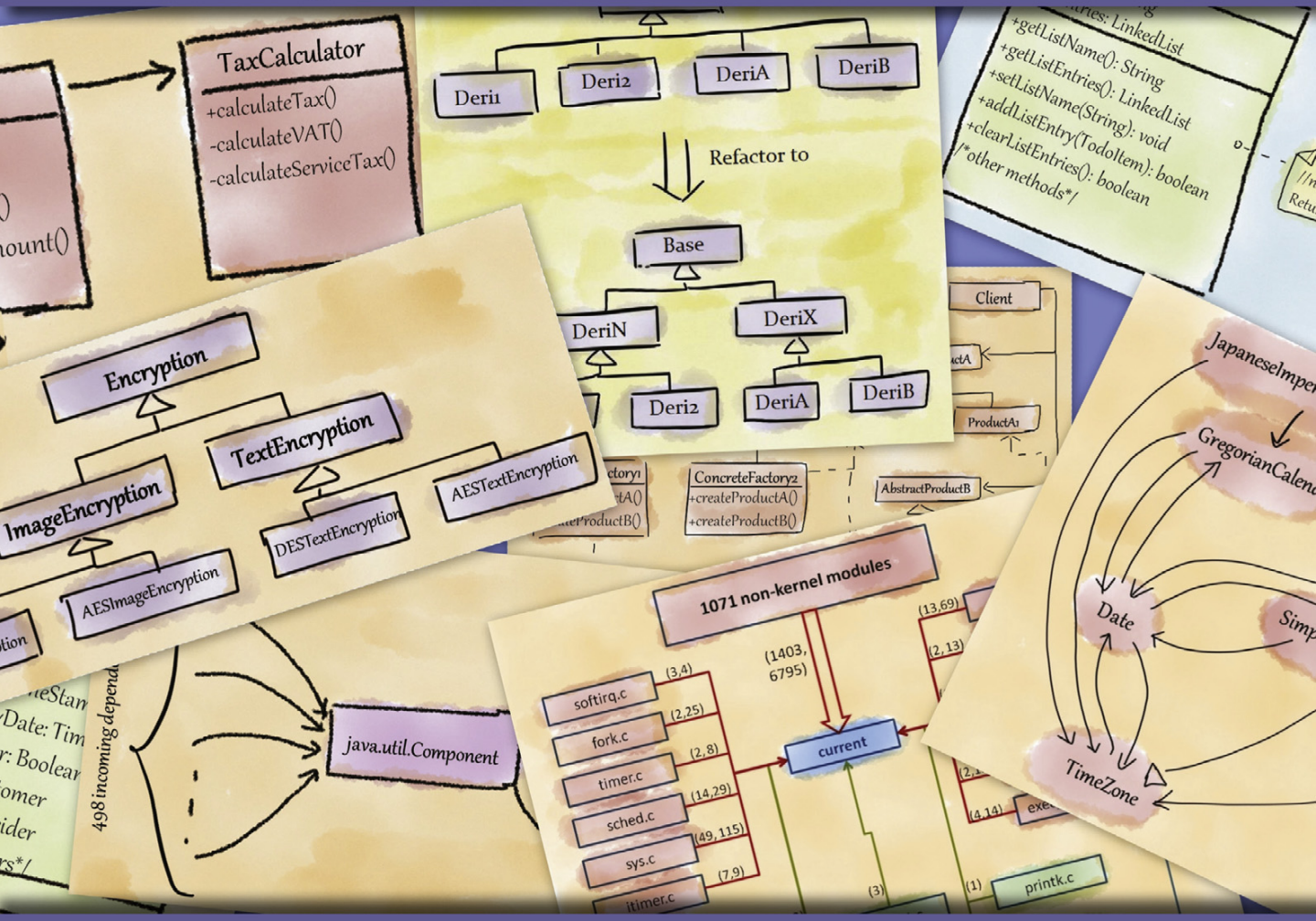
Refactoring for Software Design Smells

Managing Technical Debt



Girish Suryanarayana,
Ganesh Samarthyam, Tushar Sharma

Forewords by Grady Booch and Stéphane Ducasse

MORGAN KAUFMANN

Refactoring for Software Design Smells

Refactoring for Software Design Smells

Managing Technical Debt

Girish Suryanarayana

Ganesh Samarthyam

Tushar Sharma

ELSEVIER

AMSTERDAM • BOSTON • HEIDELBERG • LONDON
NEW YORK • OXFORD • PARIS • SAN DIEGO
SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO

Morgan Kaufmann is an imprint of Elsevier

MK
MORGAN KAUFMANN

Acquiring Editor: Todd Green
Editorial Project Manager: Lindsay Lawrence
Project Manager: Punithavathy Govindaradjane
Designer: Mark Rogers

Morgan Kaufmann is an imprint of Elsevier
225 Wyman Street, Waltham, MA, 02451, USA

Notices

Knowledge and best practice in this field are constantly changing. As new research and experience broaden our understanding, changes in research methods, professional practices, or medical treatment may become necessary.

Practitioners and researchers must always rely on their own experience and knowledge in evaluating and using any information, methods, compounds, or experiments described herein. In using such information or methods they should be mindful of their own safety and the safety of others, including parties for whom they have a professional responsibility.

To the fullest extent of the law, neither the Publisher nor the authors, contributors, or editors, assume any liability for any injury and/or damage to persons or property as a matter of products liability, negligence or otherwise, or from any use or operation of any methods, products, instructions, or ideas contained in the material herein.

Library of Congress Cataloging-in-Publication Data
Suryanarayana, Girish
Refactoring for software design smells: managing technical debt/Girish Suryanarayana, Ganesh Samarthyam, Tushar Sharma.
pages cm
Includes bibliographical references and index.
ISBN: 978-0-12-801397-7 (paperback)
1. Software refactoring. 2. Software failures. I. Samarthyam, Ganesh. II. Sharma, Tushar. III. Title.

QA76.76.R42S86 2015
005.1′6–dc23

2014029955

British Library Cataloguing-in-Publication Data
A catalogue record for this book is available from the British Library.

ISBN: 978-0-12-801397-7

Contents

Foreword by Grady Booch

There is a wonderful book by Anique Hommels called *Unbuilding Cities: Obduracy in Urban Sociotechnical Change* that speaks of the technical, social, economic, and political issues of urban renewal. Cities grow, cities evolve, cities have parts that simply die while other parts flourish; each city has to be renewed in order to meet the needs of its populace. From time to time cities are intentionally refactored (many cities in Europe were devastated in World War II and so had to be rebuilt); most times cities are refactored in bits (the Canary Warf renewal in London comes to mind), but sometimes the literal and figurative debt is so great a once great city falls into despair (present day Detroit, for example).

Software-intensive systems are like that. They grow, they evolve, sometimes they wither away, and sometimes they flourish. The concept of technical debt is central to understanding the forces that weigh upon systems, for it often explains where, how, and why a system is stressed. In cities, repairs on infrastructure are often delayed and incremental changes are made rather than bold ones. So it is again in software-intensive systems. Users suffer the consequences of capricious complexity, delayed improvements, and insufficient incremental change; the developers who evolve such systems suffer the slings and arrows of never being able to write quality code because they are always trying to catch up.

What delights me about this present book is its focus on technical debt and refactoring as the actionable means to attend to it.

When you have got a nagging tiny gas leak on a city block, a literal smell will lead you to the underlying cause. Software-intensive systems are like that as well, although the smells one may observe therein are far more subtle and invisible to all the senses save to the clever cognitive ones. As you read on in this book, I think you will find one of the more interesting and complex expositions of software smells you will ever find. Indeed, you have in your hand a veritable field guide of smells, sort of like the very real *Birds of the West Indies*, except about software, not sparrows.

