGO language, 36

long descriptive names, 39

long names, for long scopes, 312

loop counters, single-letter names for, 25

**M**

magic numbers obscuring intent, 295

replacing with named constants, 300–301

main function, moving construction to, 155, 156

423

managers, role of, 6 mandated comments, 63

manual control, over a serial ID, 272

Map

adding for ArgumentMarshaler, 221 methods of, 114

maps, breaking the use of, 222–223 marshalling implementation,

214–215 meaningful context, 27–29 member variables

f prefix for, 257 prefixing, 24

renaming for clarity, 259 mental mapping, avoiding, 25

messy code. *See also* bad code; dirty code total cost of owning, 4–12

method invocations, 324 method names, 25 methods

affecting the order of execution, 188 calling a twin with a flag, 278 changing from static to instance, 280 of classes, 140

dependencies between, 329–333 eliminating duplication between,

173–174

minimizing assert statements in, 176 naming, 25

tests exposing bugs in, 269 minimal code, 9

misleading comments, 63

misplaced responsibility, 295–296, 299 MOCK OBJECT, assigning, 155 monadic argument, 40

monadic forms, of arguments, 41 monads, converting dyads into, 42 Monte Carlo testing, 341

Month enum, 278 MonthConstants class, 271 multithread aware, 332

multithread-calculation, of throughput, 335

<https://www.facebook.com/groups/phattrienlaptrinh>

424

multithreaded code, 188, 339–342 mumbling, 59–60

mutators, naming, 25

mutual exclusion, 183, 336, 337

**N**

named constants, replacing magic numbers, 300–301

name-length-challenged languages, 23 names

abstractions, appropriate level of, 311 changing, 40

choosing, 175, 309–310 of classes, 270–271 clever, 26

descriptive, 39–40 of functions, 297

heuristics on, 309–313 importance of, 309–310 intention-revealing, 18–19 length of corresponding to scope,

22–23

long names for long scopes, 312 making unambiguous, 258 problem domain, 27 pronounceable, 21–22

rules for creating, 18–30 searchable, 22–23

shorter generally better than longer, 30 solution domain, 27

with subtle differences, 20 unambiguous, 312

at the wrong level of abstraction, 271 naming, classes, 138

naming conventions, as inferior to structures, 301

navigational methods, in Active Records, 101

near bugs, testing, 314

negative conditionals, avoiding, 302 negatives, 258

nested structures, 46

**Index**

Newkirk, Jim, 116 newspaper metaphor, 77–78 niladic argument, 40

no preemption, 337 noise

comments, 64–66 scary, 66

words, 21

nomenclature, using standard, 311–312 nonblocking solutions, 327–328 nonconcurrency-related code, 181 noninformative names, 21

nonlocal information, 69–70 nonpublic code, javadocs in, 71

nonstatic methods, preferred to static, 296 nonthreaded code, getting working

first, 187

nonthread-safe classes, 328–329 normal flow, 109

null

not passing into methods, 111–112 not returning, 109–110

passed by a caller accidentally, 111 null detection logic, for ArgumentMarshaler,

214 NullPointerException, 110, 111 number-series naming, 21

**O**

*Object Oriented Analysis and Design with Applications*, 8

object-oriented design, 15 objects

compared to data structures, 95, 97 compared to data types and proce-

dures, 101 copying read-only, 181 defined, 95

obscured intent, 295 obsolete comments, 286 obvious behavior, 288–289 obvious code, 12

<https://www.facebook.com/groups/phattrienlaptrinh>

**Index**

“Once and only once” principle, 289 “ONE SWITCH” rule, 299

one thing, functions doing, 35–36, 302 one-offs, 180, 187, 191

OO code, 97 OO design, 139

Open Closed Principle (OCP), 15, 38 by checked exceptions, 106 supporting, 149

operand stack, 324

operating systems, threading policies, 188 operators, precedence of, 86

optimistic locking, 327

optimizations, LAZY-EVALUATION as, 157

optimizing, decision making, 167–168 orderings, calculating the possible, 322–323 organization

for change, 147–150 of classes, 136

managing complexity, 139–140 outbound tests, exercising an interface, 118 output arguments, 41, 288

avoiding, 45

need for disappearing, 45 outputs, arguments as, 45

overhead, incurred by concurrency, 179 overloading, of code with description, 310

