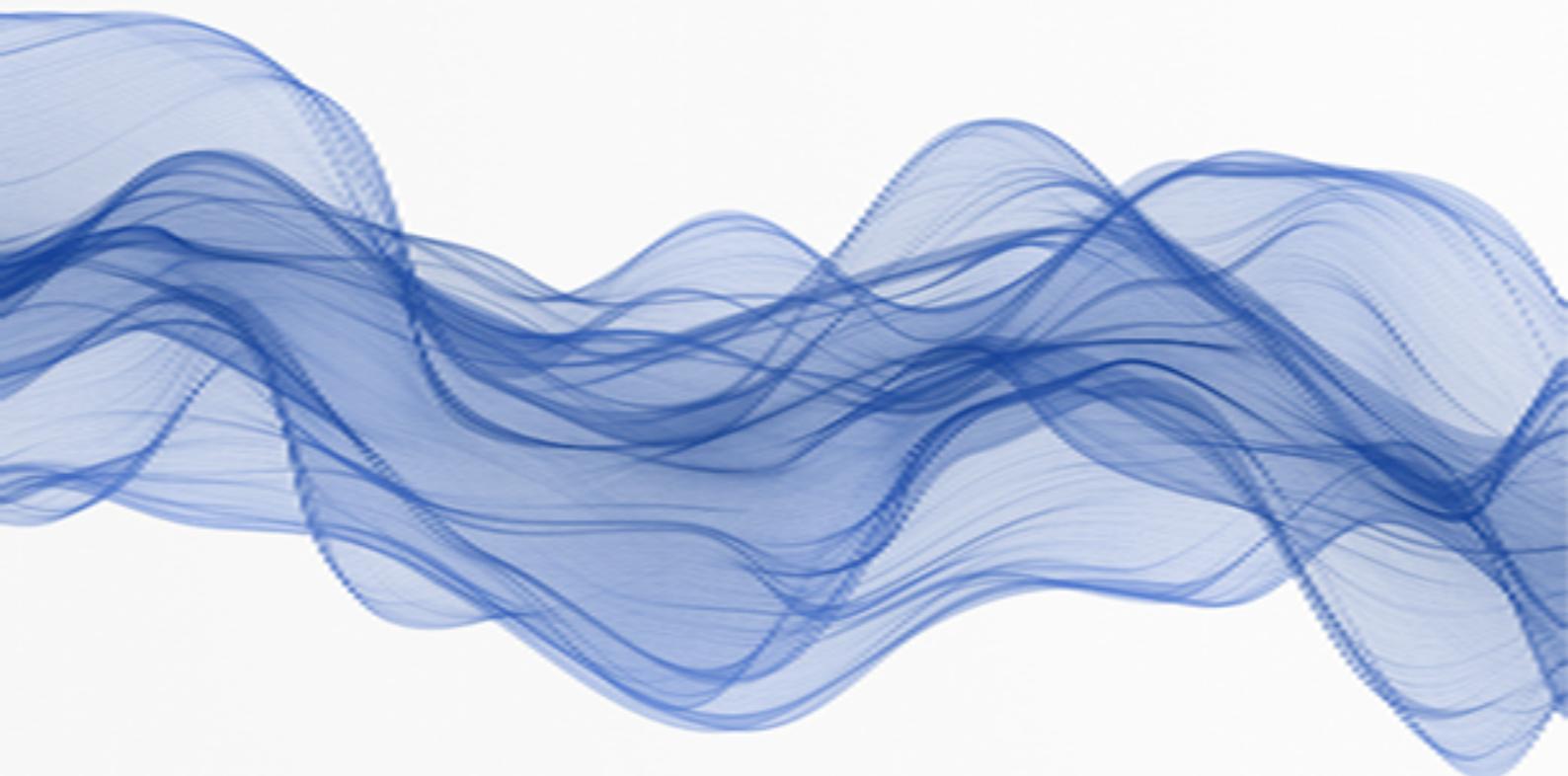


Clean Code Fundamentals



A **Hands-on Guide** to
Understand the Fundamentals of
Software Craftsmanship and
Clean Code in Java

Martin Hock

Clean Code Fundamentals

Hands-on Guide to Understand the Fundamentals of Software Craftsmanship and Clean Code in Java

Martin Hock

This book is for sale at <http://leanpub.com/clean-code-fundamentals>

This version was published on 2021-06-20



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2019 - 2021 Martin Hock

Tweet This Book!

Please help Martin Hock by spreading the word about this book on [Twitter](#)!

The suggested hashtag for this book is [#CleanCodeFundamentals](#).

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

[#CleanCodeFundamentals](#)

Contents

Preface	i
What is the goal of this book?	i
Which topics can you expect?	i
Who should read this book?	ii
What you must learn about Software Development yourself?	ii
What about the code examples and typographic conventions?	iii
Which literature is this book based on?	iii
Giving feedback?	iv
1. Introduction to Software Craftsmanship and Clean Code	1
1.1 A Passion for Software Development	1
1.2 Manifesto for Software Craftsmanship	3
1.3 Clean Code Developer	3
1.4 Boy Scout Rule	6
1.5 Broken Windows Theory	7
1.6 Cargo Cult Programming	8
1.7 Knowledge - Expertise	8
2. Basics of Software Design	9
2.1 Software Design Pyramid	9
2.2 Basic concepts of OOD	10
2.3 Goals of Software Design	13
2.4 Symptoms of <i>bad</i> design	14
2.5 Criteria for <i>good</i> design	14
2.6 Information Hiding	15
2.7 Cohesion	17
2.8 Coupling	18
2.9 Cohesion - Coupling	19
2.10 Big Ball of Mud	20
2.11 Architecture Principles	21
2.12 Cognitive Psychology and Architectural Principles	22
2.13 Layered Architecture	23
2.13.1 Use of Layered Architecture	23
2.13.2 Violated Layered Architecture	25
2.13.3 Horizontal Layering	26
2.13.4 Feature Based Layering - Single package	27
2.13.5 Feature Based Layering - Slices before layers	29

CONTENTS

2.13.6	Feature Based Layering – Hexagonal Architecture	30
2.13.7	The Java Module System	31
2.14	Architecture Documentation	32
2.15	Testing the Architecture and Design	34
2.16	Software Engineering Values	35
2.17	Team Charter	36
3.	Java Best Practices	37
3.1	Communicate through code	37
3.1.1	Use Java code conventions and avoid misinformation	37
3.1.2	Choose an expressive name and avoid mental mapping	38
3.1.3	Make differences clear with meaningful variable names	38
3.1.4	Use pronounceable names	38
3.1.5	Do not hurt the readers	39
3.1.6	Don't add redundant context	40
3.1.7	Don't add words without additional meaning	41
3.1.8	Don't use <i>and</i> or <i>or</i> in method names	42
3.1.9	Respect the order within classes	43
3.1.10	Group by line break	45
3.1.11	Prefer self-explanatory code instead of comments	46
3.1.12	Refactor step by step	48
3.2	Bad comments	49
3.2.1	Redundant comments	49
3.2.2	Misleading comments	49
3.2.3	Mandatory comments	50
3.2.4	Diary comments	50
3.2.5	Gossip	51
3.2.6	Position identifier	51
3.2.7	Write-ups and incidental remarks	52
3.2.8	Don't leave commented out code in your codebase	52
3.2.9	Rules for commenting	52
3.3	Classes and objects	53
3.3.1	Classes	53
3.3.2	Functions	53
3.3.3	Variables	54
3.4	Shapes of code	55
3.4.1	Spikes	55
3.4.2	Paragraphs	55
3.4.3	Paragraphs with headers	56
3.4.4	Suspicious comments	56
3.4.5	Intensive use of an object	56
3.5	Avoid instantiation for utility classes	57
3.6	Use immutable objects	58
3.7	Prefer records for immutability	62
3.8	Provide immutable decorators for sensitive mutable classes	64
3.9	Avoid constant interfaces	66

CONTENTS

3.10	Avoid global constant classes	67
3.11	Favour composition over inheritance	69
3.12	Use the <code>@Override</code> Annotation	72
3.13	Use the <code>@FunctionalInterface</code> Annotation	73
3.14	Prefer returning Null-Objects over <code>null</code>	74
3.15	Avoid <code>null</code> as method parameter	77
3.16	Prefer enhanced loops over for loops	79
3.17	Code against interfaces not implementations	80
3.18	Use existing exceptions	80
3.19	Validate method parameter	81
3.20	Prevent <code>NullPointerException</code> for <code>String</code> comparison	82
3.21	Safely cast <code>long</code> to <code>int</code>	83
3.22	Convert integers to floating point for floating-point math operations	84
3.23	Use Generics in favour of raw types	85
3.24	Prefer enums over <code>int</code> constants	86
3.25	Be aware of the contract between <code>equals</code> and <code>hashCode</code>	88
3.26	Use text blocks for multi-line strings	91
3.27	Use always braces for the body of all statements	93
3.28	Pre calculate the length in loops	94
3.29	Avoid slow instantiation of <code>String</code>	94
3.30	Use <code>StringBuilder</code> for concatenation	95
3.31	Reduce lookups in collection containers	96
3.32	Instantiate wrapper objects with <code>valueOf</code>	97
3.33	Use <code>entrySet</code> for iterating	98
3.34	Use <code>isEmpty()</code> for <code>String</code> length	99
3.35	Reduce the number of casts	99
4.	Software Quality Assurance	100
4.1	Test Pyramid	100
4.2	Test Classification	101
4.3	Test-driven Development (TDD)	102
4.4	Unit testing with JUnit 5	104
4.4.1	Unit Tests	104
4.4.2	JUnit 5	104
4.4.3	First unit test	106
4.4.4	Assertions	107
4.4.4.1	<code>assertEquals()</code> / <code>assertArrayEquals()</code>	107
4.4.4.2	<code>assertSame()</code> / <code>assertNotSame()</code>	107
4.4.4.3	<code>assertTrue()</code> / <code>assertFalse()</code> / <code>assertAll()</code>	108
4.4.4.4	<code>assertNull()</code> / <code>assertNotNull()</code>	108
4.4.4.5	<code>assertThrows()</code>	108
4.4.4.6	<code>assertTimeout()</code>	109
4.4.4.7	<code>fail()</code>	109
4.4.5	Annotations	110
4.4.5.1	<code>@Test</code>	111
4.4.5.2	<code>@BeforeEach</code> / <code>@AfterEach</code>	112

CONTENTS

4.4.5.3	@BeforeAll / @AfterAll	113
4.4.5.4	@Disabled	114
4.4.5.5	@DisplayName	115
4.4.5.6	@Tag	115
4.4.5.7	@Timeout	116
4.4.6	Assumptions	117
4.4.6.1	assumeFalse()	117
4.4.6.2	assumeTrue()	117
4.4.6.3	assumingThat()	117
4.4.7	Parameterized Tests	118
4.4.7.1	@ValueSource	118
4.4.7.2	@MethodSource	119
4.4.7.3	@CsvSource	120
4.4.7.4	@CsvFileSource	120
4.4.7.5	@ArgumentsSource	121
4.5	More on Unit Tests	122
4.5.1	Heuristics	122
4.5.2	Naming of test methods	125
4.5.3	Object Mother	126
4.5.4	Test Data Builder	127
4.5.5	F.I.R.S.T	129
4.6	Mocking with Mockito	130
4.6.1	Types of Test Double	130
4.6.2	Activation	131
4.6.3	Annotations	132
4.6.3.1	@Mock	132
4.6.3.2	@Spy	133
4.6.3.3	@Captor	133
4.7	Code Coverage	134
4.8	Static Code Analysis	136
4.9	Continuous Integration	137
4.9.1	Differences between CI, CD and CD	138
4.9.2	CI Workflow	140
4.9.3	Preconditions	141
4.9.4	Advantages and Disadvantages	142
4.9.5	Best Practices	142
5.	Design Principles	143
5.1	Goal of Design Principles	143
5.2	Overview of Design Principles	143
5.3	SOLID Principles	145
5.3.1	Single Responsibility Principle	145
5.3.1.1	Example: Modem	145
5.3.1.2	Example: Book	148
5.3.2	Open Closed Principle	150
5.3.2.1	Example: LoanRequestHandler	150

CONTENTS

5.3.2.2	Example: Shape	153
5.3.2.3	Example: HumanResourceDepartment	155
5.3.2.4	Example: Calculator	157
5.3.2.5	Example: FileParser	159
5.3.3	Liskov Substitution Principle	161
5.3.3.1	Example: Rectangle	161
5.3.3.2	Example: Coupon	164
5.3.4	Interface Segregation Principle	168
5.3.4.1	Example: MultiFunctionDevice	168
5.3.4.2	Example: TechEmployee	171
5.3.5	Dependency Inversion Principle	174
5.3.5.1	Example: UserService	174
5.3.5.2	Example: Logger	175
5.4	Packaging Principles - Cohesion	177
5.4.1	Release Reuse Equivalency Principle	177
5.4.2	Common Closure Principle	177
5.4.3	Common Reuse Principle	178
5.5	Packaging Principles - Coupling	178
5.5.1	Acyclic Dependencies Principle	178
5.5.1.1	Example: Cyclic dependency	179
5.5.2	Stable Dependencies Principle	182
5.5.3	Stable Abstractions Principles	184
5.6	Further Design Principles	187
5.6.1	Speaking Code Principle	187
5.6.2	Keep It Simple (and) Stupid!	187
5.6.3	Don't Repeat Yourself / Once and Only Once	187
5.6.4	You Ain't Gonna Need It!	188
5.6.5	Separation Of Concerns	188
6.	Design Patterns of the Gang of Four	189
6.1	Creational	192
6.1.1	Singleton	192
6.1.1.1	Example: Lazy loading	192
6.1.1.2	Example: Eager loading	193
6.1.1.3	Example: Enum singleton	193
6.1.2	Builder	194
6.1.2.1	Example: MealBuilder	195
6.1.2.2	Example: PizzaBuilder	197
6.1.2.3	Example: Email	199
6.1.2.4	Example: ImmutablePerson	201
6.1.3	Factory Method	204
6.1.3.1	Example: Logger	204
6.1.3.2	Example: Department	206
6.1.4	Abstract Factory	209
6.1.4.1	Example: Car	210
6.1.5	Prototype	215

CONTENTS

6.1.5.1	Example: Person - Shallow copy	215
6.1.5.2	Example: Person - Deep copy	218
6.1.5.3	Example: Person - Copy constructor / factory	220
6.2	Structural	223
6.2.1	Facade	223
6.2.1.1	Example: Travel	224
6.2.2	Decorator	225
6.2.2.1	Example: Message	226
6.2.2.2	Example: Window	227
6.2.3	Adapter	230
6.2.3.1	Example: Sorter	231
6.2.3.2	Example: TextFormatter	232
6.2.4	Composite	234
6.2.4.1	Example: Graphic	235
6.2.4.2	Example: Organization Chart	237
6.2.5	Bridge	239
6.2.5.1	Example: Message	239
6.2.5.2	Example: Television	242
6.2.6	Flyweight	245
6.2.6.1	Example: Font	246
6.2.6.2	Example: City	248
6.2.7	Proxy	251
6.2.7.1	Example: Spaceship	251
6.2.7.2	Example: ImageViewer	253
6.3	Behavioural	255
6.3.1	State	255
6.3.1.1	Example: MP3Player	255
6.3.1.2	Example: Door	258
6.3.2	Template Method	261
6.3.2.1	Example: Compiler	261
6.3.2.2	Example: Callbackable	262
6.3.3	Strategy	266
6.3.3.1	Example: Compression	266
6.3.3.2	Example: LogFormatter	268
6.3.4	Observer	271
6.3.4.1	Example: DataStore	272
6.3.4.2	Example: Influencer	273
6.3.5	Chain of Responsibility	275
6.3.5.1	Example: Purchase	275
6.3.5.2	Example: Authentication	278
6.3.6	Command	281
6.3.6.1	Example: FileSystem	281
6.3.6.2	Example: Television	284
6.3.7	Interpreter	287
6.3.8	Iterator	288
6.3.9	Mediator	289

CONTENTS

6.3.10	Memento	290
6.3.11	Visitor	291
6.3.11.1	Example: <code>Fridge</code>	291
6.3.11.2	Example: <code>Figures</code>	294

Bad code is like a joke, if you have to explain it, it is bad.

Preface

This is a *Forever Edition*. That means that this book will see periodic updates.

If you purchased this book, thank you very much! Writing a book like this takes some hours of effort. Please treat your book as your own personal copy and do not redistribute the book without authorization.

If you find this book useful, I would greatly appreciate you purchasing a copy. By doing so, you'll be letting me know that books like this are useful to you.

What is the goal of this book?

Readers of this book will acquire in-depth knowledge and skills for the analysis, assessment and improvement of software quality. You will be able to apply principles, patterns, techniques and tools needed to write clean code.

Understanding important concepts of software quality in Java projects:

- Best Practices
- Functional and technological scope
- Development and programming models
- Tools

Learning Objectives:

- Acquisition of knowledge and skills to analyze, assess and improve software quality
- Learning principles, patterns, techniques and tools
- Deeping technology knowledge

Which topics can you expect?

Software testing is not limited to a specific phase of a project. Already during the coding phase or the build process, critical and difficult to find software defects can be detected in the source code. The necessary procedures and tools are presented in this book.

Among other things, the following topics will be covered:

- Overview of the basics of software quality
- Software metrics, metric application in practice
- Structured design, cohesion and coupling
- Overview of Principles, Best Practices and Code Smells

- Compliance and verification of Java code conventions
- Static software testing, especially review techniques and static program analysis
- Ensure software quality with tools such as SonarQube, PMD, SpotBugs, Checkstyle, ArchUnit and Dependency-Track
- Software tests with JUnit and Mockito
- Checking the test code coverage
- CI/CD
- Design Principles
- Design Patterns (GoF)

Who should read this book?