Abstraction, encapsulation, modularization, and hierarchy: these are all elements fundamental to software engineering best practices, and you will see these highlights explained as well. In all, I think you will find this a delightful, engaging, actionable read.

Grady Booch
IBM Fellow and Chief Scientist for
Software Engineering, IBM Research

Foreword by Dr. Stéphane Ducasse

Working with real and large object-oriented applications made me and my coauthors (O. Nierstrasz and S. Demeyer) think about object-oriented reengineering patterns. Our idea was to record good approaches to handle these systems at the design level but also with a bit of process. This is how Object-Oriented Reengineering Patterns (http://scg.unibe.ch/download/oorp/) came to life after 3 years of intensive work. In the same period I reviewed the book "Refactoring: Improving the Design of Existing Code" by Martin Fowler because we were working on language-independent refactorings. To me, books like these are milestones. They create a foundation for our engineering science.

When as JOT associate editor, I saw the authors' article describing the work presented in this book, I became an immediate supporter. I immediately asked if they would write a book and if I could be a reviewer. Why? Because a good book on Smells was still missing. And we daily face code smells. "Refactoring for Software Design Smells" is an excellent book. It is another milestone that professionals will use. It captures the deep knowledge experts have when facing real code. It puts in shape and word the complex knowledge that experts acquire after years of experience. I daily program either Pharo runtime and libraries or the Moose software and data analysis platform, and there is not a single day where I'm not challenged by design. Why? Because design, real design, is not simple. Even with a lot of practice and experience design is challenging. This is why having good abstractions to play with is important. Patterns and smells are really effective ways to capture such experience and reflection about the field.

I like the idea that I have access to excellent books that I can suggest to my students, colleagues, and friends. I like books that fill you up and make you think about your own practices. There are few of such ones and they are like gems. And "Refactoring for Software Design Smells" is one of those.

It is the book I would have loved to write (but I probably would not have been able to) as a complement to the Fowler's refactoring book, and my Object-Oriented Reengineering Patterns book. I'm sure that you will learn a lot from it and that you will enjoy it. I want to thank Girish, Ganesh, and Tushar (the authors of this book) to have spent their time to write it. I know that the process is long and tedious but this is an important part of our culture and knowledge that they capture and offer to you. I also want to thank them for their invitation to write this foreword.

Stéphane Ducasse

(Dr. Stéphane Ducasse is an expert in and fond of object design, language design, and reflective programming. He has contributed to Traits which got introduced in Pharo, Perl-6, PHP 5.4, and Squeak and influenced Fortress and Scala. He is one of the lead developers of Pharo (http://www.pharo.project.org/), an open-source live programming language, IDE and platform. He is an expert in software analysis and reengineering. He is one of the developers of the Moose analysis platform http://www.moosetechnology.org and he recently cocreated http://www.synectique.eu, a company delivering advanced tools adapted to client problems.)

Preface

As a program is evolved its complexity increases unless work is done to maintain or reduce it.

Lehman's law of Increasing Complexity [1]

WHAT IS THIS BOOK ABOUT?

Change is inevitable and difficult! This is true not only about life but also about software. Software is expected to evolve continuously to meet the ever-increasing demands of its users. At the same time, the intangible nature of software makes it difficult to manage this continuous change. What typically results is poor software quality[1] and a huge technical debt.

This book tells you how to improve software quality and reduce technical debt by discovering and addressing smells in your design. Borrowing a phrase from the health care domain "a good doctor is one who knows the medicines but a great doctor is one who knows the disease," our approach is grounded on the philosophy that "a good designer is one who knows about design solutions but a great designer is one who understands the problems (or smells) in the design, how they are caused, and how they can be addressed by applying proven and sound design principles." The goal of this book is, therefore, to guide you into becoming a better designer—one who can recognize and understand the "disease" in his design, and can treat it properly, thereby, improving the quality of the software and keeping technical debt under control.

WHAT DOES THIS BOOK COVER?

This book presents a catalog of 25 structural design smells and their corresponding refactoring towards managing technical debt. We believe that applying software design principles is the key to developing high-quality software. We have, therefore, organized our smell catalog around four basic design principles. Smells are named after the specific principle they violate. The description of each smell reveals the design principle that the smell violates, discusses some factors that can cause that smell to occur, and lists the key quality attributes that are impacted by the smell. This allows the reader to get an idea of the technical debt incurred by the design.

[1] Capers Jones [3] finds that poor software quality costs more than US $150 billion per year in the United States and greater than US $500 billion per year worldwide.

Each smell description also includes real-world examples as well as anecdotes based on experience with industrial projects. Accompanying each example are potential refactoring solutions that help address the particular instance of the smell. We believe that the impact of a smell can only be judged based on the design context in which it occurs. Therefore, we also explicitly consider situations wherein a smell may be purposely introduced either due to constraints (such as language or platform limitations) or to realize an overarching purpose in the design.

Smells can be found at different levels of granularity, including architecture and code. Similarly, smells can be either structural or behavioral in nature. Rather than surveying a wide range of smells pertaining to different levels of granularity and nature, we focus only on "structural" and "design-level" smells in this book. Further, the book discusses only those smells that are commonly found in real-world projects and have been documented in literature.

WHO SHOULD READ THIS BOOK?

Software Architects and Designers—If you are a practicing software architect or a designer, you will get the most out of this book. This book will benefit you in multiple ways. As an architect and designer, your primary responsibility is the software design and its quality, and you (more than anyone else) are striving to realize quality requirements in your software. The knowledge of design smells and the suggested refactorings covered in this book will certainly help you in this regard; *they offer you immediately usable ideas for improving the quality of your design*. Since this book explicitly describes the impact of smells on key quality attributes, you will gain a deeper appreciation of how even the smallest design decision has the potential to significantly impact the quality of your software. Through the real-world anecdotes and case studies, you will become aware of what factors you should be careful about and plan for in order to avoid smells.

Software Developers—We have observed in practice that often developers take shortcuts that seemingly get the work done but compromise on the design quality. We have also observed that sometimes when the design does not explicitly address certain aspects, it is the developer who ends up making key design decisions while coding. In such a case, if design principles are incorrectly applied or not applied at all, smells will arise. We believe that reading this book will help developers realize how their seemingly insignificant design decisions or shortcuts can have an immense impact on software quality. This realization will help them transform themselves into better developers who can detect and address problems in the design and code.

Project Managers—Project managers constantly worry about the possible schedule slippages and cost overruns of their projects. There is an increased awareness today that often the primary reason for this is technical debt, and many project managers are, therefore, always on the look out for solutions to reduce technical

debt. Such project managers will benefit from reading this book; they will gain a better understanding of the kinds of problems that manifest in the design and have an increased appreciation for refactoring.

Students—This book is very relevant to courses in computer science or software engineering that discuss software design. A central focus of any software design course is the fundamental principles that guide the modeling and design of high-quality software. One effective way to learn about these principles is to first study the effects (i.e., smells) of wrong application or misapplication of design principles and then learn about how to apply them properly (i.e., refactoring). We, therefore, believe that reading this book will help students appreciate the value of following good design principles and practices and prepare them to realize high-quality design in real-world projects post completion of their studies.

WHAT ARE THE PREREQUISITES FOR READING THIS BOOK?

We expect you to have basic knowledge of object-oriented programming and design and be familiar with at least one object-oriented language (such as C++, Java, or C#). We also expect you to have knowledge of object-oriented concepts such as class, abstract class, and interface (which can all be embraced under the umbrella terms "abstraction" or "type"), inheritance, delegation, composition, and polymorphism.