**P**

paperback model, as an academic model, 27

parameters, taken by instructions, 324 parse operation, throwing an

exception, 220 partitioning, 250

paths of execution, 321–326

pathways, through critical sections, 188 pattern names, using standard, 175 patterns

of failure, 314

as one kind of standard, 311

425

performance

of a client/server pair, 318 concurrency improving, 179 of server-based locking, 333

permutations, calculating, 323 persistence, 160, 161 pessimistic locking, 327

phraseology, in similar names, 40 physicalizing, a dependency, 299 Plain-Old Java Objects. *See* POJOs platforms, running threaded code, 188 pleasing code, 7

pluggable thread-based code, 187 POJO system, agility provided by, 168 POJOs (Plain-Old Java Objects)

creating, 187

implementing business logic, 162 separating threaded-aware code, 190 in Spring, 163

writing application domain logic, 166 polyadic argument, 40

polymorphic behavior, of functions, 296 polymorphic changes, 96–97 polymorphism, 37, 299

position markers, 67 positives

as easier to understand, 258 expressing conditionals as, 302 of decisions, 301precision

as the point of all naming, 30 predicates, naming, 25 preemption, breaking, 338 prefixes

for member variables, 24

as useless in today’s environments, 312–313

pre-increment operator, ++, 324, 325, 326 “prequel”, this book as, 15

principle of least surprise, 288–289, 295 principles, of design, 15

PrintPrimes program, translation into Java, 141

private behavior, isolating, 148–149

<https://www.facebook.com/groups/phattrienlaptrinh>

426

private functions, 292

private method behavior, 147 problem domain names, 27 procedural code, 97

procedural shape example, 95–96 procedures, compared to objects, 101 process function, repartitioning, 319–320 process method, I/O bound, 319 processes, competing for resources, 184 processor bound, code as, 318

producer consumer execution model, 184 producer threads, 184

production environment, 127–130 productivity, decreased by messy code, 4 professional programmer, 25 professional review, of code, 268 programmers

as authors, 13–14 conundrum faced by, 6

responsibility for messes, 5–6 unprofessional, 5–6

programming defined, 2 structured, 48–49

programs, getting them to work, 201 pronounceable names, 21–22 protected variables, avoiding, 80 proxies, drawbacks of, 163

public APIs, javadocs in, 59 puns, avoiding, 26–27

PUTFIELD instruction, as atomic, 325

**Q**

queries, separating from commands, 45–46

**R**

random jiggling, tests running, 190 range, including end-point dates in, 276 readability

of clean tests, 124 of code, 76

**Index**

Dave Thomas on, 9 improving using generics, 115

readability perspective, 8 readers

of code, 13–14 continuous, 184

readers-writers execution model, 184 reading

clean code, 8

code from top to bottom, 37 versus writing, 14

reboots, as a lock up solution, 331 recommendations, in this book, 13 redesign, demanded by the team, 5 redundancy, of noise words, 21 redundant comments, 60–62, 272, 275,

286–287 ReentrantLock class, 183

refactored programs, as longer, 146 refactoring

Args, 212

code incrementally, 172

as an iterative process, 265 putting things in to take out, 233 test code, 127

*Refactoring* (Fowler), 285 renaming, fear of, 30

repeatability, of concurrency bugs, 180 repeatable tests, 132

requirements, specifying, 2

resetId, byte-code generated for, 324–325 resources

bound, 183

processes competing for, 184

threads agreeing on a global ordering of, 338

responsibilities

counting in classes, 136 definition of, 138 identifying, 139 misplaced, 295–296, 299

splitting a program into main, 146 return codes, using exceptions instead,

103–105

<https://www.facebook.com/groups/phattrienlaptrinh>

**Index**

reuse, 174

risk of change, reducing, 147 robust clear code, writing, 112 rough drafts, writing, 200 runnable interface, 326

run-on expressions, 295

run-on journal entries, 63–64

runtime logic, separating startup from, 154

**S**

safety mechanisms, overridden, 289 scaling up, 157–161

scary noise, 66 schema, of a class, 194

schools of thought, about clean code, 12–13

scissors rule, in C++, 81 scope(s)

defined by exceptions, 105 dummy, 90

envying, 293

expanding and indenting, 89 hierarchy in a source file, 88 limiting for data, 181

names related to the length of, 22–23, 312

of shared variables, 333 searchable names, 22–23 Second Law, of TDD, 122 sections, within functions, 36

selector arguments, avoiding, 294–295 self validating tests, 132 Semaphoreclass, 183