I wrote the book, because I want to help developers getting experience in clean code or improve it. Get them a feeling what it is about to be a Software Craftsman. It is about Professionalism, Pragmatism and Pride!

It's a mindset, where software developers choose to be responsible for their own careers, learning new tools and techniques and constantly getting better. The journey to a Software Craftsmanship is a long one.

That's why I wanted to share some techniques and practises for improving code quality and motivate developers.

What you must learn about Software Development yourself?

- **Developers are always wrong** - only the degree to which different developers are wrong in their endless architecture discussions differs!
- **If something can break, it will break** - make sure that a project is fool proof!
- **All code is bad** - there are only gradual gradations of the defectiveness of a piece of software!
- **There is always a bug somewhere** - you just must search long enough!
- **The most important thing is the customer** - and they don't give a damn about the technologies and processes used!
- **Project development on paper does not work** - problems are only identified during the development process!
- **Less is more** - if something is not necessary, just leave it out!
- **Only 20% of our work consists of coding** - the rest is designing, debugging, testing, meetings, discussing etc.
- **The customer never knows what he wants** - everyone has only a vague idea of the desired result!
- **Someone has done it before** - don't reinvent the wheel!

What about the code examples and typographic conventions?

All the code examples in this book are in Java. I don't want to put much boilerplate code within the code examples, so I just leave it out.

Class names, data types, method and function names and code are not included in proportional font.

Which literature is this book based on?

Schneider, Kurt: *Abenteuer Software Qualität – Grundlagen und Verfahren für Qualitätssicherung und Qualitätsmanagement*, dpunkt.verlag, 2007

Robert, Martin: *Clean Code – A Handbook of Agile Software Craftsmanship*, Prentice Hall, 2009

Lilienthal, Carola: *Sustainable Software Architecture. Analyze and Reduce Technical Debt*, Dpunkt Verlag, 2019

Bloch, Joshua: *Effective Java – Third Edition*, Addison-Wesley, 2017

Roock, Stefan: *Refactorings in grossen Softwareprojekten*, Dpunkt Verlag, 2004

Gamma, Erich: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional, 1994

Robert C. Martin: *Agile Software Development: Principles, Patterns and Practices*, Prentice Hall, 2003

Bugayenko, Yegor: *Elegant Objects Volume 1*, CreateSpace Independent Publishing Platform, 2016

Bugayenko, Yegor: *Elegant Objects Volume 2*, CreateSpace Independent Publishing Platform, 2017

Harrer, Simon: *Java by Comparison: Become a Java Craftsman in 70 Examples*, O'Reilly UK Ltd., 2018

Robert, Martin: *Clean Architecture: A Craftman's Guide to Software Structure and Design*, Prentice Hall, 2017

Robert, Martin: *The Clean Coder: A Code of Conduct for Professional Programmers*, Prentice Hall, 2011

David, Thomas: *Pragmatic Programmer special 2nd*, Addison-Wesley Professional, 2019

Kaczanowski, Tomek: *Practical Unit Testing with TestNg*, 2012

Meszaros, Gerard: *xUnit Test Patterns*, Addison-Wesley, 2007

Long, Fred: *Java Coding Guidelines: 75 Recommendations for Reliable and Secure Programs*, Addison-Wesley Professional, 2013

Sandro, Mancuso: *The Software Craftsman: Professionalism, Pragmatism, Pride*, Pearson, 2014

Freeman, Steve: *Growing Object-Oriented Software, Guided by Tests*, Addison-Wesley, 2009

Giving feedback?

If you have anything to say about this book, I'd love to receive your feedback.

Found a typo? Discovered a code bug? Have a content suggestion? Let me know and be a part of this book by providing feedback.

For contributions, please always tell me the chapter name, heading reference, and a brief snippet of text as a reference.

You can use the Leanpub forum of this book to post errata and ask questions about this book.

Since this book is self-published, I will create as many releases as I wish.

Thank you for your time to help making this book better!



You can provide feedback under this e-mail:

clean-code-fundamentals@gmail.com

or via Twitter:

#CleanCodeFundamentals

1. Introduction to Software Craftsmanship and Clean Code

1.1 A Passion for Software Development

Software Craftsmanship is a long journey to mastery. It's a mindset where software developers choose to be responsible for their own careers, constantly learning new tools and techniques and constantly bettering themselves.

—Sandro Mancuso, The Software Craftsman

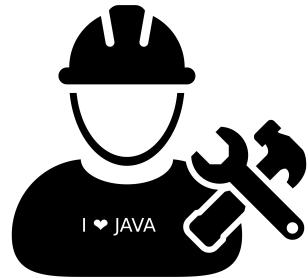
The basis of the Software Craftsmanship movement is the publication of *Clean Code* by Robert C. Martin. The author is a co-founder of the Software Craftsmanship movement. There are many initiatives like the *Manifesto for Software Craftsmanship* or the *Clean Code Developer* movement which raised the bar regarding software development - luckily.

Software Craftsmanship focuses on the adoption of good technical practices in the development process. It is obvious, that everyone likes Software Craftsmanship. Everybody wants to have a clean code base, but it's difficult to achieve. There are several steps that should also be done in the right way. The practices defined under the Software Craftsmanship movement are the ones that will help you to achieve this.

Software Craftsmanship should be the goal you are aiming for. It is not only the maturity of your project, it is the quality of your software and your code base as well. As Benjamin Franklin said: *The bitterness of poor quality remains long after the sweetness of low price is forgotten.*

There are developers who are enthusiastic about their work and have already acquired these values to provide quality and kindness to the code. They have the necessary skills to realise products according to their values. Unfortunately, many others know little or not at all that there is another way to develop software. Even today, many development practices that help to create well-crafted software are simply not taught in schools.

If you work in an environment where these practices simply are not applied or there is no room for well-crafted software, ask yourself the question if you can change this. Be a craftsman, spread the professionalism of software development out to the team with the hope that individuals will follow you. Adopt the culture where the values of Software Craftsmanship have the appropriate value. Do not rank other developers, maybe they had never the chance to work in this way or had never the luck having a good mentor. When you have success with this approach, fine. If not, you may be in the wrong company for you!



Software Craftsmanship

Practices that helps you strengthen the passion as a software developer:

Understand the importance of *good enough*.

A working good enough solution is usually better than a non-existent perfect solution. Learn to balance constraints. Don't get bogged down in the war of beliefs about technologies and the right solution, be pragmatic and effective in what you do. Know where and when to compromise and when to say *enough*.

If you get stuck, don't hesitate to ask for help.

Don't spend your time endlessly trying to solve a problem that can easily be solved by someone else. There are people in your team that can help to solve your problem much faster and give you valuable tips. Know when to keep trying and when to stop and ask for help.

If you are not able to communicate your ideas and connect with others, you will not get far.

Learn how to communicate effectively at every level with the people around you. Get involved with other ideas, listen to them and expand your horizons with other points of view. This will help you to question your own views and, in the end, to communicate them to others.

You will be criticised. Learn how to deal with it.

Understand the value of criticism and learn how to make the most of it. These are opportunities for you to grow and should be valuable feedback provided for you.

If you want to make a difference, you need to be effective in prioritising.

Not everything you can do is always worth doing. Understand how to identify high-impact work and dismiss the rest. Understand the basics and concepts and deepen them. Focus on a few things that you are very interested in while keeping track of other things. Nowadays frameworks come and go, it is more important to understand the concepts behind them. You can apply these to all frameworks coming.

Learn how you be effective in learning.

Find out the best way for you to learn new things over time. Explaining things to others and sharing your knowledge is a very effective way to deepen your own knowledge and learn new things.

Be selective with your valuable time.

You cannot do all by yourself. If you learn how to delegate effectively, you can multiply your time for topics which provide more value to you. Be realistic with your goals, you can't know everything.

Share all you learn with everyone else.

Not only you will help others around you to become better, but you will also be strengthening your own understanding and growth your knowledge. When knowledge grows in a team and exchange takes place, everyone wins.

Learn how to motivate others.

If you know how to light a fire and keep it alive, everyone will want to stay around you. Allow your team members to work in their own way and give them the freedom they need to work

creatively and effectively. Set a good example and find out what your colleagues need to be motivated.

Learn to be proactive.

Don't wait for problems to become problems. Take care of things in a proactive way.

1.2 Manifesto for Software Craftsmanship

The famous [Manifesto for Software Craftsmanship](#)¹ describe professional software developers, and itself reads like an addition to the [Agile Manifesto](#)²:

As aspiring Software Craftsmen, we are raising the bar of professional software development by practicing it and helping others learn the craft. Through this work we have come to value:

- Not only working software, but also **well-crafted software**
- Not only responding to change, but also **steadily adding value**
- Not only individuals and interactions, but also a **community of professionals**
- Not only customer collaboration, but also **productive partnerships**

That is, in pursuit of the items on the left we have found the items on the right to be indispensable.

1.3 Clean Code Developer

The [Clean Code Developer \(CCD\)](#)³ is an initiative for more professionalism in software development. CCD define a collection of principles and practices. These are divided into different degrees. However, it also includes a collection of specific values, such as **evolvability**, **correctness**, **production efficiency** and **continuous improvement**.

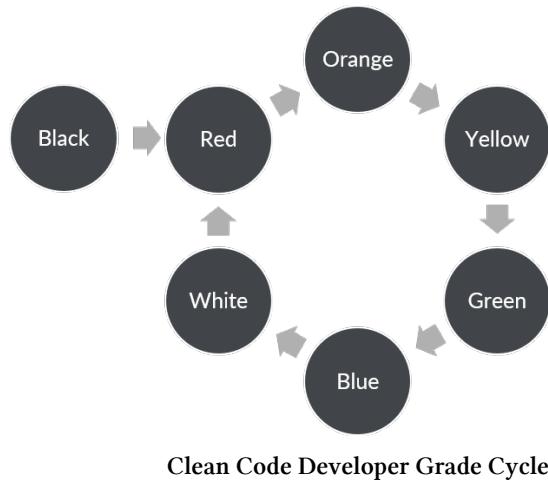
Through constant iteration of the different degrees, these principles and practices are becoming part of the way we act. This creates an awareness of quality within the development process. Those who practice daily will be able to intuitively retrieve this knowledge

Below you can see an overview of the degrees of the CCD system, which the iterative cycle goes through. The degrees build on each other in terms of difficulty and benefits. A developer who has not yet started with CCD owns the black grade. If someone not yet started with CCD own the black grade. A black grade only indicates the initial interest in CCD.

¹<http://manifesto.softwarecraftsmanship.org/>

²<http://agilemanifesto.org/>

³<https://clean-code-developer.com/>



Clean Code Developer Grades

Grade	Principles	Practices
Red	Don't Repeat Yourself (DRY) Keep it simple, stupid (KISS) Beware of Optimizations Favour composition over inheritance	Using a Version Control System Refactoring Patterns Rename and Extract Method Boy Scout Rule Root Cause Analysis Daily Reflection
Orange	One level of abstraction Single Responsibility Principle (SRP) Separation of Concerns (SoC) Source Code Convention	Issue Tracking Automatic Integration Tests Read, Read, Read Reviews
Yellow	Information hiding Principle of least astonishment (POLA) Liskov Substitution Principle (LSP) Interface Segregation Principle (ISP) Dependency Inversion Principle (DIP)	Attend Conferences Automatic Unit Tests Mockups Code Coverage Analysis Complex Refactoring
Green	Open Closed Principle (OCP) Tell, don't ask Law of Demeter (LoD)	Continuous Integration Static Code Analysis Inversion of Control Container Share your Experience
Blue	Implementation matches Design You Ain't Gonna Need It (YAGNI) Separation of Design and Implementation	Continuous Deployment Component Orientation Iterative Development TDD
White	All principles flow together and includes all other grades	All practices flow together and includes all other grades

A developer finishes with the white grade when he is aware of all the CCD values. In this grade he follows a constant repetition of the cycle, which serves the purpose of refining the use of the

aspects of the individual grades. This reflects our job as developers, in which we constantly keep our knowledge up to date and refine our tooling and practises. Only experienced and advanced software developers will be able to work in white grade.

A more visualized overview of all principles, practices and values can be found here.⁴⁵

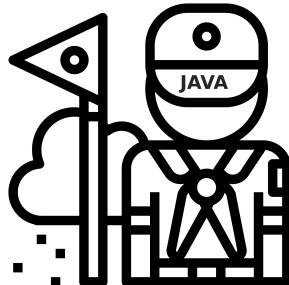
⁴<https://unclassified.software/de/topics/clean-code>

⁵<http://michael.hoennig.de/download/CCD-Poster.pdf>

1.4 Boy Scout Rule

Always leave the campground cleaner than you found it.

—Robert C. Martin⁶



Boy Scout Rule

Boy Scouts have a rule regarding camping that they should leave the campground cleaner than they found it. By ensuring this rule, they can guarantee that they do not cause new problems, at least as far as the cleanliness of the area is concerned.

What if we apply this rule to our code?

The Boy Scout Rule suggests that you simply try to make sure that each time you commit, you leave the code better than when you found it. This means that every time you make a change, you make an improvement to the code base. Maybe only slightly. Improve a variable name, clean-up an incomplete documentation, extract a large method to multiple smaller ones or add a missing test.

Keeping code clean is a constant challenge, and developers and software teams must decide if, when and how they want to keep their code clean. Resolving these [technical debts](#)⁷ through refactoring is necessary to keep the code in a state where it remains economically viable to extend and to maintain it.

When does it make sense to spend time working on improving the code?

Over time, the quality of the source code on which a system is based tends to deteriorate, leading to increasing technical debt. Some teams take the approach of discontinuing all value-adding work and try to clean up the code base. This usually results in some refactoring sprints or even a complete rewrite. Neither of these is obviously a solution from a business perspective. Feature development and bug-fixing must continue, and refactoring should not take up most of the time. According to the Boy Scout Rule, teams can improve the quality of their code over time while continuing to deliver value to their customers and stakeholders. In this way, continuous improvement allows you to move in a corridor where technical debt does not dominate your system. Do not let your system exceed a point where the application becomes unmaintainable. Thus “*Try to leave the world a little better than you found it*”⁸.

⁶<http://cleancoder.com/>

⁷Ward Cunningham: *The WyCash Portfolio Management System*: Experience Report, OOPSLA '92, 1992

⁸Robert Baden-Powell, the father of the Boy Scouts, and the original of this expression. https://en.wikipedia.org/wiki/Robert_Baden-Powell,_1st_Baron_Baden-Powell

1.5 Broken Windows Theory

The importance of fixing the small problems in your code, the “broken windows”, so they don’t grow into large problems.

—The Pragmatic Programmer⁹



The broken windows theory is a criminological theory that states that visible signs of crime, anti-social behavior, and civil disorder create an urban environment that encourages further crime and disorder, including serious crimes. The theory suggests that policing methods that target minor crimes, such as vandalism, loitering, public drinking, jaywalking and fare evasion, help to create an atmosphere of order and lawfulness, thereby preventing more serious crimes. —[Wikipedia](#)¹⁰



Broken Windows Theory

The Broken Window Theory can be used as a metaphor for software development and technical debt on a project.

Every time you delay a fix for a known problem, you create debt. You might know that something is broken, but you don’t fix it now, you’re in debt. If you have some of these and you are aware of them, that is fine. You make a notice and remove them later. But just like with real debt, it doesn’t take much to get to a point where you can never pay it back. It is the point where you have so many problems that you can never go back and solve them.

You do not want to let technical debt get out of hand. Stop the small problems before they grow into bigger ones. Don’t allow broken windows in your project, but fix them. If you can’t fix them now, create a technical debt ticket to make sure everyone knows they’re broken. A technical debt log helps to provide an overview within a team.

As soon as something is broken and cannot be repaired, it starts to spread throughout the team. The team or new team members get used to these Broken Windows and are not afraid to add new ones. This leads to even worse code and the loss of respect for how you handle the code. Take care of technical debts and solve problems no matter how small they are. Don’t let the code become rotten. The better the code, the less other team members will be tempted to check in bad code.

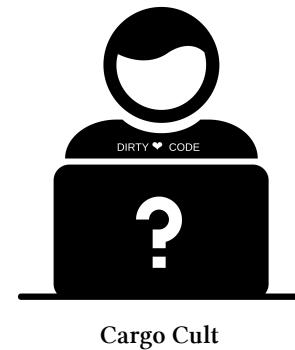
⁹<https://pragprog.com/titles/tpp20/the-pragmatic-programmer-20th-anniversary-edition/>

¹⁰https://en.wikipedia.org/wiki/Broken_windows_theory

1.6 Cargo Cult Programming

The [Cargo Cult Programming](#)¹¹ is a style of computer programming characterized by the ritual of code or program structures that serve **no real purpose**. The term cargo cult programmer may apply, when an **unskilled or novice computer programmer** copies some program code from one place to another with little or **no understanding** of how it works or whether it is required in its new position.