HOW TO READ THIS BOOK?

This book is logically structured into the following three parts:

- **Chapters 1 and 2** set the context for this book. Chapter 1 introduces the concept of technical debt, the factors that contribute to it, and its impact on software projects. Chapter 2 introduces design smells and describes the principle-based classification scheme that we have used to categorize and name design smells in this book.
- **Chapters 3–6** present the catalog of 25 design smells. The catalog is divided into four chapters that correspond to the four fundamental principles (abstraction, encapsulation, modularization, and hierarchy) that are violated by the smells. For each design smell, we provide illustrative examples, describe the impact of the smells on key quality attributes, and discuss the possible refactoring for that smell in the context of the given examples.
- **Chapters 7 and 8** present a reflection of our work and give you an idea about how to repay technical debt in your projects. Chapter 7 revisits the catalog to pick up examples and highlight the interplay between smells and the rest of the design in which the smells occur. Chapter 8 offers practical guidance and tips on how to approach refactoring of smells to manage technical debt in real-world projects.

The appendices contain a listing of design principles referenced in this book, a list of tools that can help detect and address design smells, a brief summary of the UML-like notations that we have used in this book, and a suggested reading list to help augment your knowledge of software design.

Given the fact that Java is the most widely used object-oriented language in the world today, we have provided coding examples in Java. However, the key take-away is in the context of design principles, and is therefore applicable to other object-oriented languages such as C++ and C# as well. Further, we have used simple UML-like class diagrams in this book which are explained in Appendix C.

Many of the examples discussed in this book are from JDK version 7.0. We have interpreted the smells in JDK based on the limited information we could glean by analyzing the source code and the comments. It is therefore possible that there may be perfectly acceptable reasons for the presence of these smells in JDK.

We have shared numerous anecdotes and case studies throughout this book, which have been collected from the following different sources:

- Experiences reported by participants of the online *Smells Forum*, which we established to collect smell stories from the community.
- Incidents and stories that have been shared with us by fellow attendees at conferences, participants during guest lectures and trainings, and via community forums.
- Books, journals, magazines, and other online publications.
- Experiences of one of the authors who works as an independent consultant in the area of design assessment and refactoring.

We want to emphasize that our goal is *not* to point out at any particular software organization and the smells in their design through our anecdotes, case studies, or examples. Rather, our sole goal is to educate software engineers about the potential problems in software design. Therefore, it should be noted that the details in the anecdotes and case studies reported in this book have been modified suitably so that confidential details such as the name of the organization, project, or product are not revealed.

This is a kind of book where you do not have to read from first page to last page—take a look at the table of contents and jump to the chapters that interest you.

WHERE CAN I FIND MORE INFORMATION?

You can get more details about the book on the Elsevier Store at http://www.store.elsevier.com./product.jsp?isbn=9780128013977. You can find more information, supplementary material, and resources at http://www.designsmells.com. For any suggestions and feedback, please contact us at designsmells@gmail.com.

WHY DID WE WRITE THIS BOOK?

Software design is an inherently complex activity and requires software engineers to have a thorough knowledge of design principles backed with years of experience and loads of skill. It requires careful thought and analysis and a deep understanding of the requirements. However, today, software engineers are expected to build really complex software within a short time frame in an environment where requirements are continuously changing. Needless to say, it is a huge challenge to maintain the quality of the design and overall software in such a context.

We, therefore, set out on a task to help software engineers improve the quality of their design and software. Our initial efforts were geared toward creating a method for assessing the design quality of industrial software (published as a paper [4]). We surveyed a number of software engineers and tried to understand the challenges they faced during design in their projects. We realized that while many of them possessed a decent theoretical overview of the key design principles, they lacked the knowledge of how to apply those principles in practice. This led to smells in their design.

Our survey also revealed a key insight—design smells could be leveraged to understand the mistakes that software engineers make while applying design principles. So, we embarked on a long journey to study and understand different kinds of smells that manifest in a piece of design. During this journey, we came across smells scattered across numerous sources including books, papers, theses, and tools and started documenting them. At the end of our journey, we had a huge collection of 530 smells!

We had hoped that our study would help us understand design smells better. It did, but now we were faced with the humongous task of making sense of this huge collection of smells. We, therefore, decided to focus only on structural design smells. We also decided to limit our focus to smells that are commonly found in real-world projects. Next, we set out on a mission to meaningfully classify this reduced collection of smells. We experimented with several classification schemes but were dissatisfied with all of them. In the midst of this struggle, it became clear to us that if we wanted to organize this collection so that we could share it in a beneficial manner with fellow architects, designers, and developers, our classification scheme should be linked to something fundamental, i.e., design principles. From this emerged the following insight:

> *When we view every smell as a violation of one or more underlying design principle(s), we get a deeper understanding of that smell; but perhaps more importantly, it also naturally directs us toward a potential refactoring approach for that smell.*

We built upon this illuminating insight and adopted Booch's fundamental design principles (abstraction, encapsulation, modularization, and hierarchy) as the basis for our classification framework. We categorized and aggregated the smells based on which of the four design principles they primarily violated, and created an initial

catalog of 27 design smells. We published this initial work as a paper [5] and were encouraged when some of the reviewers of our paper suggested that it would be a good idea to expand this work into a book. We also started delivering corporate training on the topic of design smells and observed that software practitioners found our smell catalog really useful. Buoyed by the positive feedback from both the reviewers of the paper and software practitioners, we decided to develop our initial work into this book.

Our objective, through this book, is to provide a framework for understanding how smells occur as a violation of design principles and how they can be refactored to manage technical debt. In our experience, the key to transforming into a better designer is to *use smells as the basis to understand how to apply design principles effectively in practice*. We hope that this core message reaches you through this book.

We have thoroughly enjoyed researching design smells and writing this book and hope you enjoy reading it!

The following extracts from Chapters 4 and 5 describe the "Leaky Encapsulation" and "Cyclically-Dependent Modularization" smells. For more great smells order your copy of the book at your favorite bookseller or store.elsevier.com/9780128013977

4.2 LEAKY ENCAPSULATION

This smell arises when an abstraction "exposes" or "leaks" implementation details through its public interface. Since implementation details are exposed through the interface, it not only is harder to change the implementation but also allows clients to directly access the internals of the object (leading to potential state corruption).

It should be noted that although an abstraction may not suffer from Deficient Encapsulation (Section 4.1) (i.e., the declared accessibility of its members are as desired), it is still possible that the methods in the pubic interface of the abstraction may leak implementation details (see examples in Section 4.2.3).

4.2.1 RATIONALE

For effective encapsulation, it is important to separate the interface of an abstraction (i.e., "what" aspect of the abstraction) from its implementation (i.e., "how" aspect of the abstraction). Furthermore, for applying the principle of hiding, implementation aspects of an abstraction should be hidden from the clients of the abstraction.

When implementation details of an abstraction are exposed through the public interface (i.e., there is a violation of the enabling technique "hide implementation details"):

- Changes made to the implementation may impact the client code.
- Exposed implementation details may allow clients to get handles to internal data structures through the public interface, thus allowing clients to corrupt the internal state of the abstraction accidentally or intentionally.

Since implementation aspects are not hidden, this smell violates the principle of encapsulation; furthermore, the abstraction "leaks" implementation details through its public interface. Hence, we term the smell Leaky Encapsulation.

4.2.2 POTENTIAL CAUSES

Lack of awareness of what should be "hidden"

It requires considerable experience and expertise to discern and separate the implementation and interface aspects of an abstraction from each other. Inexperienced designers often inadvertently leak implementation details through the public interface.