semicolon, making visible, 90

“serial number”, SerialDate using, 271 SerialDateclass

making it right, 270–284 naming of, 270–271 refactoring, 267–284

SerialDateTests class, 268 serialization, 272

server, threads created by, 319–321

427

server application, 317–318, 343–344 server code, responsibilities of, 319 server-based locking, 329

as preferred, 332–333

with synchronized methods, 185 “Servlet” model, of Web applications, 178 Servlets, synchronization problems, 182 set functions, moving into appropriate

derivatives, 232, 233–235 setArgument, changing, 232–233 setBoolean function, 217

setter methods, injecting dependencies, 157

setup strategy, 155 SetupTeardownIncluder.java listing,

50–52 shape classes, 95–96

shared data, limiting access, 181 shared variables

method updating, 328 reducing the scope of, 333

shotgun approach, hand-coded instrumen-tation as, 189

shut-down code, 186 shutdowns, graceful, 186 side effects

having none, 44 names describing, 313

Simmons, Robert, 276 simple code, 10, 12

Simple Design, rules of, 171–176 simplicity, of code, 18, 19

single assert rule, 130–131

single concepts, in each test function, 131–132

Single Responsibility Principle (SRP), 15, 138–140

applying, 321 breaking, 155

as a concurrency defense principle, 181

recognizing violations of, 174 server violating, 320

<https://www.facebook.com/groups/phattrienlaptrinh>

428

Single Responsibility Principle (SRP), *continued*

Sql class violating, 147 supporting, 157

in test classes conforming to, 172 violating, 38

single value, ordered components of, 42 single-letter names, 22, 25

single-thread calculation, of throughput, 334

SINGLETON pattern, 274 small classes, 136

*Smalltalk Best Practice Patterns*, 296 smart programmer, 25

software project, maintenance of, 175 software systems. *See also* system(s)

compared to physical systems, 158 SOLID class design principle, 150 solution domain names, 27

source code control systems, 64, 68, 69 source files

compared to newspaper articles, 77–78

multiple languages in, 288 *Sparkle* program, 34

spawned threads, deadlocked, 186 special case objects, 110 SPECIAL CASE PATTERN, 109 specifications, purpose of, 2 spelling errors, correcting, 20 SpreadsheetDateFactory, 274–275 Spring AOP, proxies in, 163 Spring Framework, 157

Spring model, following EJB3, 165 Spring V2.5 configuration file, 163–164 spurious failures, 187

Sql class, changing, 147–149 square root, as the iteration limit, 74

SRP. *See* Single Responsibility Principle standard conventions, 299–300

standard nomenclature, 175, 311–312 standards, using wisely, 168

startup process, separating from runtime logic, 154

**Index**

starvation, 183, 184, 338 static function, 279 static import, 308

static methods, inappropriate, 296 *The Step-down Rule*, 37

stories, implementing only today’s, 158 STRATEGY pattern, 290 stringarguments, 194, 208–212, 214–225 string comparison errors, 252 StringBuffers, 129

Stroustrup, Bjarne, 7–8

structure(s). *See also* data structures hiding, 99

hybrid, 99

making massive changes to, 212 over convention, 301

structured programming, 48–49 SuperDashboard class, 136–137 swapping, as permutations, 323 switch statements

burying, 37, 38 considering polymorphism

before, 299

reasons to tolerate, 38–39 switch/casechain, 290

synchronization problems, avoiding with Servlets, 182

synchronized block, 334 synchronized keyword, 185

adding, 323

always acquiring a lock, 328 introducing a lock via, 331 protecting a critical section

in code, 181 synchronized methods, 185 synchronizing, avoiding, 182 synthesis functions, 265

system(s). *See also* software systems file sizes of significant, 77

keeping running during development, 213

needing domain-specific, 168 system architecture, test driving,

166–167

<https://www.facebook.com/groups/phattrienlaptrinh>

**Index**

system failures, not ignoring one-offs, 187

system level, staying clean at, 154 system-wide information, in a local

comment, 69–70

**T**

tables, moving, 275

target deployment platforms, running tests on, 341

task swapping, encouraging, 188 TDD (Test Driven Development), 213

building logic, 106

as fundamental discipline, 9 laws of, 122–123

team rules, 90 teams

coding standard for every, 299–300 slowed by messy code, 4

technical names, choosing, 27 technical notes, reserving comments

for, 286

TEMPLATE METHOD pattern addressing duplication, 290 removing higher-level duplication,