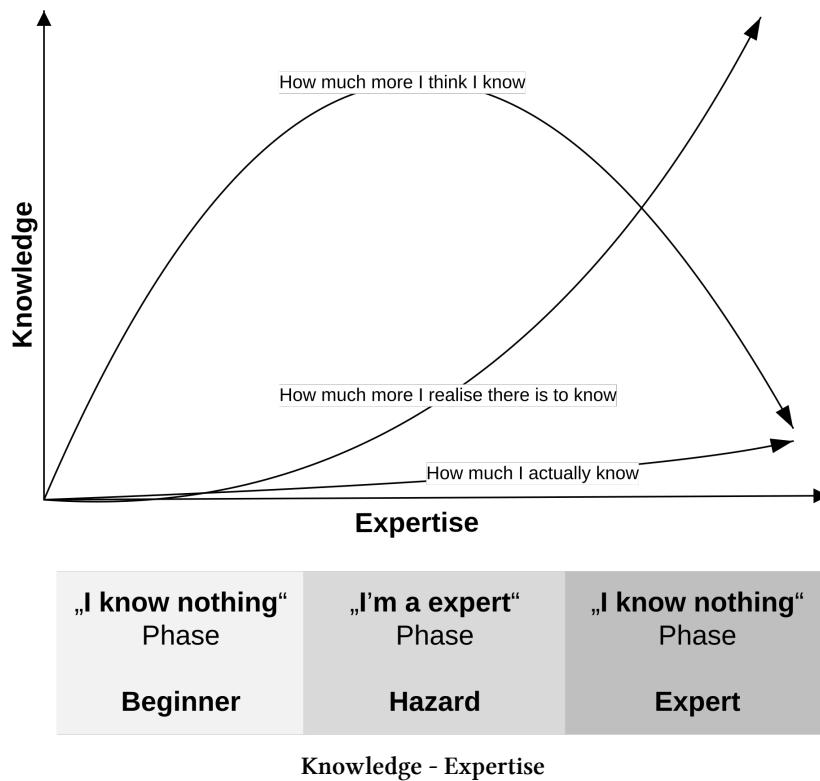
It can also refer to the results of applying a design pattern or coding style blindly **without understanding** the reasons behind that design principle.



Cargo Cult

1.7 Knowledge - Expertise

Simon Wardley published this graph as a [joke](#)¹² which it's not based on any research. Nevertheless, you can recognize yourself as a developer in this graphic. The more you study a programming language or technology, the more you think you actually know about it. But at some point, you have to admit to yourself that you basically know nothing at all. But we shouldn't despair, we should remain curious and should constantly learn.

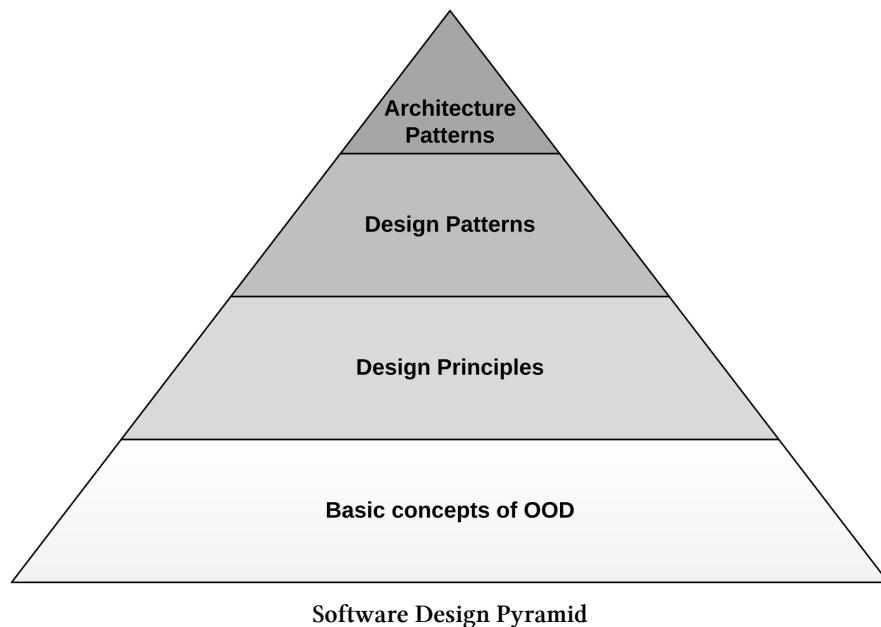


¹¹https://archive.org/details/professionalsof00mcco_0

¹²<https://threadreaderapp.com/thread/1045567546034987008.html>

2. Basics of Software Design

2.1 Software Design Pyramid



Basic concepts of OOD

The basic concepts of OOD are the prerequisite for this book and will be discussed only briefly in the following chapter. However, they are the basis for understanding the design principles and design patterns in the later chapters. The language specification and features of Java and the implementation of polymorphism, encapsulation, abstraction and inheritance help to understand and implement them.

Design Principles

The Design Principles are based on the concepts of the OOD. The best known are the SOLID principles of Robert C. Martin. SOLID is an acronym for five Design Principles intended to make software designs more understandable, flexible, and maintainable. These are explained in detail in this book. Besides these, there are many others, which are also described and explained in this book.

Design Patterns

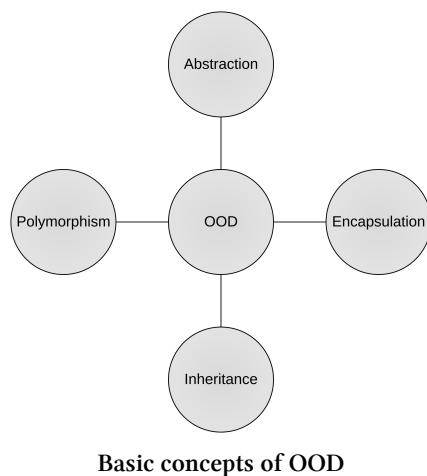
The Design Patterns go one step further and offer solution patterns for recurring problems. Probably the best known are the GoF Design Patterns, which are covered in this book with many examples. Besides the GoF Patterns, there are many more which are also worth to study.

Architecture Patterns

The Architectural Patterns are the top of the design pyramid. Architectural Patterns are methods of proven good design structures. More specifically, an Architectural Pattern is a set of design decisions that is repeated in practice, has well-defined characteristics which can be reused. There are many Architecture Patterns like microkernel, micro-services, layered architecture, event-based and many more.

2.2 Basic concepts of OOD

Object-oriented programming has four basic concepts: **abstraction**, **encapsulation**, **inheritance** and **polymorphism**. Although these concepts may seem complex, understanding how they work will help you to understand the basics of a software design.



Abstraction

Abstraction implies that the client only interacts with specific attributes and methods of an object. Abstraction uses simplified high-level access to a complex object.

- Hiding of complexity by ignoring irrelevant details
- Solves the problem at design level

Encapsulation

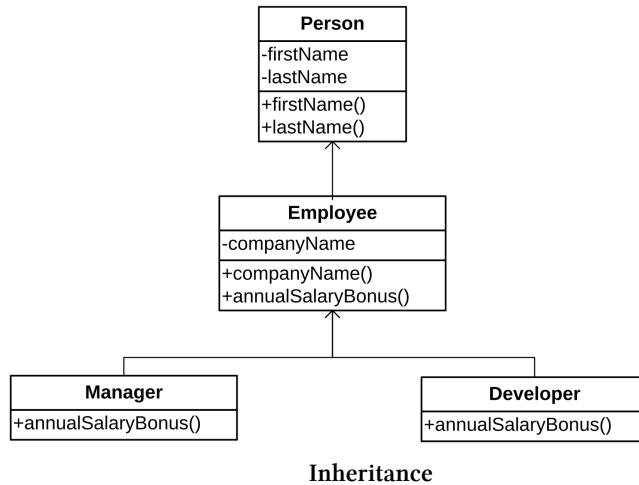
Encapsulation is the mechanism for hiding the data implementation by restricting access to classes, methods and variables. An example would be achieving this, by keeping instance variables private and making access methods more visible. To reduce couplings between software components, encapsulation mechanisms are essential.

- Information and data hiding
- Solves the problem at implementation level

Inheritance

Inheritance allows classes to inherit characteristics of other classes. In practice, parent classes extend attributes and behaviours to subclasses.

- Extension and specialisation
- Inheritance supports reusability

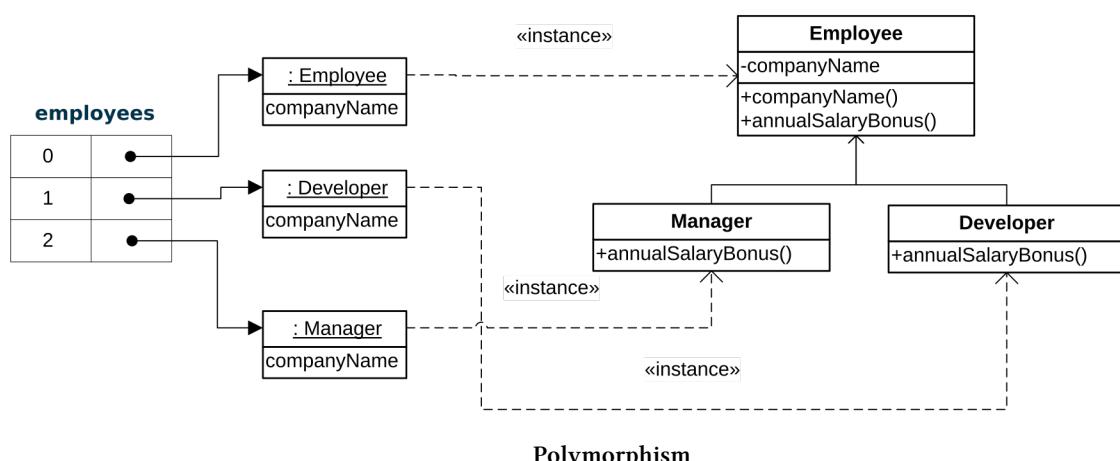


Inheritance

Polymorphism

Polymorphism means one name but many forms. It allows designing objects that share behaviours. Using inheritance, objects can overwrite parent behaviour with specific behaviour. Polymorphism allows the same method to perform different behaviour in a static and dynamic way. The static one is achieved by method overloading and the dynamic by method overriding. Thus, polymorphism is closely connected with inheritance. We can write code that works on the superclass, and it works with any subclass type as well.

- Substitutability
- Same interface, different behaviour



Polymorphism

Polymorphism

```
1 public class Client {
2
3     public static void main(String[] args) {
4         List<Employee> employees = List.of(
5             new Employee("Employee", "Max", "Company"),
6             new Developer("Developer", "Max", "Company"),
7             new Manager("Manager", "Max", "Company"));
8
9         employees.forEach(e -> System.out.printf(assembleMessage(e)));
10    }
11
12    private static String assembleMessage(Employee employee) {
13        return String.format("%s with an annual salary bonus of %g!\n",
14            employee.getClass().getName(),
15            employee.annualSalaryBonus());
16    }
17 }
```

Output

```
1 Employee with and an annual salary bonus of 1000,00!
2 Developer with and an annual salary bonus of 3000,00!
3 Manager with and an annual salary bonus of 5000,00!
```

In this example the compiler does not know what type the reference attribute `employee` is. They do not know which operation is called on `annualSalaryBonus()`. The method of the superclass `Employee` or the redefined `annualSalaryBonus()` of the subclass `Developer` OR `Manager`.

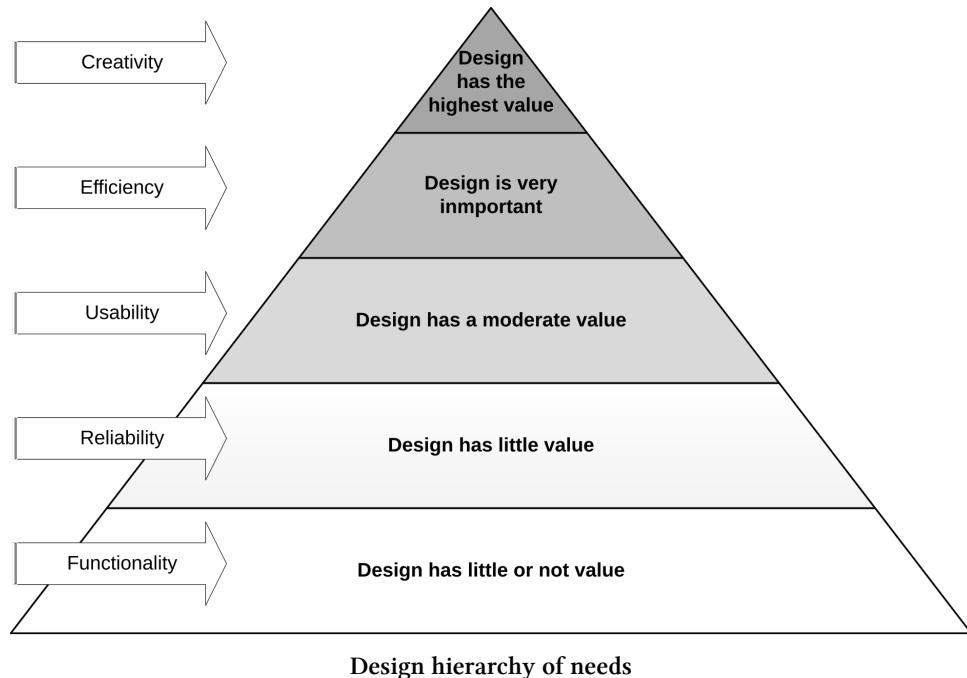
Furthermore, the operation `assembleMessage()` does not need to be modified if additional subclasses are added to the `Employee` superclass.

2.3 Goals of Software Design

Software design is invisible: software that meets all functional requirements can still be poorly designed.

- **Design is good when**
 - it breaks down the complexity of the software into manageable and simple problems.
 - small interfaces were defined.
 - the components are decoupled.
 - the components have clearly defined responsibilities.
 - the software is maintainable.
 - the software can be easily changed and extended.
 - the software is stable.
 - Bugs can be fixed quickly.
 - the software is reusable in other software projects.
 - the code is understandable.

The figure below shows the value that software design can have in a project.



Unfortunately, software design has not always a high priority in a project. This can be due to various reasons, a lack of technical experience or simply the fact that it is considered to be unimportant. From my point of view, it is a fatal mistake if a solid foundation is not created at the beginning of a project. It is almost like trying to build a house. You cannot build the house without a good foundation, and you start building from there.

2.4 Symptoms of *bad design*

- **Never-touch-running-code Syndrome**
 - Developers are afraid to change code.
 - Many workarounds, code is developed around it.
 - Changes have unknown and undetected side effects.
- **Small change in requirements leads to big changes in code**
- **Reuse through code duplication (Copy-Paste)**
 - Developers chase after the places that need to be changed.
 - The more code the more difficult it becomes to keep track of duplicates.
 - Errors have to be patched several times at different places.
- **Cyclic relations between artifacts**
 - Artifacts that are cyclically coupled cannot be tested individually.
 - Artifacts that are used in different cycles often play several roles, which makes them difficult to understand.
 - Artifacts that are used in different cycles cannot be exchanged easily.

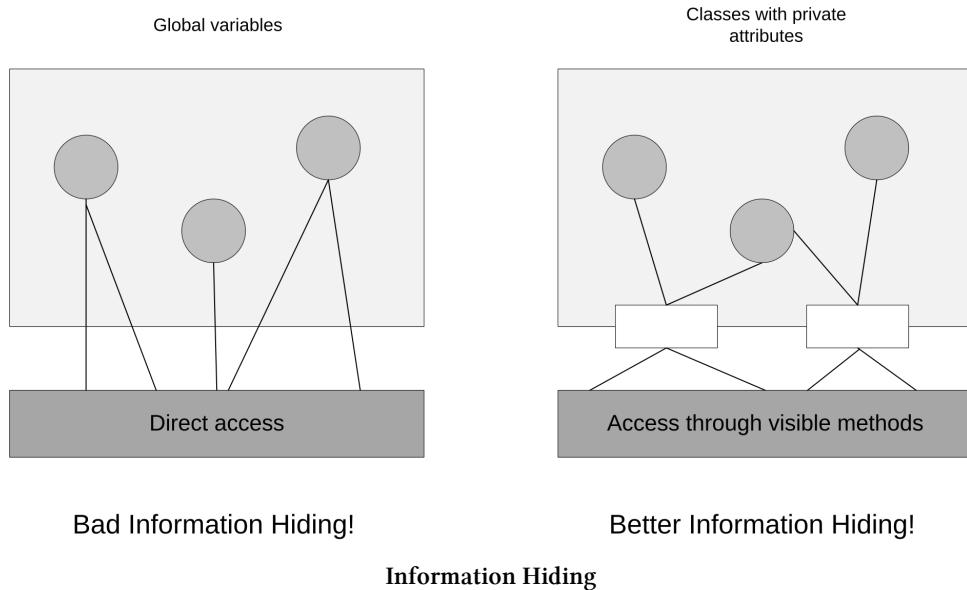
2.5 Criteria for *good design*

These criteria are fractal, i.e., they apply to all levels of the design like architecture, subsystems and components.

- **Correctness**
 - Fulfilment of requirements
 - Playback of all functions of the system model
 - Ensuring the non-functional requirements
- **Comprehensibility**
 - Self-explanatory code and design
 - Good documentation
- **High cohesion**
- **Low coupling**
- **Reusability**
- **Customizability**
- **No cyclic dependencies**

2.6 Information Hiding

Simplify and reduce access to a class by hiding details, methods and members that shouldn't be called and accessed by a client.



In Java, classes and class members are access controlled. This mechanism prevents the users of a package or class from depending on unnecessary details of the implementation of that package or class.

The access is specified by the access modifiers `public`, `protected` or `private`. In the absence of an access modifier, it is the default access, also called package-private access.

The following table and illustration show the access control rules in Java.

Visible for:	public	protected	(default)	private
Same Class	yes	yes	yes	yes
Different Class, Same Package	yes	yes	yes	no
Different Class, Different Package	yes	no	no	no
Subclass, Same Package	yes	yes	yes	no
Subclass, Different Package	yes	yes	no	no

Visibility in Java

According to the **Java Language Specification (JLS)**¹:

`public`

A `public` class member or constructor is accessible throughout the package where it is declared and from any other package.