Viscosity

It requires considerable thought and effort to create "leak-proof" interfaces. However, in practice, due to project pressures and deadlines, often designers or developers resort to quick and dirty hacks while designing interfaces, which leads to this smell.

Use of fine-grained interface

Often, this smell occurs because fine-grained methods are directly provided in the public interface of a class. These fine-grained methods typically expose unnecessary implementation details to the clients of the class. A better approach is to have logical coarse-grained methods in the public interface that internally make use of fine-grained methods that are private to the implementation.

4.2.3 EXAMPLES

Example 1

Consider a feature-rich e-mail application similar to Microsoft® Outlook® or Mozilla Thunderbird®. One of the features that such an application supports is maintaining a list of to-do items. Assume that the content of each to-do item is maintained in a `TodoItem` class. The class `TodoList` maintains the list of `TodoItems`. Figure 4.6 outlines the members in `TodoList` class (distilled to a very simple form to maintain focus on the smell under discussion).

One of the methods in `TodoList` class is the public method `getListEntries()`, which returns the list of to-do items maintained by that object. The problem is with the return type of the method i.e., `LinkedList`. The method exposes the internal detail that the `TodoList` class uses a linked list as the mechanism for maintaining the list of `TodoItems`.

Presumably, the application may have mostly inserts and deletes, and hence the class designer may have chosen a `LinkedList` for implementation in `TodoList` class. In the future, if it is found that the application performs search operations more frequently

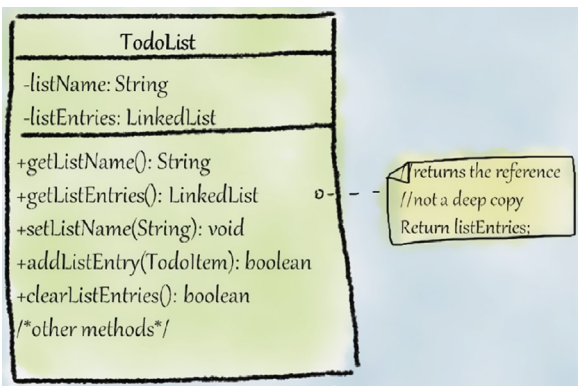


FIGURE 4.6

TodoList class exposing its internal use of linked list (Example 1).

than modifications to the list, it is better to change the implementation to use some other data structure such as an `ArrayList` or a `HashMap` (which provides faster lookup than a `LinkedList`). Furthermore, a new requirement tomorrow may require that at runtime the application should be able to switch to a specific implementation based on the context. However, since `getListEntries()` is a public method and returns `LinkedList`, changing the return type of this method may break the clients that are dependent on it. This suggests that it may be very difficult to support future modifications to the `TodoList` class.

Another serious problem with the `getListEntries()` method is that it returns a handle (i.e., reference) to the internal data structure. Using this handle, the clients can directly change the `listEntries` data structure bypassing the methods in `TodoList` such as `addListEntry()`.

To summarize, `TodoList` class "leaks" an implementation detail in its public interface, thereby binding the `TodoList` class to use a specific implementation.

ANECDOTE

A participant of the Smells Forum reported an example of this smell in an e-commerce application. In this application, the main logic involved processing of orders, which was abstracted in an `OrderList` class. The `OrderList` extended a custom tree data structure, since most use cases being tested by the team involved inserting and deleting orders.

When the application was first deployed, a customer filed a change request complaining that the application was not able to handle large number of orders and it "hanged." Profiling the application showed `OrderList` to be a "bottleneck class." The tree implementation internally used was not a self-balancing tree, and hence the tree often got skewed and looked more like a "list." To fix this problem, a developer quickly added a public method named `balance()` in the `OrderList` class that would balance the tree. He also ensured that the `balance()` method was invoked from the appropriate places in the application to address the skewing of the tree. When he tested the application, this solution appeared to solve the performance problem and so he closed the change request.

In a few months, more customers reported similar performance problems. Taking a relook at the class revealed that most of the time orders were being "looked-up" and not inserted or deleted. Hence the team now considered replacing the internal tree implementation to a hash-table implementation.

However, the `OrderList` class was tied to the "tree" implementation because it extended a custom `Tree` class. Furthermore, changing the implementation to use a more suitable data structure such as a hash table was difficult because the clients of the `OrderList` class used implementation-specific methods such as `balance()`! Although refactoring was successfully performed to use hash table instead of unbalanced tree in the `OrderList`, it involved making changes in many places within the application.

There are two key insights that are revealed when we reflect on this anecdote:

- Having the `OrderList` *extend* the custom tree instead of *using* the custom tree forced the `OrderList` to unnecessarily expose implementation details to the clients of the `OrderList` class. Due to the choice of using inheritance over delegation, the clients of the `OrderList` could not be shielded from the changes that arose when a different data structure, i.e., a hash table was used inside the `OrderList`.
- Adding `balance()` method in the public interface of the class is clearly a work-around. Such workarounds or quick-fix solutions appear to solve the problem, but they often backfire and cause more problems. Hence, one must invest a lot of thought in the design process.

Design smells such as Leaky Encapsulation can cause serious problems in applications, and we need to guard against such costly mistakes.

Example 2

Consider a class `CustomListBox`, which is a custom implementation of a ListBox UI control. One of the operations that you may perform on a `ListBox` object is to sort it. Figure 4.7 shows only the sort-related methods in `CustomListBox` class.

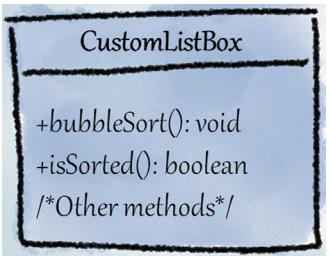


FIGURE 4.7

CustomListBox leaks the name of the sorting algorithm it uses (Example 2).

The problem with this class is that the name of the public method `bubbleSort()` exposes the implementation detail that the `CustomListBox` implementation uses "bubble sort" algorithm to sort the list. What if the sorting algorithm needs to be replaced with a different sorting algorithm, say, "quick sort" in the future? There are two possibilities:

- **Option 1**: Change the name of the algorithm to `quickSort()` and replace the algorithm accordingly. However, this change will break existing clients who are depending on the public interface of the class.
- **Option 2**: Replace only the implementation of the `bubbleSort()` method with the quick sort algorithm without changing the name of the existing method; i.e., the method name remains `bubbleSort()` but the method implements quick sort algorithm. The problem with this bad hack is that it will end up misleading the clients of this class that a "bubble sort" algorithm is being used because the name of the algorithm is mentioned in the method.

Thus both options are undesirable. In reflection, the root cause of the problem is that the name of the algorithm is "leaked" in the public interface of the class, thus leading to a Leaky Encapsulation smell.

4.2.4 SUGGESTED REFACTORING

The suggested refactoring for this smell is that the interface should be changed in such a way that the implementation aspects are not exposed via the interface. For example, if details of the internal algorithms are exposed through the public interface, then refactor the public interface in such a way that the algorithm details are not exposed via the interface.

It is also not proper to return a handle to internal data structures to the clients of a class, since the clients can directly change the internal state by making changes through the handle. There are a few ways to solve the problem:

- Perform deep copy and return that cloned object (changes to the cloned object do not affect the original object).
- Create an immutable object and return that cloned object (by definition, an immutable object cannot be changed).

If low-level fine-grained methods are provided in the public interface, consider making them private to the implementation. Instead, logical coarse-grained methods that make use of the fine-grained methods can be introduced in the public interface.

Suggested refactoring for Example 1

The suggested refactoring for the `ToDoList` example is to change the return type of the `getListEntries()` in such a way that it does not reveal the underlying implementation. Since `List` interface is the base type of classes such as `ArrayList` and `LinkedList`, one refactoring would be to use `List` as the return type of `getListEntries()` method. Or, a designer could choose to use `Collection` as the return type, which is quite general and does not reveal anything specific about the implementation.