174–175 using, 130

temporal coupling. *See also* coupling exposing, 259–260

hidden, 302–303

side effect creating, 44

temporary variables, explaining, 279–281 test cases

adding to check arguments, 237 in ComparisonCompactor, 252–254 patterns of failure, 269, 314 turning off, 58

test code, 124, 127

TEST DOUBLE, assigning, 155 Test Driven Development. *See* TDD test driving, architecture, 166–167 test environment, 127–130

429

test functions, single concepts in, 131–132 test implementation, of an interface, 150 test suite

automated, 213

of unit tests, 124, 268 verifying precise behavior, 146

testable systems, 172

test-driven development. *See* TDD testing

arguments making harder, 40 construction logic mixed with

runtime, 155

testing language, domain-specific, 127 testNG project, 76, 77

tests

clean, 124–127 cleanliness tied to, 9

commented out for SerialDate, 268–270

dirty, 123

enabling the -ilities, 124 fast, 132

fast versus slow, 314 heuristics on, 313–314 ignored, 313 independent, 132 insufficient, 313 keeping clean, 123–124

minimizing assert statements in, 130–131

not stopping trivial, 313 refactoring, 126–127 repeatable, 132

requiring more than one step, 287 running, 341

self validating, 132

simple design running all, 172 suite of automated, 213 timely, 133

writing for multithreaded code, 339–342

writing for threaded code, 186–190 writing good, 122–123

<https://www.facebook.com/groups/phattrienlaptrinh>

430

Third Law, of TDD, 122 third-party code

integrating, 116 learning, 116 using, 114–115

writing tests for, 116 this variable, 324 Thomas, Dave, 8, 9, 289 thread(s)

adding to a method, 322 interfering with each other, 330 making as independent as

possible, 182

stepping on each other, 180, 326 taking resources from other

threads, 338

thread management strategy, 320 thread pools, 326

thread-based code, testing, 342 threaded code

making pluggable, 187 making tunable, 187–188 symptoms of bugs in, 187 testing, 186–190

writing in Java 5, 182–183 threading

adding to a client/server application, 319, 346–347

problems in complex systems, 342 thread-safe collections, 182–183, 329 throughput

causing starvation, 184 improving, 319 increasing, 333–335 validating, 318

throws clause, 106 tiger team, 5

tight coupling, 172 time, taking to go fast, 6

Time and Money project, 76 file sizes, 77

timely tests, 133

**Index**

timer program, testing, 121–122 “TO” keyword, 36

TO paragraphs, 37 TODO comments, 58–59

tokens, used as magic numbers, 300 Tomcat project, 76, 77

tools

ConTest tool, 190, 342 coverage, 313

handling proxy boilerplate, 163 testing thread-based code, 342

train wrecks, 98–99 transformations, as return values, 41

transitive navigation, avoiding, 306–307 triadic argument, 40

triads, 42

try blocks, 105

try/catch blocks, 46–47, 65–66

try-catch-finally statement, 105–106 tunable threaded-based code, 187–188 type encoding, 24

**U**

ubiquitous language, 311–312 unambiguous names, 312 unchecked exceptions, 106–107

unencapsulated conditional, encapsulating, 257

unit testing, isolated as difficult, 160 unit tests, 124, 175, 268 unprofessional programming, 5–6 uppercase C, in variable names, 20 usability, of newspapers, 78

use, of a system, 154

users, handling concurrently, 179

**V**

validation, of throughput, 318 variable names, single-letter, 25

<https://www.facebook.com/groups/phattrienlaptrinh>

**Index**

variables

1 based versus zero based, 261 declaring, 80, 81, 292 explaining temporary, 279–281 explanatory, 296–297

keeping private, 93 local, 292, 324

moving to a different class, 273 in place of comments, 67 promoting to instance variables of

classes, 141

with unclear context, 28 venting, in comments, 65 verbs, keywords and, 43 Version class, 139

versions, not deserializing across, 272 vertical density, in code, 79–80 vertical distance, in code, 80–84 vertical formatting, 76–85

vertical openness, between concepts, 78–79

vertical ordering, in code, 84–85 vertical separation, 292

431

**W**

wading, through bad code, 3

Web containers, decoupling provided by, 178

what, decoupling from when, 178 white space, use of horizontal, 86 wildcards, 307

*Working Effectively with Legacy Code*, 10

“working” programs, 201 workmanship, 176 wrappers, 108

wrapping, 108

writers, starvation of, 184 “Writing Shy Code”, 306

**X**

XML

deployment descriptors, 160 “policy” specified configuration

files, 164

<https://www.facebook.com/groups/phattrienlaptrinh>