¹<https://docs.oracle.com/javase/specs/jls/se15/html/jls-6.html#jls-6.6>

protected

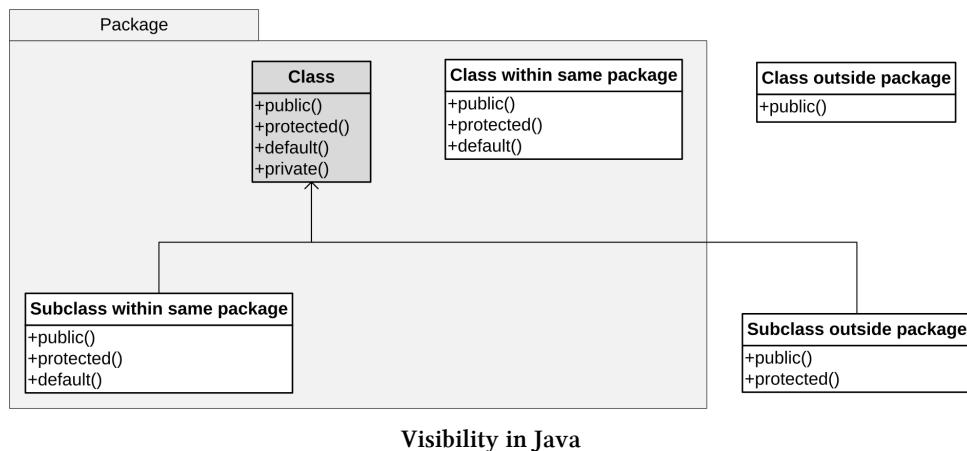
A `protected` member or constructor of an object may be accessed from outside the package in which it is declared only by code that is responsible for the implementation of that object. Can only be applied to fields, constructors and methods, not to classes.

package-private

If none of the access modifiers `public`, `protected`, or `private` are specified, a class member or constructor has package-private access. It is accessible throughout the package that contains the declaration of the class in which the class member is declared, but the class member or constructor is not accessible in any other package.

private

A `private` class member or constructor is accessible only within the body of the top-level class that encloses the declaration of the member or constructor.



Declare members as `private` and provide accessible methods for `private` fields. Let distributed classes communicate only through this method calls.

If a class, interface, method or field is part of a published API, it can be declared `public`. Other classes and members should be declared as either package-private or `private`. The exposure of fields and methods that give access to mutable state of a class via interfaces must be avoided. This is because interfaces only allow publicly accessible methods, which are part of the API of the class. An exception is the implementation of methods that expose a public immutable view of a modifiable object. Modifiable classes should provide copy functionality to allow secure passing of instances to the client code.

Inheritance and visibility:

If a method is overwritten, the new method cannot have stronger access control than the original. In the case of `private` no inheritance is possible. More precisely, no overwriting takes place and the access to the superclass method via `super` is not possible.

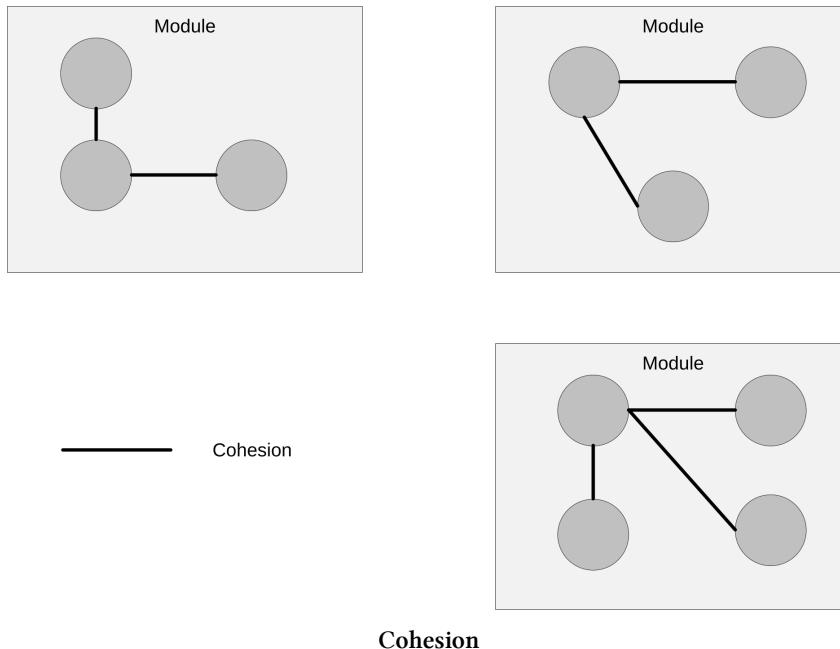
From / To:	public	protected	(default)	private
public	yes	no	no	no
protected	yes	yes	no	no
(default)	yes	yes	yes	no
private	no	no	no	no

Visibility and Inheritance

2.7 Cohesion

*Measure of the **affiliation** of the elements of a component.*

High cohesion of a component improves understanding, maintenance and adaptation.



Easy to maintain code usually has a high cohesion. The elements within the module are directly related to the functionality that the module is intended to provide. We can easily design, write and test our code because the code for a module is all together and works together.

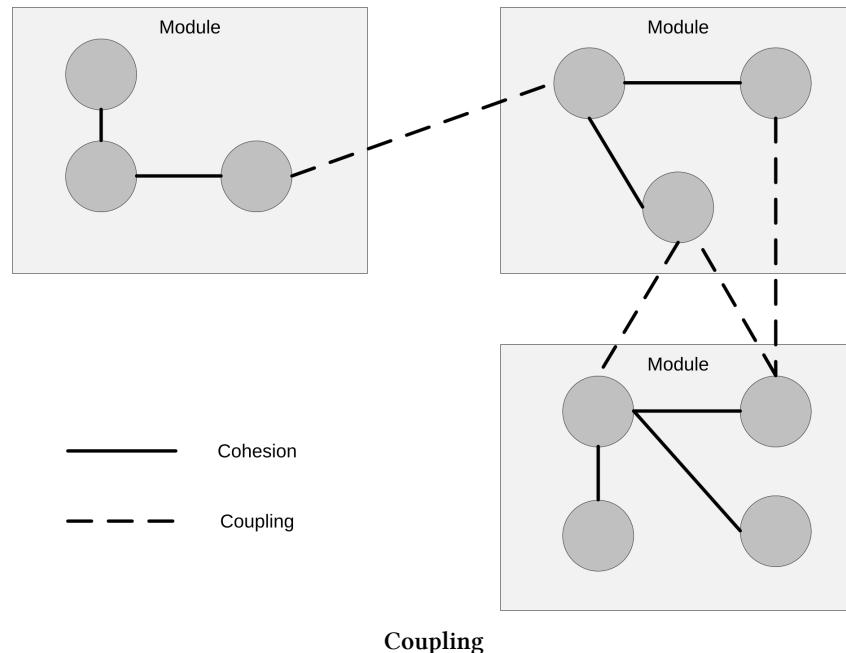
Low cohesion would mean that the functionality is distributed throughout your code base.

- **Degrees of cohesion**
 - functional cohesion
 - sequential cohesion
 - communicational cohesion
 - procedural cohesion
 - temporal cohesion
 - logical cohesion
 - coincidental cohesion

2.8 Coupling

Measure for the dependencies between components.

Low coupling facilitates maintainability and makes the system more stable.



Modules should be as independent as possible from other ones. Changes to one module should not have a major impact on other modules. A high coupling would mean that modules know too much about the internal of other application parts. Knowing too much about others make changes difficult to coordinate and increase the fragility.

With a low coupling, you can easily make changes to the internals without worrying about their effects on other modules. Low coupling also makes it easier to design, write and test code, as our modules are independent. We also have the advantage that our code is easy to reuse.

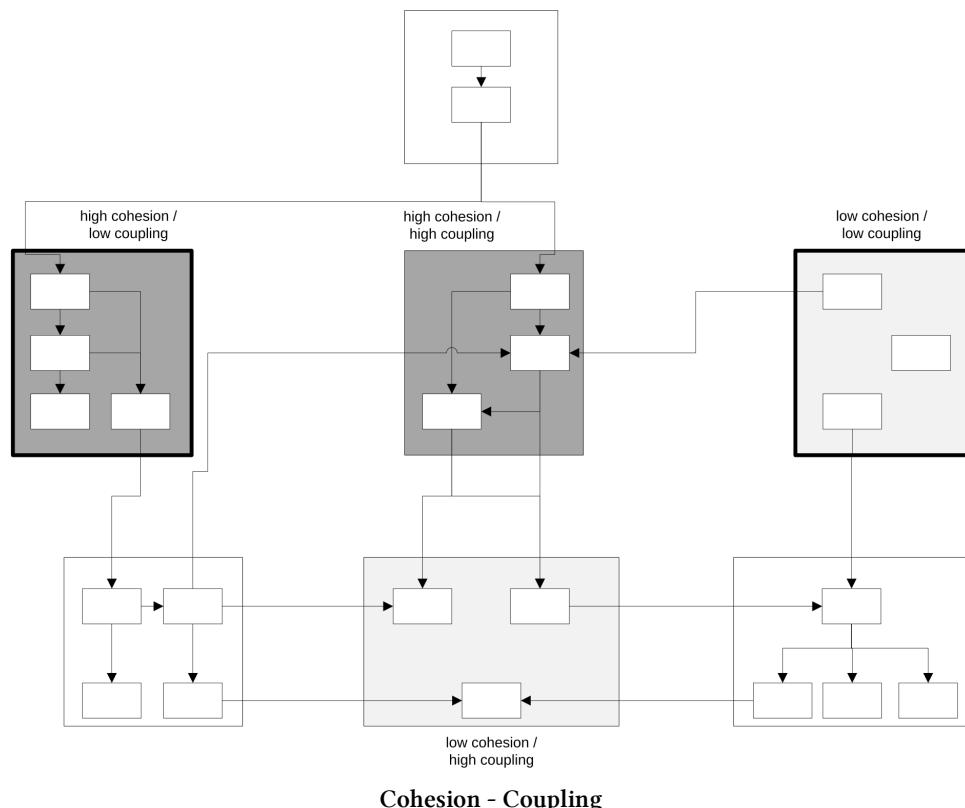
- **Consequences of high coupling**
 - Difficult to maintain, difficult to adapt
 - Monolithic
 - Exchange of components hardly possible
 - System becomes fragile
 - Difficult to reuse individual components
- **Types of coupling**
 - Data coupling
 - Interface coupling
 - Structural coupling
- **Reduction of coupling**
 - Coupling can never be reduced to zero!
 - Higher interface coupling increases flexibility
- **High cohesion enables low coupling**

2.9 Cohesion - Coupling

Cohesion and coupling are among the most important attributes for the quality of a design. In software development, coupling means the measure for the dependencies between components, while cohesion stands for the measure of the unity of the elements of a component. It is very important to understand what impact these two parameters can have on a system and how they can be influenced.

To sum it up:

- Cohesion represents the degree to which a part of a code base represents a logically single, atomic unit.
- Coupling represents the degree to which a single unit is independent of others.
- Encapsulate information, make modules **highly cohesive**, and **decrease coupling** among modules.
- Try to follow the guideline at all levels of your code base.
- It is impossible to achieve complete decoupling without affecting cohesion, and vice versa.



2.10 Big Ball of Mud



A Big Ball of Mud is a haphazardly structured, sprawling, sloppy, duct-tape-and-baling-wire, spaghetti-code jungle. These systems show unmistakable signs of unregulated growth, and repeated, expedient repair. Information is shared promiscuously among distant elements of the system, often to the point where nearly all the important information becomes global or duplicated.

The overall structure of the system may never have been well defined.

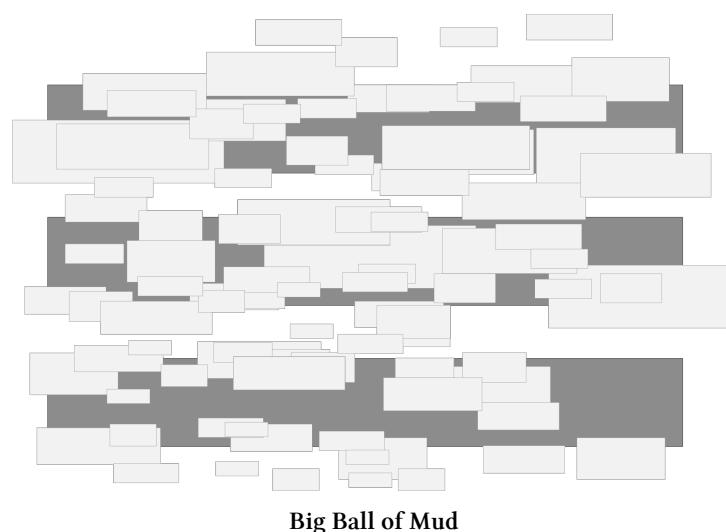
If it was, it may have eroded beyond recognition. Programmers with a shred of architectural sensibility shun these quagmires. Only those who are unconcerned about architecture, and, perhaps, are comfortable with the inertia of the day-to-day chore of patching the holes in these failing dikes, are content to work on such systems.

—Brian Foote and Joseph Yoder, *Big Ball of Mud*².

Big Ball of Mud is an anti-pattern of software architecture and describes a program that has no recognizable software architecture.

Most frequent causes:

- Insufficient experience
- Lack of awareness for software architecture
- Not modularized, high coupling and low cohesion
- Does not comply with the principles of good design
- Pressure on the implementation team
- Constantly changing requirements
- High time pressure
- Low budget
- Employee fluctuation



²<http://www.laputan.org/mud/mud.html#BigBallOfMud>

2.11 Architecture Principles

Keep it simple stupid (KISS)

- Think of several options and then try the simplest option first.
- Always ask: *Is there an easier way to do this?*
- Think about the future maintainers, assume that they will be you, and work on that basis.
- When you look at existing systems, ask: *Do I need all this stuff or Do we need it at all?*

You're not going to need it (YAGNI)

- The architecture should be intended to support current and future requirements agreed with business stakeholders.

Principle of least astonishment (POLA)

- Keep things consistent so that people are not surprised when they find that a similar task is being done in a different place in a different way.
- If a difference is needed, document why.
- Consistency guides understanding, if you name things wrong or call the same thing by different names, you increase complexity.

Separation of concerns (SoC)

- Keep related things together, unrelated things apart.

If it hurts, do it more often

- When we try to avoid difficulties, we are faced with increasing complexity. So, complexity is reduced if we perform difficult tasks more often.
- <https://martinfowler.com/bliki/FrequencyReducesDifficulty.html>

Leave your code better than you found it (Boy Scout Rule)

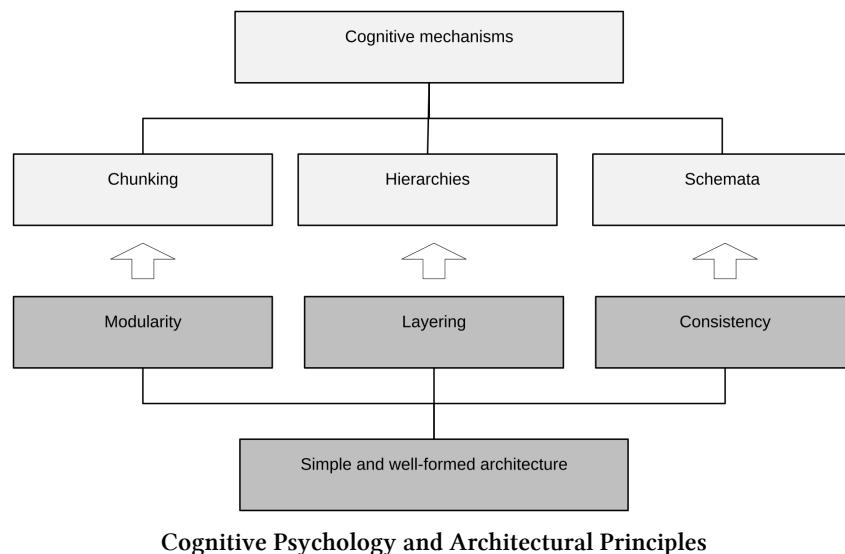
- Keep improving your code continuously, meaning that every time that we make a change in our code base, make an improvement in the code.
- No matter what the reasons are, we don't want to end up with an additional technical debt. Every time we see code smell, we should try to eliminate the rot.
- If we all follow this approach, the system would gradually improve, and the deterioration would stop.

Talk to people

- You can't find all the information you need in the documentation or existing code - you must talk to people.

2.12 Cognitive Psychology and Architectural Principles

In the long period of evolution, the human brain has acquired some impressive mechanisms that help us to deal with complex structures. These techniques should be used in software development so that maintenance and enhancement can be performed in a fast way. Our goal is to be able to develop our software products for a long-term period with a constant quality, also with varying development teams.



This mechanism our brain has developed for complex structures are **chunking**, **formation of hierarchies** and **building of schemata**. These techniques have direct mappings into the architectural design of a system. When these and their implementation in architecture are obvious to the development team, an essential base for a high-quality architecture is set.

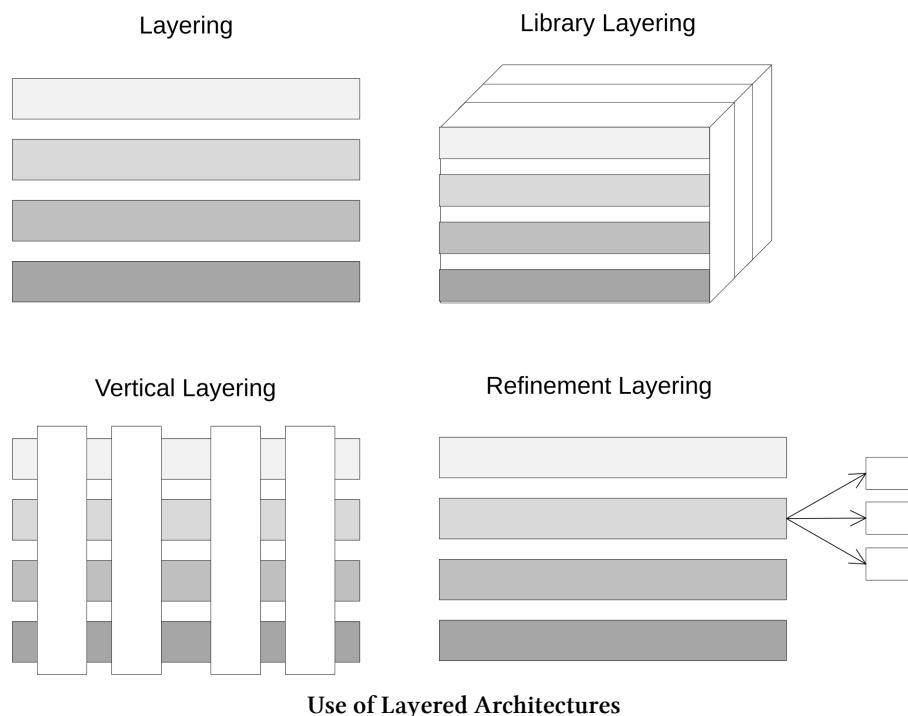
Carola Lilienthal described in her book *Sustainable Software Architecture, Analyze and Reduce Technical Debt* in detail how the cognitive mechanisms can be mapped to the software architecture. There you can find further details on the implementation of modularity, hierarchisation and pattern consistency.