However, it is also important to ensure that the clients of the `ToDoList` are not provided a handle to directly change its internal state. Therefore, consider returning a deep copy or a read-only copy of the internal list to the clients. At this point, let us take a step back and think about the clients of the `TodoList` class. Why would they be interested in `getListEntries()` method? If the end goal was to traverse the list and access each entry, then a better refactoring solution would be to return an `Iterator` instead. This kind of logical structured reasoning in practice will allow you to come up with a better and more effective design.

Suggested refactoring for Example 2

The suggested refactoring for the `CustomListBox` example is to replace the method name `bubbleSort()` with `sort()` (see Figure 4.8). With this, the `sort()` method can internally choose to implement any suitable sorting algorithm for sorting the

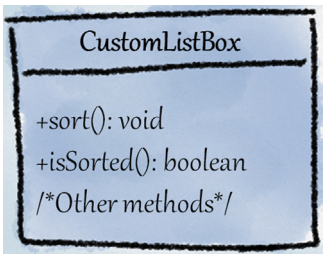


FIGURE 4.8

Suggested refactoring for CustomListBox (Example 2).

elements in the `CustomListBox`. For instance, the class `CustomListBox` can internally use Strategy pattern to employ the relevant algorithm at runtime.

Note that this class is part of the public API, so changing the method name will break the clients of the class. A work-around, to address this problem and the smell at the same time, is to introduce a new `sort()` method and deprecate the existing `bubbleSort()` method. Design is hard; API design is harder!

CASE STUDY

Consider the case of a software system in which a class implements operations on an image object. The operation to display an image completes in four steps that must be executed in a specific sequence, and the designer has created public methods corresponding to each of these steps: i.e., `load()`, `process()`, `validate()`, and `show()` in the `Image` class. Exposing these methods creates a constraint on using these four methods; specifically, the methods corresponding to these four steps—load, process, validate, and show—must be called in that specific sequence for the image object to be valid.

This constraint can create problems for the clients of the `Image` class. A novice developer may call `validate()` without calling `process()` on an image object. Each of the operations makes a certain assumption about the state of the image object, and if the methods are not called in the required sequence, then the operations will fail or leave the image object in an inconsistent state. Clearly, the `Image` class exposes "how" aspects of the abstraction as part of the public interface. It thus exhibits the Leaky Encapsulation smell.

How can this problem be addressed? One potential solution is to have a state flag to keep track of whether the operations are performed in a sequence. However, this is a really bad solution, since maintaining a state flag is cumbersome: if we maintain the state flag in the `Image` class, every operation must check for the proper transition and handle any exceptional condition.

At this juncture, we can take a step back and question why the four internal steps need to be exposed to the client who is concerned only with the display of the image. We can then see that the problem can be solved by exposing to the clients of the `Image` class only one method, say `display()`, which internally calls each of the four steps in sequence.

If we reflect upon this anecdote, we see that the underlying problem in this example is that the fine-grained steps were exposed as public methods. Hence, the key take-away here is that it is better to expose coarse-grained methods via the public interface rather than fine-grained methods that run the risk of exposing internal implementation details.

4.2.5 IMPACTED QUALITY ATTRIBUTES

- **Understandability**—One of the main objectives of the principle of encapsulation is to shield clients from the complexity of an abstraction by hiding its internal details. However, when internal implementation details are exposed in the case of a Leaky Encapsulation smell, the public interface of the abstraction may become more complex, affecting its understandability.
- **Changeability and Extensibility**—When an abstraction "exposes" or "leaks" implementation details through its public interface, the clients of the abstraction may depend directly upon its implementation details. This direct dependency makes it difficult to change or extend the design without breaking the client code.

- **Reusability**—Clients of the abstraction with Leaky Encapsulation smell may directly depend upon the implementation details of the abstraction. Hence, it is difficult to reuse the clients in a different context, since it will require stripping the client of these dependencies (for reference, see Command Pattern [54]).
- **Reliability**—When an abstraction "leaks" internal data structures, the integrity of the abstraction may be compromised, leading to runtime problems.

4.2.6 ALIASES

This smell is also known in literature as:

- Leaking implementation details in API [49]—This smell occurs when an API leaks implementation details that may result in confusing users or inhibiting freedom to change implementation.

4.2.7 PRACTICAL CONSIDERATIONS

Low-level classes

Consider an example of embedded software that processes and plays audio in a mobile device. In such software, the data structures that store metadata about the audio stream (e.g., sampling rate, mono/stereo) need to be directly exposed to the middleware client. In such cases, when public interface is designed purposefully in this way, clients should be warned that the improper use of those public methods might result in violating the integrity of the object. Also, it is important to ensure that such classes are not part of the higher-level API, such as the one that is meant for use by the mobile application software, so that runtime problems can be avoided.

5.3 CYCLICALLY-DEPENDENT MODULARIZATION

This smell arises when two or more abstractions depend on each other directly or indirectly (creating a tight coupling between the abstractions).

5.3.1 RATIONALE

A cyclic dependency is formed when two or more abstractions have direct or indirect dependencies on each other. Cyclic dependencies between abstractions violate the Acyclic Dependencies Principle (ADP)[79] and Ordering Principle [80]. In the presence of cyclic dependencies, the abstractions that are cyclically-dependent may need to be understood, changed, used, tested, or reused together. Further, in case of cyclic dependencies, changes in one class (say A) may lead to changes in other classes in the cycle (say B). However, because of the cyclic nature, changes in B can have ripple effects on the class where the change originated (i.e., A). Large and indirect cyclic dependencies are usually difficult to detect in complex software systems and are a common source of subtle bugs. Since this smell is a result of not adhering to the enabling technique, "create acyclic dependencies," we name this smell Cyclically dependent Modularization.

A special form of cyclic dependency is sometimes exhibited within an inheritance hierarchy. A subtype has a dependency on its supertype because of the inheritance relationship. However, when the supertype also depends on the subtype (for instance, by having an explicit reference to the subtype), it results in a cyclic dependency. We refer to this special form of cyclic dependency as Cyclic Hierarchy smell, and discuss it in detail in Section 6.10.

UNDERSTANDING CYCLES

To better understand the impact of cyclic dependencies, let us first understand some terms related to cycles. A *dependency diagram* is a directed graph that shows the dependency relationship among abstractions. In a dependency diagram, if you start from one abstractions and reach the same abstraction by following one or more path(s) formed by the dependency edges, the abstractions in the followed path form a *cycle*. A *tangle* consists of more than one cycle. When abstractions are tightly coupled by a large number of direct or indirect cyclic dependencies, they form a *tangled design*, and the dependency graph looks unpleasantly complex.

Figure 5.10 shows a tangle between six abstractions in `java.util` package. In this graph, you can see cycles of various lengths. (It should be noted that the graph also shows Cyclic Hierarchy smell.)

In this design, any change to an abstraction involved in this dependency chain has the potential to affect other abstractions that depend on it, causing ripple effects or cascade of changes. A designer must, therefore, strive for designs that do not consist of tangles.