2.13 Layered Architecture

2.13.1 Use of Layered Architecture

Architectural styles and patterns play an important role in software engineering. There are many possibilities for the architecture and design of an application. The complexity of an application depends on different requirements. Basically, a system should be designed in such a way that the architecture is easily reusable, extendable and maintainable and allows a clear separation of components.

Architectures should not be about frameworks. Frameworks are tools to be used which supports you in your desired design of an application. You should not be built your architecture around a framework.



A common approach is the so-called n-tier architecture. Here, the building blocks of the application are divided into layers according to their task. This layering was the de-facto standard for many applications and therefore is widely known by most architects, designers, and developers.

Although the layer architecture pattern does not specify the number and types of layers, most layer architectures consist of three or four layers. For example, smaller applications may have only three layers, while larger and more complex business applications may have five or more layers.

A three-tier architecture consists of, for example, the persistence layer, the business layer and the presentation layer. It includes clearly defined interfaces and a defined direction of the data flow. To minimise the coupling of the layers, domain objects are used to transport the data from layer to layer.

Traditional software architecture defines the following three main layers for a layered software design:

Presentation Layer

The task of the presentation layer is to respond to client requests. On the one hand, it presents the data on an interface in a proper form and on the other hand, it includes the execution and reaction to user actions. When implementing and realising this layer, the challenge is to keep it free of business logic. For this reason, the presentation layer should not implement business logic, but only call and use it.

Business Layer

The business layer represents a logical and presentation-independent layer. In it the business logic is programmed, i.e., everything that must be done independently of a concrete presentation. It contains the entire functionality of the business requirements developed in the use cases. The focus during implementation is on the grouping of related components and abstraction of the architecture to be developed.

Persistence Layer

The persistence layer is responsible for the actual access to the data in the database. The database access must not be performed by any other layer than the persistence layer in order to achieve the highest possible level of encapsulation. For the higher layers it must be irrelevant in which kind the objects are stored. For this reason, no direct database call may be used in the business or presentation layer. The persistence layer is only called via the business objects of the business layer.

How do you organize your classes into packages?

Imagine you're writing a web application. In this application you handle tickets and reservations. Your classes include classes like `TicketController`, `TicketService`, `TicketRepository` for the ticket handling and `RegistrationController`, `RegistrationService` and `RegistrationRepository` for registration. So, how do you organize your classes into packages and create a valid architecture?

There are two different ways to structure your packages. You can focus on the technical tiers:

- `com.swsc.web`
- `com.swsc.service`
- `com.swsc.repository`

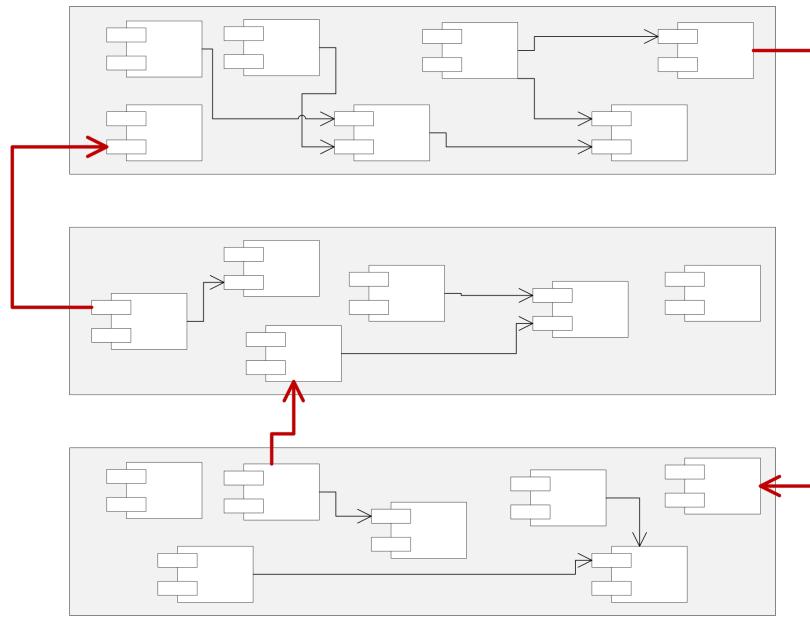
Or you can focus on the domain and separate the packages into feature driven approach:

- `com.swsc.registration`
- `com.swsc.ticket`

This approach will be discussed in detail in the following chapters.

2.13.2 Violated Layered Architecture

In this example a request not moves from layer to layer, these violates the layered architecture. It must go through the layer right below it to get to the next layer below that one.



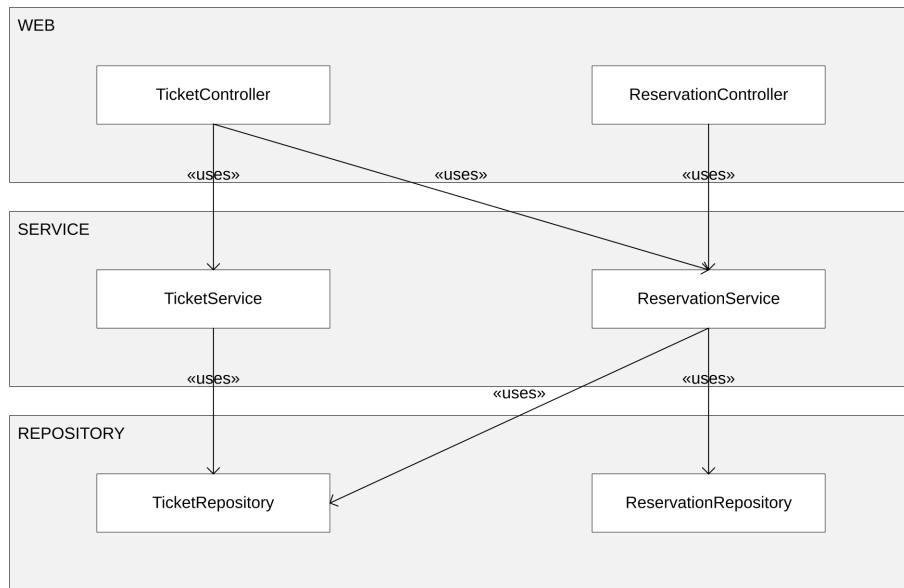
Violated Layered Architecture

The access of a lower layer to an upper one is also a violation in this example.

The access and the agreements concerning the architecture of an application should be known within the team or preferably checked automatically. A method that makes this possible is presented in the chapter *Testing the Architecture and Design*. This architecture should not disappear over time and should be valid.

2.13.3 Horizontal Layering

The classic horizontally layered architectural pattern was a solid general-purpose pattern and therefore a good choice for most applications, especially if you were not sure which architectural pattern was best for your application.



Horizontal Layering

Today, however, it is no longer the preferred solution. There are a few things to keep in mind when choosing this pattern from an architectural point of view. With this pattern you run the risk of building a monolith. This pattern cuts the layers horizontally according to purely technical responsibility. This paradigm **eliminates low coupling and high cohesion**. This is exactly the opposite of what we want to achieve.

The different modules become quickly dependent on other parts of the application and are not clearly separated from each other. If an application is structured purely according to horizontal layers and not according to features like in this case *Ticket* and *Reservation* you end up with a pure layer architecture without a clear division into feature-based modules.

When you look at the package structure you will recognize that almost all classes must have a `public` modifier in order to have access to the lower layer. Which is certainly not the best approach and violates the principle of information hiding.

Package structure - Horizontal Layering

```

1 com.swcs.web
2     (C) TicketController
3     (C) ReservationController
4 com.swcs.service
5     (C) TicketService
6     (C) ReservationService
7 com.swcs.repository
8     (C) TicketRepository
9     (C) ReservationRepository

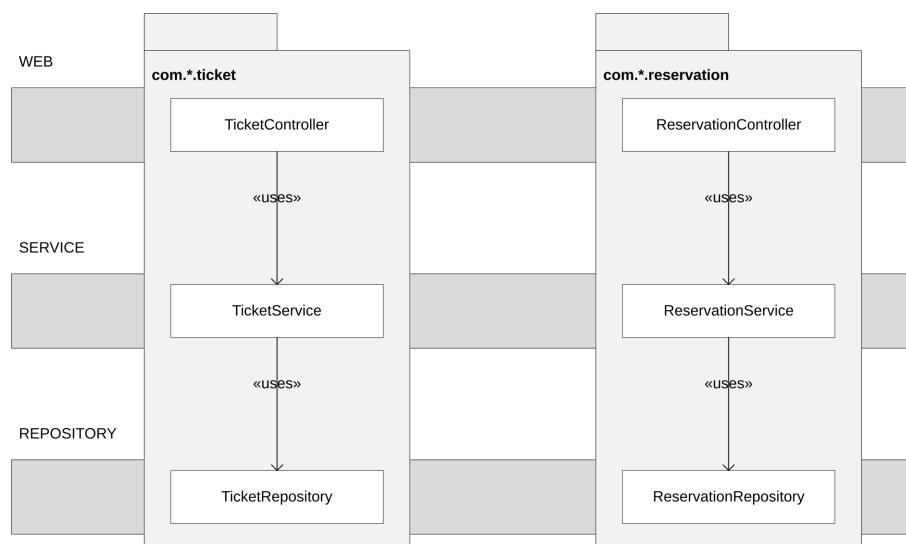
```

2.13.4 Feature Based Layering - Single package

A better approach would therefore be to divide the application in **vertical slices** according to functional responsibilities and keep the technical layering as well. Each module is given responsibility for a clear and separate functional context. So, instead of coupling across a layer, we couple vertically along a slice. Packaging by feature improves coupling and cohesion by keeping all classes related in the same package or module. This minimizes coupling between slices and maximizes coupling in a slice. In the long run, this decoupling in slices simplifies code maintenance because it is clear which part of the code does what. We can make changes without touching or fully understanding the rest of the code. It makes it easier to make changes to individual parts of the application, improving both maintainability and expandability.

The two concepts of cohesion and coupling are useful and important, but there is another point that comes into play in this approach.

The package and module structure by “Package by Feature” help the developer to see what it does and what functionality it has. The architecture literally jumps out at you. This is a step towards what Uncle Bob calls [Screaming Architecture³](#).



Feature Based Layering - Single package

With the single package approach, no sub-packages are used for the individual horizontal layers. This makes it possible to set all classes to package-private and thus prevent the use of these outside the package. However, this can affect the clarity of large modules.

Both horizontal and vertical slicing have their own use cases, but it is obvious that vertical combined with the horizontal slicing enables a more modern and adaptable architecture.

This approach makes it possible to reduce the visibility of the classes. They can be set to package-private and are therefore only available within the feature slices. In the horizontal layering, this was not possible because related classes had to communicate across packages, forcing us to make everything public. The horizontal layering within a slice is still present. The naming conventions like `*Controller`, `*Service` and `*Repository` define the layering.

³<https://blog.cleancoder.com/uncle-bob/2011/09/30/Screaming-Architecture.html>

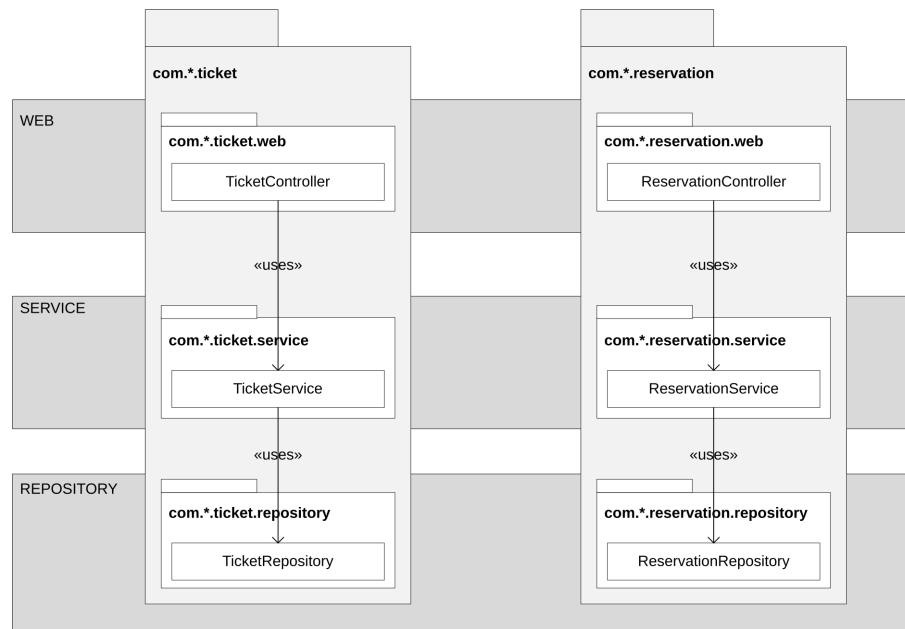
Package structure - Feature Based Layering - Single package

```
1 com.swcs.ticket
2   (C) TicketController
3   (C) TicketService
4   (C) TicketRepository
5
6 com.swcs.reservation
7   (C) ReservationController
8   (C) ReservationService
9   (C) ReservationRepository
```

The advantages are obvious:

- You get a low coupling and a high cohesion of the slices.
- The code structure corresponds to the domain, which ensures a consistent understanding and communication between the developers and the department.
- As new features and enhancements are mostly technical driven, it is easier to identify the appropriate modules and to determine the level of change.
- Changes within a single functional domain can happen within a single part of the system. Side effects are avoided or reduced in this way, because the modules are independent of each other.
- Vertical separation promotes forming hard contextual boundaries. Vertical layering enables easier architectural transformation from a monolithic structure to a distributed structure built like micro-services.
- Testability is better because the technical logic of a module is separate from other modules.

2.13.5 Feature Based Layering - Slices before layers



Feature Based Layering - Slices before layers

The approach of *Slices before layers* divides the vertically feature slice into technical layers such as web, service and repository. This makes the technical layers within the package easier to recognize. This serves mainly the organization of the code base and improves the structure of the code.

However, a very important advantage is lost. In order to access each other, the classes have to be marked with public with the consequence that they are visible for other feature slices.

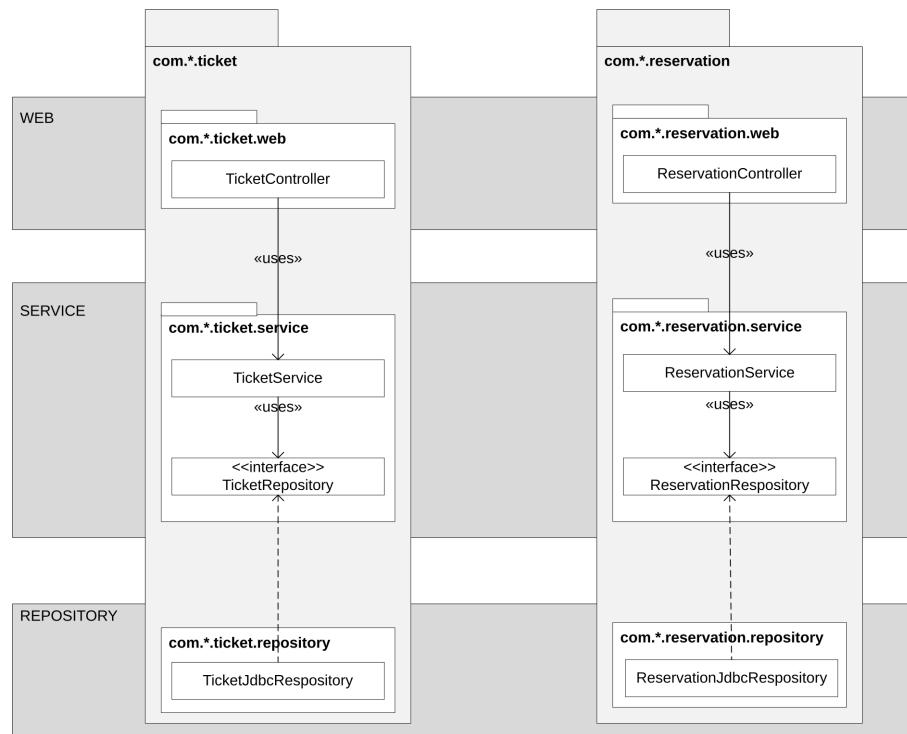
Package structure - Feature Based Layering - Slices before layers

```

1 com.swcs.ticket.web
2     (C) TicketController
3 com.swcs.ticket.service
4     (C) TicketService
5 com.swcs.ticket.repository
6     (C) TicketRepository
7
8 com.swcs.reservation.web
9     (C) ReservationController
10 com.swcs.reservation.service
11     (C) ReservationService
12 com.swcs.reservation.repository
13     (C) ReservationRepository

```

2.13.6 Feature Based Layering – Hexagonal Architecture



Feature Based Layering - Hexagonal Architecture

The [hexagonal architecture](#)⁴, or port and adapter architecture, is an architectural pattern used in software design to structure large applications. It is designed to create loosely coupled application components by using ports and adapters.

This architectural style can also be applied to code within a module. The principle behind it is that the domain logic is centrally encapsulated. The interaction with the user and server side with this domain lies outside the core. The user side can be an API or a web interface. The server side can be a database. The key point here is that the outside world always depends on the domain core and never the other way around.

In the example, the service package defines the interfaces like `TicketRepository` for the actual access logic to the database. The actual implementation of the interface `TicketJdbcRepository` is located in the package repository and therefore points in the direction of the domain package.

Package structure - Feature Based Layering - Hexagonal Architecture

```

1 com.swcs.ticket.web
2   (C) TicketController
3 com.swcs.ticket.service
4   (C) TicketService
5   (I) TicketRepository
6 com.swcs.ticket.repository
7   (C) TicketJdbcRepository
8
9 com.swcs.reservation.web
10  (C) ReservationController

```

⁴<https://alistair.cockburn.us/hexagonal-architecture/>

```
11 com.swcs.reservation.service
12     (C) ReservationService
13     (I) ReservationRepository
14 com.swcs.reservation.repository
15     (C) ReservationJdbcRepository
```

The advantage of this approach is that the entire business logic is defined centrally in a business component and is decoupled from the outside world. But the visibility here is also too much open.

Reducing dependencies with application events

Direct dependencies between modules are usually created by direct calls and the use of classes from another module. Dependencies should be reduced as far as possible. Cyclic dependencies should be avoided at all costs.

There are several ways to reduce them like application events. Application events can be used for loosely coupled components to exchange information. By replacing the direct call by an application event, you can reverse the dependencies. A module creates an event of a certain type, which is located within the module. Another module registers an event listener, which listens and reacts to this event type. The received module thus has a dependency on the sending module, because it knows this event type and not vice versa. This is especially useful for calls with direct commands to another module like audit logs.

2.13.7 The Java Module System

With the [Java Module System⁵](#) introduced in Java 9, you get another way to add visibility rules to your architecture. The above examples were designed without the capabilities of the module system. But if you use this, it will make it easier for you to organize your code.

The modularity adds a higher level of aggregation above the packages. The packages in one module are only accessible to other modules if the module explicitly exports them. Even then, another module can only use these packages if it explicitly declares that it needs the functionality of the other module.

You will find that taking the module system into account helps you to develop cleaner, more consistent designs. You don't need a modular system to design for modularity, but a modular system makes this much easier.

The book *The Java Module System* by Nicolai Parlog gives a detailed insight into the features of the module system.

⁵<https://openjdk.java.net/jeps/261>

2.14 Architecture Documentation

Designing architecture means making decisions! The documentation is a persistent and communicable artefact for all decisions. It is taken into account during the planning of the project and focuses more on non-functional than functional requirements. The team works closely together to build a system that meets all requirements. Some of the development work will address needs other than functional requirements. Therefore, the work on improving the quality of the system must take up a substantial part of the team's time.

In order to give all stakeholders of a system, that require information about the architecture, e.g., the internal structure, crosscutting concepts or fundamental decisions a fundamental architecture overview, it is important to provide an architecture documentation.

[Arc42⁶](#) provides a comprehensive but condensed template for describing architectures. This template produces a uniformed architecture documentation which helps to understand the big picture and connect the dots by looking at multiple area.

The template is structured as follows, and contains 12 structure [items⁷](#):

1. Introduction and Goals

Short description of the requirements, driving forces, extract of requirements. Top three or max five quality goals for the architecture which have highest priority for the major stakeholders. A table of important stakeholders with their expectation regarding architecture.

2. Constraints

Anything that constrains teams in design and implementation decisions or decision about related processes. Can sometimes go beyond individual systems and are valid for whole organizations and companies.

3. Context and Scope

Delimits your system from its external communication partners. Specifies the external interfaces.

4. Solution Strategy

Summary of the fundamental decisions and solution strategies that shape the architecture. Can include technology, top-level decomposition, approaches to achieve top quality goals and relevant organizational decisions.

5. Building Block View

Static decomposition of the system, abstractions of source-code, up to the appropriate level of detail.

6. Runtime View

Behaviour of building blocks as scenarios, covering important use cases or features, interactions at critical external interfaces, operation and administration plus error and exception behaviour.

7. Deployment View

⁶<https://arc42.org/>

⁷<https://arc42.org/overview/>

Technical infrastructure with environments, computers, processors, topologies. Mapping of software building blocks to infrastructure elements.

8. Crosscutting Concepts

Overall, principal regulations and solution approaches relevant in multiple parts of the system.

9. Architectural Decisions

Important, expensive, critical, large scale or risky architecture decisions including rationales.

10. Quality Requirements

Quality requirements as scenarios, with quality tree to provide high-level overview.

11. Risks and Technical Debt

Known technical risks or technical debt.

12. Glossary

Important domain and technical terms that stakeholders use when discussing the system.

2.15 Testing the Architecture and Design

The architecture of a system usually is a design that was made at a certain moment in time. However, as the system evolves and developers come and go, it's hard to keep everyone up to date on a reference architecture or encourage developers to make changes to it, since it's usually in the form of documents, confluence pages, or something similar. This can result in code that violates architectural specifications increasing the technical debt.

IntelliJ⁸

IntelliJ comes with a simple tool for architecture analysis. With it you can generate a dependency matrix for your code.

Eclipse⁹

In eclipse you can install various plugins to visualise dependencies and to detect cycles. eDepend¹⁰, STAN¹¹, jDepend¹², Java Dependency Viewer¹³

ArchUnit¹⁴

With ArchUnit you can check the architecture constraints of your Java application automatically and can run as part of a test suite. It is a free, simple and extensible library for checking the architecture of Java code, which includes the ability to check dependencies between packages and classes, layers and slices, cyclic dependencies and more.

jQAssistant¹⁵

jQAssistant is a QA tool, which allows the definition and validation of project specific rules on a structural level. It is built upon the graph database Neo4j and can easily be plugged into the build process to automate detection of constraint violations and generate reports about user defined concepts and metrics.

Structure 101¹⁶

With Structure 101 you can understand, analyse, (drag 'n drop) refactor, and control large, complex codebases.

Sonargraph¹⁷

Sonargraph is a powerful static code analyser that allows you to monitor a software system for technical quality and enforce rules regarding software architecture, metrics and other aspects in all stages of the development process.

Lattix¹⁸

Lattix enables you to quickly identify and remediate architectural issues. It gives software architects a fast and visual way to represent an application's architecture with the Dependency Structure Matrix technology.

Teamscale¹⁹

Teamscale supports your team to analyse, monitor, and improve the quality of your software.

⁸<https://www.jetbrains.com/>

⁹<https://www.eclipse.org/>

¹⁰<https://marketplace.eclipse.org/category/free-tagging/edepend-graphical-dependency-analysis-tool-340>

¹¹<http://stan4j.com/eclipse/>

¹²<https://marketplace.eclipse.org/category/free-tagging/jdepend>

¹³<https://marketplace.eclipse.org/content/java-dependency-viewer>

¹⁴<https://www.archunit.org/>

¹⁵<https://jqassistant.org/>

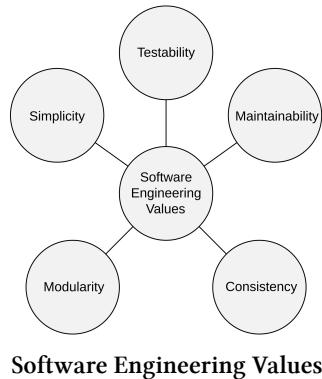
¹⁶<https://structure101.com/>

¹⁷<https://www.hello2morrow.com/products/sonargraph>

¹⁸<https://www.lattix.com/>

¹⁹[https://www.cqse.eu/en/teamscale/overview/](https://www.cqse.eu/en/teamscale/overview)

2.16 Software Engineering Values



Testability

- To ensure the functional correctness of the system, it is essential that each unit of the system is verifiable and has been tested.
- A system is only complete if its functional correctness and completeness is proven by automatically triggered tests.

Maintainability

- We build durable systems, which means we always must consider how to maintain the systems.
- The colleagues who maintain a system can change over time. Therefore, a maintainable system should be easy to understand. The level of system specific knowledge required to carry out maintenance should be kept to a minimum.

Consistency

- To improve flexibility and maintenance, the number of different solutions like code-level patterns, libraries, third party services and application integration patterns, for the same problem should be kept to a minimum.
- Having multiple solutions to a problem is better than having no solutions, having a single solution to a problem is the best.

Modularity

- Things are always changing. To protect our code from this fundamental fact, we need to limit the impact of change and divide it into reasonably independent, coherent units at all levels.

Simplicity

- The simplest parts of a system to change are those that do not exist. For any given requirement, the simplest, minimalist solution should be provided which meets the previous values.

2.17 Team Charter

A team charter is a document which clarifies the direction of the team by setting a **vision**, **mission** and **boundaries** that everyone on the team supports. As a source, it illustrates the focus and direction of the team to others in the organization. The charter should be developed within the team to promote clear understanding and agreement, providing clarity and reducing confusion in cases where conflicting asks or projects arise.

A Team Charter could looks like the follows example:

From a team-facing practice view we want

- create an open, no blame, learning and feedback culture
- create an environment that fosters teamwork and innovation
- create a collective ownership with no knowledge silos
- developing and utilizing everyone's abilities to the fullest
- communicate effectively and talk to each other
- provide honest feedback, act upon the feedback and drive for continuous improvement
- collaborate and do pair programming at a level that drives highest quality
- be a whole team and the unit of delivery
- be a professional software craftsman

From a technical practice view we want

- keep things simple
- encourage continual improvement through refactoring
- closing the feedback loop frequently through CI/CD
- that everybody helps to fix the build immediately
- monitor CI/CD/SonarQube after push
- keep our products free of technical debt
- follow our standards and best practices
- enforce the DoD and our Code review Checklist

3. Java Best Practices

3.1 Communicate through code

There are only two hard things in Computer Science: cache invalidation and naming things.

– Phil Karlton

3.1.1 Use Java code conventions and avoid misinformation

Bad code

```
1 public static final double pi = 3.14159265358979323846;
2 int YEAR;
3 String first_name;
4 public class convert_strategy {}
5 Set carList;
```

Good code

```
1 public static final double PI = 3.14159265358979323846;
2 int year;
3 String firstName
4 public class ConvertStrategy {}
5 Set cars;
```

Bad code

```
1 Customer Customer = new Customer("Darth Vader");
2 ...
3 Customer.name() // static access?
```

Good code

```
1 Customer customer = new Customer("Darth Vader");
2 ...
3 customer.name()
```



Java code conventions can be found here:

[Oracle Java code conventions¹](https://www.oracle.com/technetwork/java/codeconventions-150003.pdf)

[Google Java Style Guide²](https://google.github.io/styleguide/javaguide.html)

¹<https://www.oracle.com/technetwork/java/codeconventions-150003.pdf>

²<https://google.github.io/styleguide/javaguide.html>

3.1.2 Choose an expressive name and avoid mental mapping

Bad code

```
1 int d; // days since birthday
2 String n; // name of user
```

Good code

```
1 int daysSinceBirthday;
2 String name;
```

3.1.3 Make differences clear with meaningful variable names

Bad code

```
1 public boolean isFileOlder(File s1, File s2) {};
2
3 public static String replaceFirst(String string, Pattern pattern, String with) {};
```

Good code

```
1 public boolean isFileOlder(File file, File referenceFile) {};
2
3 public static String replaceFirst(String text, Pattern regex, String replacement) {};
```

3.1.4 Use pronounceable names

Bad code

```
1 class DRcd {
2     private int rtd;
3     private String stna;
4     private LocalDateTime genymdhms;
5     private LocalDateTime modymdhms;
6 }
```

Good code

```
1 class Student {
2     private int registrationId;
3     private String name;
4     private LocalDateTime generated;
5     private LocalDateTime modified;
6 }
```

3.1.5 Do not hurt the readers

We will read more code than we write, so do not hurt the readers. Declare class variables with useful constants. In this way, the meaning or intended use of each literal is clearly indicated. If the constant needs to be changed, the change is also limited to the declaration. There is no need to search and change the code for this literal.

Bad code

```
1 // What is 86400000?  
2 setTokenTimeout(86400000);
```

Good code

```
1 private static final int MILLISECONDS_IN_A_DAY = 24 * 60 * 60 * 1000;  
2  
3 setTokenTimeout(MILLISECONDS_IN_A_DAY);
```

Bad code

```
1 public final class Sphere {  
2  
3     private Sphere() {  
4         }  
5  
6     public static double area(double radius) {  
7         return 3.14 * radius * radius;  
8     }  
9  
10    public static double volume(double radius) {  
11        return 4.19 * radius * radius * radius; // Why 4.19?  
12    }  
13 }
```

Good code

```
1 public final class Sphere {  
2  
3     private Sphere() {  
4         }  
5  
6     public static double area(double radius) {  
7         return Math.PI * radius * radius;  
8     }  
9  
10    public static double volume(double radius) {  
11        return 4.0 / 3.0 * Math.PI * radius * radius * radius;  
12    }  
13 }
```

3.1.6 Don't add redundant context

Bad code

```
1 class CarData {  
2     private String carManufacturer;  
3     private String carModel;  
4 }
```

Good code

```
1 class Car {  
2     private String manufacturer;  
3     private String model;  
4 }
```

Bad code

```
1 public interface UserRepository {  
2  
3     Optional<User> findUserById(long userId);  
4  
5     Iterable<User> findAllUsers();  
6  
7     User saveUser(User user);  
8  
9     Iterable<User> saveAllUsers(Iterable<User> users);  
10  
11    void deleteUserById(long userId);  
12  
13    void deleteAllUsers();  
14  
15    long countUsers();  
16 }
```

Good code

```
1 public interface UserRepository {  
2  
3     Optional<User> findById(long id);  
4  
5     Iterable<User> findAll();  
6  
7     User save(User user);  
8  
9     Iterable<User> saveAll(Iterable<User> users);  
10  
11    void deleteById(long id);  
12  
13    void deleteAll();  
14  
15    long count();  
16 }
```

3.1.7 Don't add words without additional meaning

There are some generic words that often don't add any additional meaning to a name. So, they mean anything when added to a name.

These words often give an indication of this constellation:

- Data
- Information
- Bean
- Value
- Manager
- Object
- Entity
- Instance

By adding this kind of words, double-check if the name still means the same if you remove it? If yes, remove it. If no, keep it or better try to find a better name.

Bad code

```
1 public final class StudentData {
2     private final String studentFirstName;
3     private final String studentLastName;
4     private final AddressInformation addressInformation;
5     private final CoursesBean coursesBean;
6
7     public StudentData(String studentFirstName, String studentLastName,
8         AddressInformation addressInformation, CoursesBean coursesBean) {
9         this.studentFirstName = studentFirstName;
10        this.studentLastName = studentLastName;
11        this.addressInformation = addressInformation;
12        this.coursesBean = coursesBean;
13    }
14 }
```

Good code

```
1 public final class Student {
2     private final String firstName;
3     private final String lastName;
4     private final Address address;
5     private final Courses courses;
6
7     public Student(String firstName, String lastName, Address address, Courses courses) {
8         this.firstName = firstName;
9         this.lastName = lastName;
10        this.address = address;
11        this.courses = courses;
12    }
13 }
```

3.1.8 Don't use *and* or *or* in method names

Function names using *and* or *or* are an indicator, that the method has to many responsibilities and combine to much functionality.

How to improve this:

- Split the method into smaller pieces and extract the method in separate methods.
- If the things really belong together:
 - Consider finding a better name for that entire operation.
 - Consider creating something else that encapsulates the combination.

Bad code

```
1 public class JavaMailSender {  
2  
3     public void createMimeMessageAndSend() throws MailException {}  
4  
5     public Transport connectAndGetTransport(Session session)  
6         throws NoSuchProviderException, MessagingException {}  
7 }
```

Good code

```
1 public class JavaMailSender {  
2  
3     public MimeMessage createMimeMessage() {}  
4  
5     public void send(MimeMessage mimeMessage) throws MailException {}  
6  
7     public Transport connectTransport() throws MessagingException {}  
8  
9     public Transport transport(Session session) throws NoSuchProviderException {}  
10 }
```

3.1.9 Respect the order within classes

- Be consistent with the order of the class attributes, the constructor parameters and its methods.
- Align the order of the constructor parameter and the methods with the order of the class attributes declaration order.
- Private methods should be below the calling method. Thus, the code tells a story with the related methods below it. Furthermore, searching and scrolling is prevented.

Bad code

```
1 public class WebServer {  
2  
3     private final int port;  
4     private final String hostname;  
5  
6     public WebServer() {  
7         this("localhost", 8080);  
8     }  
9  
10    public WebServer(String hostname, int port) {  
11        this.port = port;  
12        this.hostname = hostname;  
13    }  
14  
15    public String hostname() {  
16        return this.hostname;  
17    }  
18  
19    public void start() {  
20        if (isPortFree()) {  
21            ...  
22        }  
23    }  
24  
25    public int port() {  
26        return this.port;  
27    }  
28  
29    public void stop() {  
30        ...  
31        destroyThreadPool();  
32        ...  
33    }  
34  
35    private boolean isPortFree() { ... }  
36  
37    private void destroyThreadPool() { ... }  
38 }
```

Good code

```
1  public class WebServer {  
2  
3      private final int port;  
4      private final String hostname;  
5  
6      public WebServer() {  
7          this(8080, "localhost");  
8      }  
9  
10     public WebServer(int port, String hostname) {  
11         this.port = port;  
12         this.hostname = hostname;  
13     }  
14  
15     public int port() {  
16         return this.port;  
17     }  
18  
19     public String hostname() {  
20         return this.hostname;  
21     }  
22  
23     public void start() {  
24         if (isPortFree()) {  
25             ....  
26         }  
27     }  
28  
29     private boolean isPortFree() { ... }  
30  
31     public void stop() {  
32         ...  
33         destroyThreadPool();  
34         ...  
35     }  
36  
37     private void destroyThreadPool() { ... }  
38 }
```

3.1.10 Group by line break

Use line breaks to make associations with related things and separate weakly related things.