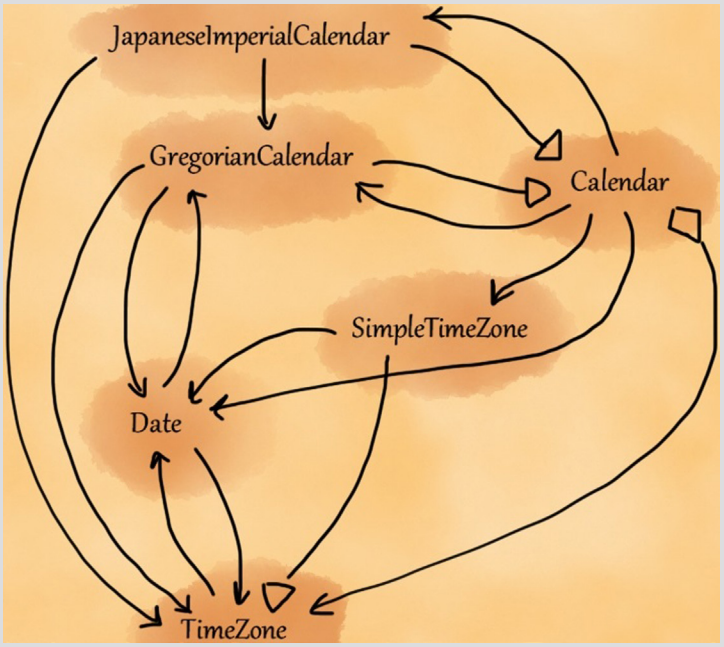


FIGURE 5.10

Cycles between six abstractions in java.util package.

5.3.2 POTENTIAL CAUSES

Improper responsibility realization

Often, when some of the members of an abstraction are wrongly misplaced in another abstraction, the members may refer to each other, resulting in a cyclic dependency between the abstractions.

Passing a self reference

A method invocation from one abstraction to another often involves data transfer. If instead of explicitly passing only the required data, the abstraction passes its own reference (for instance, via "this") to the method of another abstraction, a cyclic dependency is created.

Implementing call-back functionality

Circular dependencies are often unnecessarily introduced between two classes while implementing call-back[2] functionality. This is because inexperienced developers

[2]A call-back method is one that is passed as an argument to another method, and is invoked at a later point in time (for instance, when an event occurs).

may not be familiar with good solutions such as design patterns [54] which can help break the dependency cycle between the concerned classes.

Hard-to-visualize indirect dependencies

In complex software systems, designers usually find it difficult to mentally visualize dependency relationships between abstractions. As a result, designers may inadvertently end up creating cyclic dependencies between abstractions.

5.3.3 EXAMPLES

Example 1

Let us consider a storage application from the Cloud ecosystem. This application allows a client to upload its data to the Cloud where it can be archived. Since security and privacy are important concerns in the Cloud context, the client-side application encrypts the data before uploading it to the Cloud. Assume that this application consists of a class named `SecureDocument` that uses a `DESEncryption`[3] object to encrypt a document. To encrypt the contents of the document, the `SecureDocument` calls a method `encrypt()` provided in `DESEncryption` class and passes the "this" pointer as its argument. For instance, the call may look like this:

```
SecureDocument encryptedDocument = desEncryption.encrypt(this);
```

The method `encrypt` in `DESEncryption` is declared as follows:

```
SecureDocument encrypt(SecureDocument docToEncrypt)
```

Using the "this" pointer within the `encrypt` method, the `DESEncryption` object fetches the contents of the document, encrypts it, and returns an encrypted document object. Thus, `SecureDocument` and `DESEncryption` know about each other, leading to the Cyclically-dependent Modularization smell between them, as shown in Figure 5.11.

Example 2

Consider the case of a medical application that supports encryption of scanned images before storing them on a drive. This application consists of a `SecurityManager` class that fetches an encrypted image from an `Image` class (Figure 5.12). The `Image` class, in turn, uses an `Encryption` class to encrypt its contents; during this process, the `Image` class passes the "this" pointer to the `encrypt()` method within the `Encryption` class. When invoked, the `encrypt()` method within the `Encryption`

[3]DES stands for Data Encryption Standard; it is a well-known symmetric key algorithm for encrypting data.

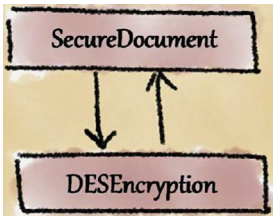


FIGURE 5.11

Cyclic dependency between SecureDocument and DESEncryption (Example 1).

FIGURE 5.12

Cyclic dependency between Image and Encryption classes (Example 2).

class fetches the `Image` contents, and returns the contents after encryption. Here, the `Image` and `Encryption` classes are dependent on each other, hence this design exhibits the Cyclically-dependent Modularization smell.

Example 3

Consider an order-processing module in an e-commerce application. In this application, assume that you have two classes named `Order` and `OrderImpl` that provide support for order processing (Figure 5.13). The `Order` class maintains information about an order and the associated information about the ordered items. The method `getAmount()` in `Order` class uses `computeAmount()` method of `OrderImpl` class. In

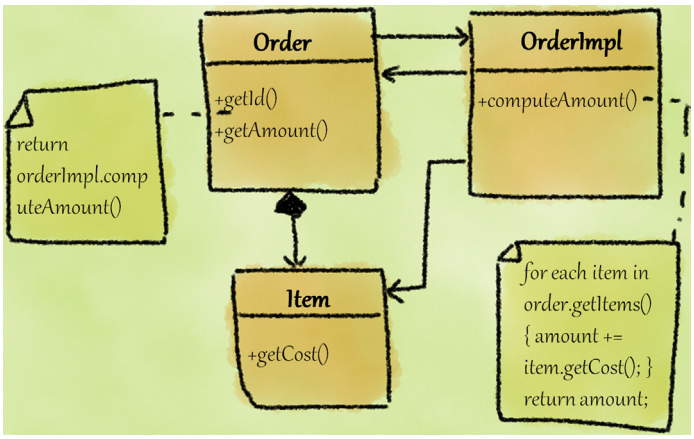


FIGURE 5.13

Cyclic dependency between Order and OrderImpl classes (Example 3).

turn, the `computeAmount()` method extracts all the items associated with the `Order` object and computes the sum of costs of all the items that are a part of the order. In effect, classes `Order` and `OrderImpl` depend on each other, hence the design has the Cyclically-dependent Modularization smell.

Example 4

Assume that an order-processing application has an `Order` class encapsulating information about an order such as `name`, `id`, and `amount`. The `getAmount()` method of the `Order` class uses `computeAmount()` method of `TaxCalculator` class. The `computeAmount()` method fetches all the items associated with the `Order` object and computes a summation of each item cost. In addition, it calls `calculateTax()` method, adds the computed tax with the running amount, and returns it. In this case, classes `Order` and `TaxCalculator` depend on each other, as shown in Figure 5.14. Clearly, the design fragment shows the Cyclically-dependent Modularization smell.

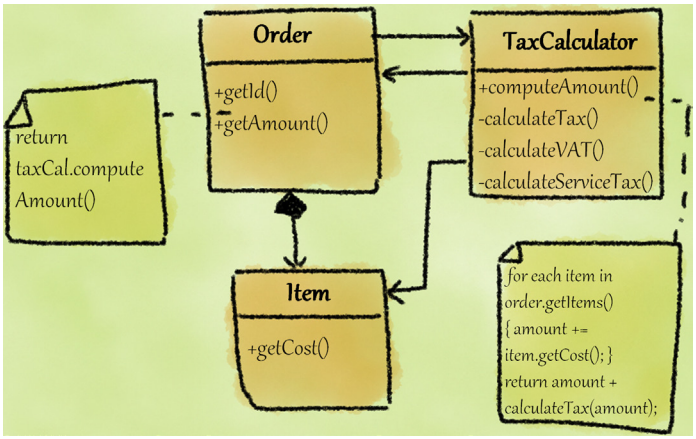


FIGURE 5.14

Cyclic dependency between Order and TaxCalculator classes (Example 4).

5.3.4 SUGGESTED REFACTORING

The refactoring for this smell involves breaking the dependency cycle. There are many strategies to do this; some important ones are:

- **Option 1:** Introduce an interface for one of the abstractions involved in the cycle.
- **Option 2:** In case one of the dependencies is unnecessary and can be safely removed, then remove that dependency. For instance, apply "move method" (and "move field") refactoring to move the code that introduces cyclic dependency to one of the participating abstractions.
- **Option 3:** Move the code that introduces cyclic dependency to an altogether different abstraction.

- **Option 4:** In case the abstractions involved in the cycle represent a semantically single object, merge the abstractions into a single abstraction.

As an illustration, consider the direct cyclic dependency between class A and class B (Figure 5.15). Figure 5.16, Figure 5.17, Figure 5.18, and Figure 5.19 respectively show the four options for refactoring to remove the cycle.

Suggested refactoring for Example 1

For the storage application from the Cloud ecosystem, a suggested refactoring is to introduce an interface `IEncryption` and have the class `DESEncryption` implement the interface. The `SecureDocument` class now depends on `IEncryption` interface instead of depending on the concrete `DESEncryption` class. This results in a design that removes the cyclic dependency. Interfaces are less likely to change than a concrete type, thus the resultant design is more stable than the original design with cyclic dependency. For instance, a

FIGURE 5.15

Cyclic dependency between classes A and B.

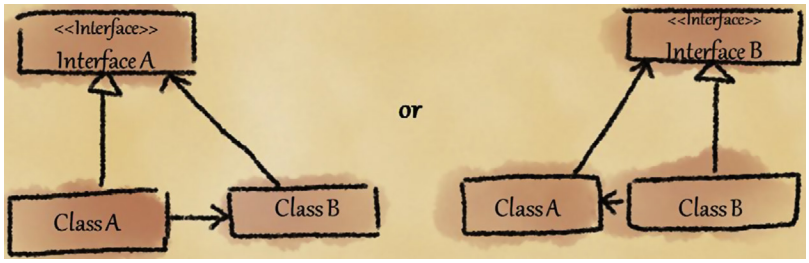


FIGURE 5.16

Breaking a cyclic dependency by introducing an interface (Option 1).

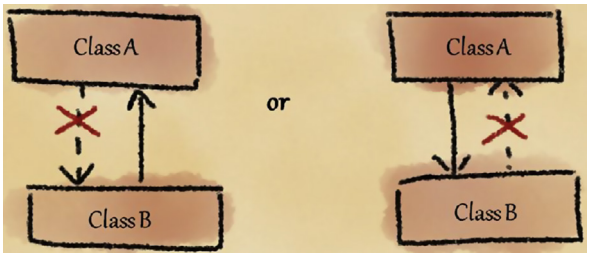


FIGURE 5.17

Breaking a cyclic dependency by removing a dependency (Option 2).

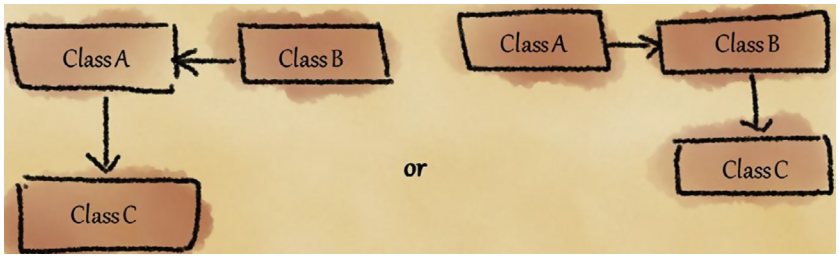


FIGURE 5.18

Breaking a cyclic dependency by introducing another abstraction (Option 3).

FIGURE 5.19

Breaking a cyclic dependency by merging the abstractions (Option 4).

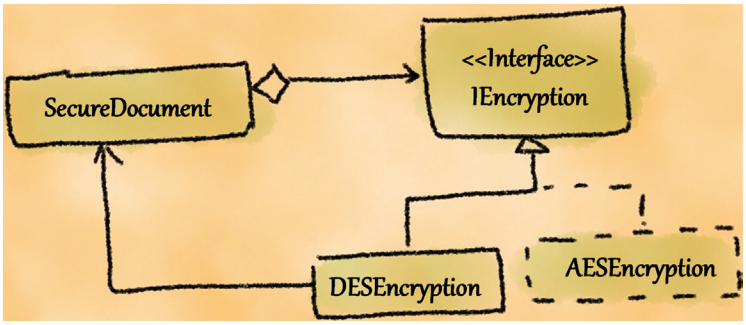


FIGURE 5.20

Suggested refactoring for cyclic dependency between SecureDocument and DESEncryption (Example 1).

new class such as `AESEncryption` (that supports the AES[4] encryption algorithm) can be added to the design without affecting the `SecureDocument` class (Figure 5.20).

Suggested refactoring for Example 2

For the healthcare application that requires image encryption, a suggested refactoring is to shift the responsibility of encrypting an image object from `Image` class to `SecurityManager` class to break the cycle. In the refactored design, `SecurityManager` class depends on `Image` and `Encryption` classes. `SecurityManger` class directly invokes image encryption on `Encryption` class by providing the

[4]Advanced Encryption Standard.

`Image` object. The `Encryption` class, in turn, uses the `Image` object reference provided by the `SecurityManager` to fetch the content of the image object, encrypt the content, and return the encrypted content to `SecurityManager`. Here, by removing the dependency from `Image` class to `Encryption` class, the cyclic dependency is broken (Figure 5.21).

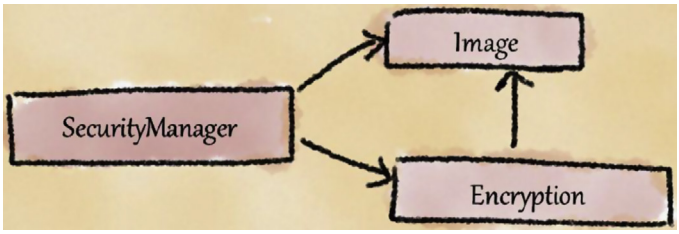


FIGURE 5.21

Suggested refactoring for cyclic dependency between Image and Encryption (Example 2).

Suggested refactoring for Example 3

In the case of the `Order` example, classes `Order` and `OrderImpl` are tightly coupled to each other, and they semantically represent an order abstraction. Since the `OrderImpl` class has only one method, this class could be merged into the `Order` class. When we apply this refactoring, the cyclic dependency disappears, since there is only one class (Figure 5.22).

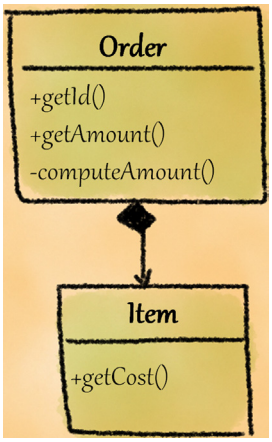


FIGURE 5.22

Suggested refactoring for cyclic dependency in order-processing module (Example 3).

Suggested refactoring for Example 4

To refactor the `Order` example with the `TaxCalculator` class, we can employ "move method" refactoring. The `computeAmount()` originally belonging to `TaxCalculator` class can be moved to `Order` class. By moving this method, we eliminate the dependency from `TaxCalculator` to `Order` class (Figure 5.23).

ANECDOTE

One of the authors was working as a software design consultant for a startup company. The product that the company was developing was originally designed by an experienced architect, but he had recently quit the company. Most developers in the team were fresh engineers, a company strategy intended to keep costs low until they could get further funding.

With lack of experienced designers or architects, development proceeded without any focus on architecture or design quality. The management prided itself on its pragmatic view and pushed the development team to meet functional requirements and get the product to the market on time.