Bad code

```

1  public final class FileUtils {
2      public static final long ONE_KB = 1024;
3      public static final long ONE_MB = ONE_KB * ONE_KB;
4      public static final long ONE_GB = ONE_KB * ONE_MB;
5      public static final File[] EMPTY_FILE_ARRAY = new File[0];
6      private FileUtils() {};
7      public static void cleanDirectory(File directory) throws IOException {
8          File[] files = verifyFiles(directory);
9          List<Exception> causes = new ArrayList<>();
10         for (File file : files) {
11             try {
12                 forceDelete(file);
13             } catch (final IOException ioe) {
14                 causes.add(ioe);
15             }
16         }
17         if (!causes.isEmpty()) {
18             throw new IOException(causes);
19         }
20     }
21 }
```

Good code

```

1  public final class FileUtils {
2
3      public static final long ONE_KB = 1024;
4      public static final long ONE_MB = ONE_KB * ONE_KB;
5      public static final long ONE_GB = ONE_KB * ONE_MB;
6
7      public static final File[] EMPTY_FILE_ARRAY = new File[0];
8
9      private FileUtils() {};
10
11     public static void cleanDirectory(File directory) throws IOException {
12         File[] files = verifyFiles(directory);
13         List<Exception> causes = new ArrayList<>();
14
15         for (File file : files) {
16             try {
17                 forceDelete(file);
18             } catch (final IOException ioe) {
19                 causes.add(ioe);
20             }
21         }
22
23         if (!causes.isEmpty()) {
24             throw new IOException(causes);
25         }
26     }
27 }
```

3.1.11 Prefer self-explanatory code instead of comments

Don't comment bad code - rewrite it.

—Brian W. Kernighan, The Elements of Programming Style

- Comments are no substitute for bad code.
- Explain it through code.
- Java is a programming language and should express itself.
- The more self-explanatory the code is, the less you need to comment it.
- Treat a comment as a danger signal.
- Comments are like deo for stinky code.
- Comment WHY not WHAT!

Bad code

```

1 final class InchToPointConvertor {
2
3     private InchToPointConvertor(){}
4
5     // convert the quantity in inches to points
6     static float parseInch(float inch) {
7         return inch * 72; // one inch contains 72 points
8     }
9 }
```

Good code

```

1 final class InchToPointConvertor {
2
3     private static final int POINTS_PER_INCH = 72;
4
5     private InchToPointConvertor(){}
6
7     static float toPoints(float inch) {
8         return inch * POINTS_PER_INCH;
9     }
10 }
```

Bad code

```

1 // Check if employee has entitlement for extra vacation days
2 if ((employee.noVacationDaysLeft && employee.age > 50)
3     || (employee.noVacationDaysLeft && employee.hasChildren))
```

Good code

```

1 if (employee.hasEntitlementForBonusVacation())
```

Bad code

```
1 class Passwords {
2 ...
3     // check if the password is complex enough
4     public boolean isComplexPassword(String password) {
5         boolean symbolOrDigitFound = false; // found a digit or symbol?
6         boolean letterFound = false; // found a letter?
7
8         for (int i = 0; i < password.length(); i++) {
9             char c = password.charAt(i);
10
11             if (Character.isLowerCase(c) || Character.isUpperCase(c)) {
12                 letterFound = true;
13             } else {
14                 symbolOrDigitFound = true;
15             }
16         }
17
18         return letterFound && symbolOrDigitFound;
19     }
20 }
```

Good code

```
1 class Passwords {
2 ...
3     public boolean isComplex(String password){
4         return containsLetter(password) && (containsDigit(password) || containsSymbol(password));
5     }
6
7     private boolean containsLetter(String password) { ... }
8
9     private boolean containsDigit(String password) { ... }
10
11    private boolean containsSymbol(String password) { ... }
12 }
```

3.1.12 Refactor step by step

Bad code

```

1 public List<int[]> getList() {
2     List<int[]> data = new ArrayList<int[]>();
3     for (int[] x : items)
4         if (x[0] == 4) // 4 represents flagged
5             data.add(x);
6     return data;
7 }
```

Better code

```

1 public List<int[]> flaggedCells() {
2     List<int[]> flaggedCells = new ArrayList<int[]>();
3     for (int[] cell : gameBoard) {
4         if (cell[STATUS] == FLAGGED)
5             flaggedCells.add(cell);
6     }
7 }
```

Good code

```

1 public List<Cell> flaggedCells() {
2     List<Cell> flaggedCells = new ArrayList<Cell>();
3     for (Cell cell : gameBoard) {
4         if (cell.isFlagged())
5             flaggedCells.add(cell);
6     }
7 }
8 return flaggedCells;
9 }
```

Craftsman code

```

1 public List<Cell> flaggedCells() {
2     return gameBoard.stream()
3         .filter(Cell::isFlagged)
4         .toList();
5 }
```

More generic craftsman code

```

1 public List<Cell> cells(Predicate<Cell> filter) {
2     return gameBoard.stream()
3         .filter(filter)
4         .toList();
5 }
```

3.2 Bad comments

3.2.1 Redundant comments

Bad code

```
1 // Deletes a directory and checks if it is a directory.
2 // Throw IOException if not exists.
3 public static void deleteDirectory(File directory) throws IOException {
4     if (directory != null && directory.isDirectory() && !directory.exists()) {
5         return;
6     }
7
8     if (!directory.delete()) {
9         throw new IOException("Unable to delete directory " + directory + ".");
10    }
11 }
```

- Reading the comment probably takes longer than reading the code itself.
- The comment is certainly no more informative than the code.
- Neither does it justify the code, nor does it state its purpose or reason for existence.
- The commentary is less precise than the code and seduces the reader to accept this lack of precision.

3.2.2 Misleading comments

Bad code

```
1 // Deletes a directory and checks if it is a directory.
2 // Throw IOException if not exists.
3 public static void deleteDirectory(File directory) throws IOException {
4     if (directory != null && directory.isDirectory() && !directory.exists()) {
5         return;
6     }
7
8     if (!directory.delete()) {
9         throw new IOException("Unable to delete directory " + directory + ".");
10    }
11 }
```

- This comment is redundant and misleading!
- This method does not throw `IOException` if the directory not exists. It throw `IOException` if the directory cannot be deleted.

3.2.3 Mandatory comments

Bad code

```

1 /**
2  * Returns the day of the year.
3 *
4 * @return the day of the year.
5 */
6 public int dayOfYear() {
7     return this.dayOfYear;
8 }
```

Bad code

```

1 /**
2 * @param title The title of the CD
3 * @param author The author of the CD
4 * @param tracks The number of tracks on the CD
5 * @param durationInMinutes The duration of the CD in minutes
6 */
7 public void addCD(String title, String author, int tracks, int durationInMinutes)
```

- This commentary provides no additional information, only obscures the code and brings the potential for misleading.
- It's bad to rule that every function of a Javadoc or every variable should have a comment.
- Such comments only make the code more confusing and lead to general confusion and disorder.

3.2.4 Diary comments

Bad code

```

1 // Changes history
2 //
3 // 2020-09-30 Implement FactoryAware to inject Factory
4 // 2020-09-30 Remove unnecessary stubbing
5 // 2020-09-30 Adapt to API changes
6 // 2020-09-30 Merge branch '2.3.x'
7 // 2020-09-30 Introduce a dedicated @Compatibility annotation
8 // 2020-09-30 Fix matching of SNAPSHOT artifacts when customizing layers
9 // 2020-09-30 Adapt to API change
10 // 2020-09-30 Merge branch '2.3.x'
11 // 2020-09-30 Start building against version 5.3.0
12 // 2020-09-30 Do not execute datasource initialization in a separate thread
13 // 2020-09-30 Start building against Java 16
14 // 2020-09-29 Reduce configuration resolution when building a layered jar
15 // 2020-09-29 Merge pull request #56701
16 // 2020-09-24 Polish
17 // 2020-09-17 Add support for Oracle
```

- A long time ago, there was a reason for the creation and maintenance of such log entries.
- Nowadays there are version control systems.
- These comments are disturbance data that spread confusion and should be removed.

3.2.5 Gossip

Bad code

```

1  /**
2   * Default constructor.
3   */
4  public Parser () {
5 }
```

Bad code

```

1  /** The day of the month. */
2  private int dayOfMonth;
```

Bad code

```

1  private void startSending() {
2      try {
3          send();
4      } catch (Exception e) {
5          try {
6              response.add(ErrorResponder.makeExceptionString(e));
7              response.closeAll();
8          } catch (Exception e1) {
9              // Who cares!?
10         }
11     }
12 }
```

- These comments are so garrulous that we learn to ignore them.
- When we read the code, our eyes just jump over it.
- At some point, the comments start lying when the surrounding code changes.

3.2.6 Position identifier

Bad code

```

1  // Methods /////////////////////////////////
```

Bad code, seen in a real project

```

1  ##########
2  //### getter and setter
3  ##########
```

3.2.7 Write-ups and incidental remarks

Bad code

```
1 // Added by Bad Programmer
```

- This information can be seen in the version management system.

3.2.8 Don't leave commented out code in your codebase

Bad code

```
1 public static String extension(String filename) {
2     // if (filename == null) { return null; }
3
4     String name = new File(filename).getName();
5     int extensionPosition = name.lastIndexOf('.');
6     if (extensionPosition < 0) {
7         return "";
8     }
9     return name.substring(extensionPosition + 1);
10 }
```

- Others who see this commented code will not have the courage to delete it.
- They will believe that there is a reason why the code is there, and that it is too important to be deleted.

3.2.9 Rules for commenting

Primary Rule

Comments are for things that **not** be expressed in code.

Redundancy Rule

Comments which **restate** code must be deleted.

Single Truth Rule

If the comment says what the code **could** say, then the code must change to make the comment redundant.



More information can be found here:

<https://agileinaflash.blogspot.com/2009/04/rules-for-commenting.html>

3.3 Classes and objects

According to *Clean Code* by Robert C. Martin there are some best practices help you write good classes, functions and variables which are easy to read and change.

This general recommendations applies to all:

- Use a descriptive name.
- Follow standard conventions.
- Keep It Simple and Stupid (KISS)
 - Reduce complexity as much as possible.
 - Keep it simple stupid. Simpler is always better.
- Follow the Principle Of Least Astonishment (POLA)
 - Be consistent.
 - If you do something a certain way, do all similar things in the same way.

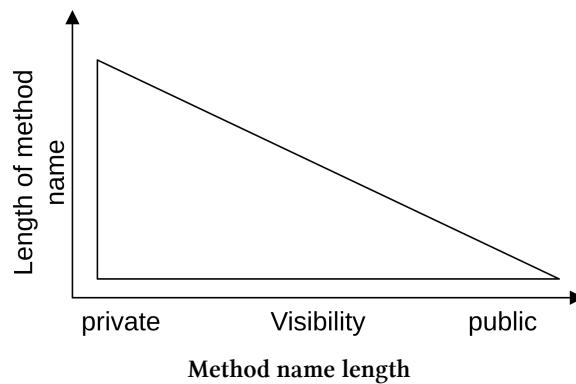
3.3.1 Classes

- Should have noun or noun phrase names.
- A class name should not be a verb.
- Hide internals and minimize the scope.
- Use dependency injection.
- Follow the Law of Demeter (LoD).
 - A class should know only its direct dependencies.
- Base class should know nothing about their derivatives.
- Prefer immutability, if possible.

3.3.2 Functions

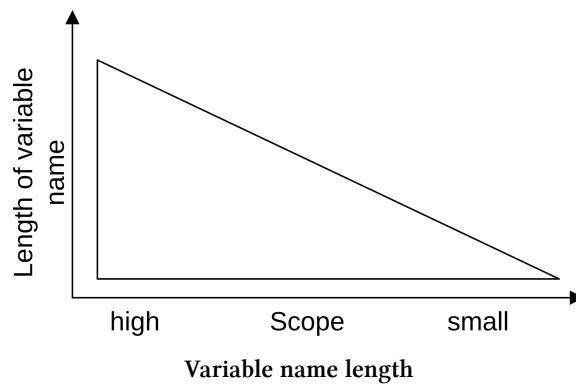
- Should have a verb or verb phrase name.
- Should be small.
 - The first rule of functions is that they should be small.
 - The second rule of functions is that they should be smaller than that.
- Do exactly one thing.
 - Functions should do one thing. They should do it well. They should do it only.
 - Follow the Single Responsibility Principle (SRP).
- Should use one level of abstraction per function.
- Don't Repeat Yourself (DRY)
- Have no side effects.
- Similar functions should be close.
- Place functions in the downward direction.
- Choose one consistent word per concept.
 - Avoid using different words to define the same concepts, like using *find*, *fetch*, *get*, *lookup*, *search* and *retrieve* for the same responsibility.

- Functions should have less than three arguments.
 - The ideal number of arguments for a function is zero. Next comes one, followed by two. Three arguments should be avoided where possible. More than three requires very special justification—and then shouldn't be used anyway.
 - Don't use flag arguments. Split method into several independent methods that can be called from the client without the flag.
- `private` function could be more descriptive and has longer names. Can be changed through Boy Scout Rule.
- As larger the visibility of the function, the shorter the name should be.



3.3.3 Variables

- Declare variables close to their usage.
- Follow the rule that the larger the scope of the variable, the longer the name. In this way, global variables get long descriptive names while scope-limited things like loop index variable can be as small as single letters.



3.4 Shapes of code

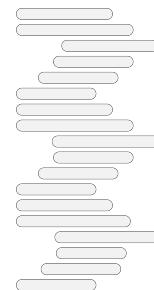
Who doesn't know it, you sit back, look at the code from a distance and recognize patterns. These patterns tell you a lot about the code you are looking at. The outlines, peaks, valleys and spaces between the lines of code can be used to detect certain antipatterns and improve the code.

So lean back, take a few steps back or set the screen resolution way too high and be excited about what you discover.

3.4.1 Spikes

Advantage for understanding:

Each spike can first be analysed on its own, even though they may be interdependent.



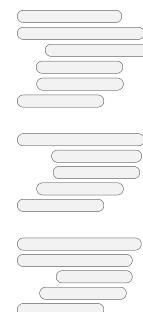
Advantage for refactoring:

Each spike is a potential candidate for transferring code into a separate private function and replacing it by calling that function.

3.4.2 Paragraphs

Advantage for understanding:

You know that the algorithm works in steps, and you know where the steps are in the code.



Advantage for refactoring:

Since steps should be somewhat distinct from each other, each step is a good candidate for outsourcing its code to a function. The resulting code would be a sequence of function calls. This would increase the level of abstraction and make the code more expressive.

3.4.3 Paragraphs with headers

Advantage for understanding:

As with paragraphs. The developer who wrote this has made your task easier by adding information to each step.

Advantage for refactoring:

As with paragraphs. You can use some terms in the comments as inspiration for function names. After refactoring, the comments become redundant and can be removed.



3.4.4 Suspicious comments

Advantage for understanding:

Not all comments are an advantage, and the code is often not a good one.

Advantage for refactoring:

Use the terms in the comments to rename the function and its parameters and remove the comments.



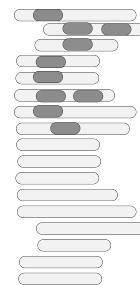
3.4.5 Intensive use of an object

Advantages for understanding:

The role of this code section is to set up this object.

Advantages for refactoring:

The function has several responsibilities, and one of them is to work with this object. Refactor this responsibility in a function to reduce the number of responsibilities (ideally to one) of the main function.



More information can be found here:

<https://agileinaflash.blogspot.com/2009/04/rules-for-commenting.html>

3.5 Avoid instantiation for utility classes

Java adds an implicit `public` constructor to every class which does not define at least one explicitly.

In the case of utility classes, which are a collection of static methods, there is no need for instantiation. Define a `private` constructor to permit instantiation. The explicit constructor is not visible outside the class. This class should be `final` as well, unless the private constructor is invoked by an inner class.

Bad code

```

1 public final class FileNameUtils {
2     public static String extension() {
3     }
4
5     public static String baseName() {
6     }
7 }
```

Good code

```

1 public final class FileNameUtils {
2
3     // Noninstantiable utility class
4     private FileNameUtils() {
5     }
6
7     public static String extension () {
8     }
9
10    public static String baseName() {
11    }
12 }
```

In cases you want guarantee that the class should not be invoked from within the class as well, you can throw an `AssertionError` in the constructor.³

Good code

```

1 public final class FileNameUtils {
2
3     // Noninstantiable utility class
4     private FileNameUtils() {
5         throw new AssertionError();
6     }
7     ...
8 }
```

Have a look into this classes and you will find a private constructor as well:

- `java.lang.Math`
- `java.util.Arrays`
- `java.util.Collections`

³Bloch, Joshua: Effective Java – Third Edition, Item 4

3.6 Use immutable objects

Always prefer immutable classes over mutable unless there's a good reason not to.

- Immutable objects are thread safe.
- Synchronized access can be avoided with immutable objects.
- Unchangeability always plays a role when objects are used together.
- Unchangeable objects can be safely passed to untrusted code.

Primitive types and `final`

The content of a variable declared as `final` cannot be changed. For primitive type variables, it means that the value of the variable does not change after initialization. So, a `final` variable may only be assigned once.

Final variables

```
1 final int max = 256;
2 max = 0; // error: does not compile
```

Reference variables and `final`

For reference variables, it means that the address stored in the reference variable does not change. This `final` reference variable refers to the object assigned to it at initialization and can never refer to another object. However, it does not mean that the referenced object is protected against changes.