In this process, the author noticed that numerous compromises were made in the design. For instance, the original design was a layered architecture with strict separation of concerns. However, with no architect or designer to oversee the development, developers started introducing business logic in UI classes such as `Buttons` and `Panels`. Worse, low-level utility classes such as `GraphUtilities` also directly referred to GUI classes. These aspects clearly indicate violation of the layering style.

To demonstrate the extent of the problems to management, the author ran dependency analysis tools and found innumerable dependency cycles across layers. He also used visualization tools to demonstrate how the whole codebase was excessively tangled. It was evident that touching code in any class could potentially break the working software. Due to the extent of the problem and the impending release, management could not take immediate remedial steps.

While the product was well received by the customers when it was launched, the design was so brittle that even small changes would break the working software. Upon the recommendation of the author, the management recruited an experienced designer as the architect. All further development was frozen, and considerable refactoring was performed to clean up the design. After a few months of refactoring, the design quality significantly improved to an extent that development tasks could commence. However, because the technical debt had to be repaid through extensive refactoring, the second release was delayed considerably from its originally scheduled date.

Key take-aways from this experience are:

- It is important to have a designated architect who actively ensures that architectural erosion or design decay does not happen.
- If the technical debt is not repaid through periodic refactoring, it will eventually stall the project.

5.3.5 IMPACTED QUALITY ATTRIBUTES

- **Understandability:** Since all the abstractions that are cyclically-dependent can be understood only together, it impacts the understandability of the design.
- **Changeability** and **Extensibility:** Making changes to an abstraction that is part of a cycle can cause ripple effects across classes in the dependency chain (including the original abstraction). This makes it difficult to understand, analyze, and implement new features or changes to any abstraction that is part of a

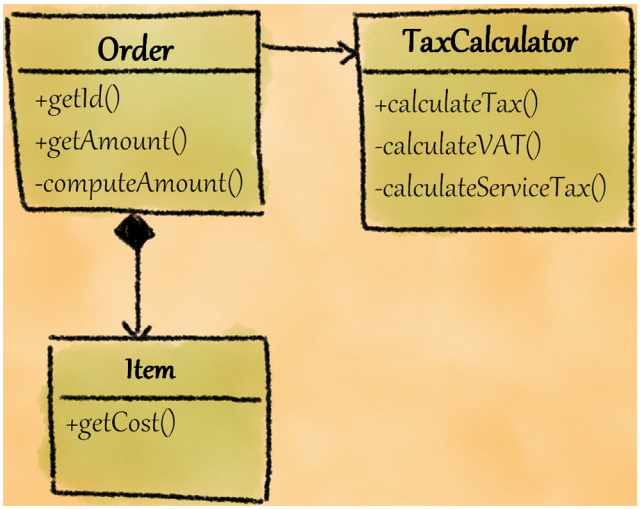


FIGURE 5.23

Suggested refactoring for cyclic dependency between Order and TaxCalculator (Example 4).

dependency cycle. Hence, this smell impacts the changeability and extensibility of the design.

- **Reusability:** The abstractions in a dependency cycle can only be reused together. Hence, this smell impacts reusability of the design.
- **Testability:** Since it is difficult to independently test the abstractions participating in the cycle, this smell impacts testability.
- **Reliability:** In a dependency chain, changes to any abstraction can potentially manifest as runtime problems across other abstractions. For instance, consider the case where a value is represented as a double in a class and that the class is part of a long cycle. If the value is changed to float type in the class, the impact of this change on other classes may not be evident at compile-time; however, it is possible that it manifests as a floating-point error at runtime in other classes.

5.3.6 ALIASES

This smell is also known in literature as:

- Dependency cycles [72, 73, 74]: This smell occurs when a class has circular references.
- Cyclic dependencies [71, 75, 76]: This smell occurs when classes are tightly coupled and mutually dependent.
- Cycles [9]: This smell occurs when one of the (directly or indirectly) used classes is the class itself.

- Bidirectional relation [57]: This smell occurs when two-way dependencies between methods of two classes are present.
- Cyclic class relationships [70]: This smell occurs when classes have improper or questionable relationships to other classes, such as codependence.

5.3.7 PRACTICAL CONSIDERATIONS

Unit cycles between conceptually related abstractions

Cycles of size one (i.e., cycles that consist of exactly two abstractions) are known as *unit cycles*. Often, unit cycles are formed between conceptually related pairs. For instance, consider the classes `Matcher` and `Pattern` (both part of `java.util.regex` package) that are cyclically-dependent on each other (see Figure 5.24). These two classes are almost always understood, used, or reused together. In real-world projects, it is common to find such unit cycles. However, since the unit cycle is small, it is easier to understand, analyze, and implement changes within the two abstractions that are part of the cycle.

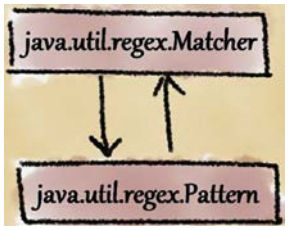


FIGURE 5.24

Cyclic dependency between Matcher and Pattern.

However, large cycles and tangles (which are commonly observed in large real-world software systems) make it considerably difficult to maintain the software. It is, therefore, important for software designers in real-world projects to focus on reducing cyclic dependencies between abstractions.

SOFTWARE DEVELOPMENT AND ENGINEERING

Refactoring for Software Design Smells

Managing Technical Debt

Girish Suryanarayana, Ganesh Samarthyam, Tushar Sharma

"... a delightful, engaging, actionable read... you have in your hand a veritable field guide of smells... one of the more interesting and complex expositions of software smells you will ever find... The concept of technical debt is central to understanding the forces that weigh upon systems, for it often explains where and how and why a system is stressed. What delights me about this present book is its focus on technical debt and refactoring as the actionable means to attend to it."
–From the foreword by Grady Booch, IBM Fellow and Chief Scientist for Software Engineering, IBM Research

"Evolving software inevitably accumulates technical debt, making maintenance increasingly painful and expensive. The authors, based on their extensive experience, categorize the major design problems (smells) that come up in software, and lucidly explain how these can be solved with appropriate refactoring."
–Diomidis Spinellis, Author of *"Code Reading"* and *"Code Quality"*, Addison-Wesley Professional

"...the book I would have loved to write... "Refactoring for Software Design Smells" is an excellent book. It is another milestone that professionals will use... I'm sure that you will learn a lot from it and that you will enjoy it."
–From the foreword by Stéphane Ducasse, Co-author of *Object-Oriented Reengineering Patterns,* Morgan Kaufmann

Key Features

- **Contains a comprehensive catalog of 25 structural design smells (organized around four fundamental design principles) that contribute to technical debt in software projects**
- **Presents a unique naming scheme for smells that helps understand the cause of a smell as well as points toward its potential refactoring**
- **Includes illustrative examples that showcase the poor design practices underlying a smell and the problems that result**
- **Covers pragmatic techniques for refactoring design smells to manage technical debt and to create and maintain high-quality software in practice**
- **Presents insightful anecdotes and case studies drawn from the trenches of real-world projects**

Girish Suryanarayana, Senior Research Scientist at Research and Technology Centre, Siemens Technology and Services Pvt. Ltd., Bangalore, India

Ganesh Samarthyam, Independent Consultant and Corporate Trainer based in Bangalore, India

Tushar Sharma, Technical Expert at Research and Technology Centre, Siemens Technology and Services Pvt. Ltd., Bangalore, India

MORGAN KAUFMANN PUBLISHERS

AN IMPRINT OF ELSEVIER

mkp.com

ISBN 978-0-12-801397-7