Final reference variables

```
1 final Date deadline = new Date();
2 deadline = new Date(100,0,1,0,0,0); // error: does not compile
3 deadline.set(new Date(100,0,1,0,0,0)); // fine: compiles
```

Follow these five rules to make a class immutable⁴:

- Don't provide methods that modify the state, the state should not be changeable after creation.
- Prevent sub classing the class by making the class `final`.
- Make all fields final thus they are not changeable after constructor initialization.
- Make all fields private and decrease the visibility as far as possible.
- Ensure exclusive access to any mutable component, make defensive copies if required.

⁴Bloch, Joshua: Effective Java – Third Edition, Item 17

Good code

```

1  public final class ImmutableRGB {
2      private final int red, green, blue;
3
4      public ImmutableRGB(int red, int green, int blue) {
5          this.red = red;
6          this.green = green;
7          this.blue = blue;
8      }
9      public ImmutableRGB invert() {
10         return new ImmutableRGB(255 - this.red, 255 - this.green, 255 - this.blue);
11     }
12 }
```

Good code, but be aware of clone()

```

1  public final class Stamp {
2      private final Date date;
3      private final String author;
4
5      public Stamp(Date date, String author) {
6          this.date = (Date) date.clone();
7          this.author = author;
8      }
9
10     public Date date() {
11         return (Date) this.date.clone();
12     }
13
14     public String author() {
15         return this.author;
16     }
17 }
```

Be aware of using the `clone()` method to make defensive copies of an object. This can be used to compromise your code by extending in this case `Date` and overriding the `clone()` method. There malicious code can be placed by an attacker. This can be avoided, by creating an new object from `Date` and not using `clone()`.⁵

Good code, by creating new objects

```

1  public final class Stamp {
2      private final Date date;
3      private final String author;
4
5      public Stamp(Date date, String author) {
6          this.date = new Date(date.getTime());
7          this.author = author;
8      }
9
10     public Date date() {
11         return new Date(this.date.getTime());
12     }
13 }
```

⁵Long, Fred: Java Coding Guidelines, Item 10

```
13
14     public String author() {
15         return this.author;
16     }
17 }
```

Craftsman code

```
1 public final class Stamp {
2     private final LocalDate date; // LocalDate is immutable
3     private final String author;
4
5     public Stamp(LocalDate date, String author) {
6         this.date = date;
7         this.author = author;
8     }
9
10    public LocalDate date() {
11        return this.date;
12    }
13
14    public String author() {
15        return this.author;
16    }
17 }
```

This applies also to arrays, because arrays are objects. If a final variable holds a reference to an array, then the components of the array may be changed by operations on the array, but the variable will always refer to the same array.

Bad code

```
1 public static final String[] VALUES = {"Potential", "security", "hole!"};
```

Good code

```
1 private static final String[] PRIVATE_VALUES = {"Not", "potential", "security", "hole!"};
2 public static final List<String> VALUES = Collections.unmodifiableList(Arrays.asList(PRIVATE_VALUES));
```

Good code

```
1 private static final String[] PRIVATE_VALUES = {"Not", "potential", "security", "hole!"};
2
3 public static final String[] values() {
4     return PRIVATE_VALUES.clone();
5 }
```

Craftsman code

```
1 // Returns an immutable list containing one element.  
2 public static final List<String> VALUES = List.of("Not", "potential", "security", "hole!"); // Java 9
```

What can we expect⁶ from an immutable object?

Since the state of an immutable cannot change, we already know what to expect from it. We can expect the state of the object to be valid throughout the object's lifetime.

If implemented correctly, the state cannot be modified at any point in the code, possibly introducing inconsistencies. Therefore, an immutable object should validate the state in which it is constructed. This means that none can create an instance of an immutable object in an invalid state. All these validations should be performed within the immutable object. Therefore, we can expect not only that the immutable object will have the same state throughout its lifetime, but also a valid state.

⁶<https://reflectoring.io/java-immutables/>

3.7 Prefer records for immutability

Records are a new type of class added in Java 16 and are immutable. They are great for classes that only need to contain fields and access to those fields. Here is a simple record for an `ImmutableRGB` object.

Records

```
1 public record ImmutableRGB(int red, int green, int blue) {  
2 }
```

Instead of the keyword `class`, the keyword `record` has to be used. The fields are defined in the class declaration. This declaration creates a record with these fields. The fields are `final`, so there are no setter methods generated for records. The methods `equals`, `hashCode` and `toString` will be generated automatically. Furthermore access methods for the defined fields will be created. However, you are able to override anything or add additional methods, if you want.

Override in records

```
1 public record ImmutableRGB(int red, int green, int blue) {  
2  
3     ImmutableRGB(int red) {  
4         this(red, 0, 0);  
5     }  
6  
7     @Override  
8     public String toString() {  
9         StringBuilder builder = new StringBuilder();  
10        builder.append("ImmutableRGB [red=").  
11        .append(this.red)  
12        .append(", green=").  
13        .append(this.green)  
14        .append(", blue=").  
15        .append(this.blue)  
16        .append("]");  
17        return builder.toString();  
18    }  
19}
```

Records can be instantiated just like normal classes. For the `ImmutableRGB` record it would be:

Initializing records

```
1 ImmutableRGB rgb = new ImmutableRGB(0, 128, 240);  
2 rgb.red();
```

Records generate accessor methods, but they do not start with the prefix `get`. Instead, the accessor method name is the same as the field name defined in the constructor. Records can contain annotations and JavaDoc. Static fields, as well as constructors and instance methods can be declared in the body as well. Other instance fields outside the record header are not allowed.

Records are `final` and can not be `abstract`, they cannot inherit from other classes or records, so inheritance is not able. However, the implementation of interfaces is supported.

Records are only immutable if all fields defined in the constructor are immutable as well. If you have objects like `java.util.Date` you have provide copy or clone methods for this kind of fields as described in the item *Use immutable objects* before.

Records with a mutable field like Date

```
1 public record Stamp(Date date, String author) {  
2  
3     public Stamp(Date date, String author) {  
4         this.date = new Date(date.getTime());  
5         this.author = author;  
6     }  
7  
8     @Override  
9     public Date date() {  
10        return new Date(this.date.getTime());  
11    }  
12}
```

Records with all immutable fields

```
1 public record Stamp(LocalDate date, String author) {  
2 }
```

3.8 Provide immutable decorators for sensitive mutable classes

If you use third party library classes which are modifiable and you want provide immutability to them than you can decorate them with a immutable decorator class. This decorator can be used within your code base and be safe against modifications.

Principal - Third party code

```

1 public interface Principal {
2     String name();
3     Set<String> roles();
4 }
```

Mutable DefaultPrincipal - Third party code

```

1 public final class DefaultPrincipal implements Principal, Serializable {
2
3     private static final long serialVersionUID = 5439751315621189928L;
4
5     private final String name;
6     private final Set<String> roles;
7
8     public DefaultPrincipal(String name, Set<String> roles) {
9         this.name = name;
10        this.roles = roles;
11    }
12
13    @Override
14    public String name() {
15        return this.name;
16    }
17
18    @Override
19    public Set<String> roles() {
20        return this.roles;
21    }
22 }
```

ImmutablePrincipal

```

1 public final class ImmutablePrincipal implements Principal {
2
3     private final Principal principal;
4
5     public ImmutablePrincipal(Principal principal) {
6         this.principal = principal;
7     }
8
9     @Override
10    public String name() {
11        return this.principal.name();
12    }
13 }
```

```
13
14     @Override
15     public Set<String> roles() {
16         return Collections.unmodifiableSet(this.principal.roles());
17     }
18 }
```

The `ImmutablePrincipal` class decorates the third party implementation `DefaultPrincipal` mutable class and provide a safe `roles()` implementation. This implementation returns an unmodifiable view to the roles.

Client

```
1 public class Client {
2
3     public static void main(String[] args) {
4         Set<String> roles = new HashSet<>();
5         roles.add("ADMIN");
6         roles.add("DEVELOPER");
7
8         Principal principal = new ImmutablePrincipal(new DefaultPrincipal("user", roles));
9         principal.roles().add("SUPER_USER"); // throws java.lang.UnsupportedOperationException
10    }
11 }
```



Immutable decorators only provide you with the ability to be safe against changes and to protect the mutable classes when the client does not have the reference to the mutable class. Otherwise the client has the possibility to change them.

3.9 Avoid constant interfaces

According to Effective Java⁷:

- The constant interface pattern is a poor use of interfaces.
- That a class uses some constants internally is an implementation detail.
- Implementing a constant interface causes this implementation detail to leak into the class's exported API.
- If a nonfinal class implements a constant interface, all its subclasses will have their namespaces polluted by the constants in the interface.

Bad code

```
1 interface JobStatus {
2     int WAITING = 1;
3     int RUNNING = 2;
4     int FINISHED = 3;
5     int ERROR = 4;
6 }
```

Better code

```
1 enum JobStatus {
2
3     WAITING(1), RUNNING(2), FINISHED(3), ERROR(4);
4
5     private final int id;
6
7     private JobStatus(int id) {
8         this.id = id;
9     }
10
11    public int id() {
12        return this.id;
13    }
14 }
```

⁷Bloch, Joshua: Effective Java – Third Edition, Item 22

3.10 Avoid global constant classes

Global constants are problematic because they introduce largely unnecessary dependencies across the code and increase the coupling.

Furthermore, it is more difficult to extend the behaviour of the code, like the example below. Imagine you have to add support for a new unix line ending.

Bad code

```
1 public final class Constants {
2     private Constants(){}
3
4     public static final String CRLF = "\r\n"; // Windows
5 }
```

Bad code

```
1 public class Records {
2     public void write(Writer out) {
3         for (Record record: this.records) {
4             out.write(record.data());
5             out.write(Constants.CRLF);
6         }
7     }
8 }
```

Bad code

```
1 public class Rows {
2     public void print(PrintStream ps) {
3         for (Row row: this.rows) {
4             ps.print(row.cells());
5             ps.print(Constants.CRLF);
6         }
7     }
8 }
```

It is better to put the constant to a separate class which enables changeable behaviour. The constant is not distributed through the code as a public constant and enhances increasing cohesion and reducing coupling.

Good code

```
1 public class LineEnding {
2     private final String origin;
3
4     public LineEnding(String origin) {
5         this.origin = origin;
6     }
7
8     @Override
9     public String toString() {
10        return String.format("%s\r\n", this.origin); // Windows
11    }
12 }
```

Good code

```
1 public class Records {
2     public void write(Writer out) {
3         for (Record record: this.records) {
4             out.write(new LineEnding(record.data()));
5         }
6     }
7 }
```

Good code

```
1 public class Rows {
2     public void print(PrintStream ps) {
3         for (Row row: this.rows) {
4             ps.print(row.cells());
5             ps.print(Constants.CRLF);
6         }
7     }
8 }
```

Good code

```
1 public class LineEnding {
2
3     private static final String WINDOWS_LINE_ENDING = "\r\n";
4     private static final String UNIX_LINE_ENDING = "\n";
5
6     private final String origin;
7
8     public LineEnding(String origin) {
9         this.origin = origin;
10    }
11
12    @Override
13    public String toString() {
14        if (isUnix()) {
15            return String.format("%s%s", this.origin, UNIX_LINE_ENDING);
16        }
17
18        return String.format("%s%s", this.origin, WINDOWS_LINE_ENDING)
19    }
20 }
```

3.11 Favour composition over inheritance

According to Effective Java⁸:

- Inheritance is not always the best way to achieve code reuse. It can lead to inappropriately fragile software.
- For implementation inheritance, where a class extends another one, inheritance violates encapsulation, when a subclass depends on the implementation details of its superclass for its proper function.

Moreover, overriding methods should uphold the aspects of the superclass contract that relate to the Liskov Substitution Principle.

In this example `ZippedDocument` uses `length()` of `PlainDocument`, is it really intended or should `length()` be overwritten as well? You have to analyse your classes you inherit from in order to implement it in a proper way.

Bad code

```
1 interface Document {  
2     int length();  
3     byte[] content();  
4 }
```

Bad code

```
1 public class PlainDocument implements Document {  
2     private final byte[] content;  
3  
4     public PlainDocument(byte[] content) {  
5         this.content = content;  
6     }  
7  
8     @Override  
9     public int length() {  
10        return this.content.length;  
11    }  
12  
13     @Override  
14     public byte[] content() {  
15         return this.content;  
16     }  
17 }
```

⁸Bloch, Joshua: Effective Java – Third Edition, Item 18

Bad code

```
1 public class ZippedDocument extends PlainDocument {  
2  
3     public ZippedDocument(byte[] content) {  
4         super(content);  
5     }  
6  
7     // length() of PlainDocument is used!  
8  
9     @Override  
10    public byte[] content() {  
11        return zip(super.content());  
12    }  
13  
14    private byte[] zip(byte[] content) {  
15        return /* Zip the raw content */  
16    }  
17}
```

Instead of extending the `PlainDocument` class, you can solve some disadvantages using inheritance with composition.

Good code

```
1 public class ForwardingDocument implements Document {  
2  
3     private final Document document;  
4  
5     public ForwardingDocument(Document document) {  
6         this.document = document;  
7     }  
8  
9     @Override  
10    public int length() {  
11        return this.document.length();  
12    }  
13  
14     @Override  
15    public byte[] content() {  
16        return this.document.content();  
17    }  
18}
```

Good code

```
1 public final class ZippedDocument extends ForwardingDocument {  
2  
3     public ZippedDocument(Document document) {  
4         super(document);  
5     }  
6  
7     @Override  
8     public int length() {  
9         return this.content().length;  
10    }
```

```
11
12     @Override
13     public byte[] content() {
14         return zip(super.content());
15     }
16
17     private byte[] zip(byte[] content) {
18         return content; /* Zip the raw content */
19     }
20 }
```

Avoid building large inheritance hierarchies and prefer **has-a** relationships over **is-a** relationships. Achieve polymorphic behaviour and code reuse by composing objects. This does not mean, that you should never use inheritance! However, keep treating composition over inheritance as a option.

Disadvantages of inheritance:

- Creating good inheritance hierarchies is difficult.
- Can make the code harder to understand.
- Hierarchies are inflexible.
- A class can typically only inherit from a superclass.
- Usually, you cannot change at runtime from which superclass a class inherits.

Advantages of composition:

- Easier to understand than large inheritance hierarchies.
- Replacing a component with an alternative variation of this component is easier, even at runtime!
- The components can be tested in isolation.

3.12 Use the `@Override` Annotation

Annotate a method declaration if is intended to override a method declaration in a supertype. If a method is annotated with this annotation type, compilers generate an error message if the method does override or implement a method declared in a supertype.

Advantages:

- If you make any mistake such as wrong method name, wrong parameter types while overriding, you would get a compile time error, because you instruct the compiler that you are overriding this method.
- It improves the readability of the code. If you change the signature of overridden methods, then all subclasses that overrides the method would throw a compilation error.

Bad code

```
1 public class User {
2     private final String name;
3
4     public User(String name) {
5         this.name = name;
6     }
7
8     public int hashCode() { // Here is the bug!
9     }
10
11    public boolean equals(Object obj) {
12    }
13 }
```

Good code

```
1 public class User {
2     private final String name;
3
4     public User(String name) {
5         this.name = name;
6     }
7
8     @Override
9     public int hashCode() {
10    }
11
12     @Override
13     public boolean equals(Object obj) {
14    }
15 }
```

3.13 Use the `@FunctionalInterface` Annotation

Use `@FunctionalInterface` and force the compiler to break when an additional, non-overriding abstract method is added, which would break the use of lambda implementations.

- An informative annotation type used to indicate that an interface is intended to be a functional interface.
- Conceptually, a functional interface has exactly one abstract method, also called as Single Abstract Method (SAM).
- Instances of functional interfaces can be created with lambda expressions, method references, or constructor references.
- Should be used in order not to be accidentally extended and not fulfil the requirements of a function interface anymore.

Java is full of this type of interfaces, such as:

- `java.lang.Runnable`
- `java.util.Comparator`
- `java.util.concurrent.Callable`
- `java.io.FileFilter`

Good code - `Runnable` interface in Java

```
1  @FunctionalInterface
2  public interface Runnable {
3
4      * When an object implementing interface <code>Runnable</code> is used
5      * to create a thread, starting the thread causes the object's
6      * <code>run</code> method to be called in that separately executing
7      * thread.
8      *
9      * <p>
10     * The general contract of the method <code>run</code> is that it may
11     * take any action whatsoever.
12     *
13     * @see      java.lang.Thread#run()
14     */
15     public abstract void run();
```

3.14 Prefer returning Null-Objects over `null`

For methods that return a set of values using an array or collection, returning an empty array or collection is a better alternative to returning `null`, since most invokers are more capable of handling an empty object than a `null` value. If the client lacks the `null` check, a `NullPointerException` will be thrown at runtime.

- With Null-Objects you avoid returning `null` of a method.
- Reduce the amount of `NullPointerException`.
- Null-Objects are immutable.

There are already implementations for the Collection API:

- `Collections.EMPTY_LIST`, `EMPTY_MAP`, `EMPTY_SET`
- `Collections.emptyList()`, `emptyMap()`, `emptySet()`

The `Collections.EMPTY_LIST`, `EMPTY_MAP`, `EMPTY_SET` fields return raw types, whereas the newer `Collections.emptyList()`, `emptyMap()`, `emptySet()` methods return generic ones and should therefore be used instead.

When returning arrays instead of collections, avoid attempts to access individual elements of an array with length zero. This prevents an `ArrayOutOfBoundsException` from being thrown.

Bad code

```

1 public List<Book> findBooksByAuthor(String author) {
2     if (author != null || author.isEmpty()) {
3         return null;
4     }
5     ...
6 }
```

Good code

```

1 public List<Book> findBooksByAuthor(String author) {
2     if (author != null || author.isEmpty()) {
3         return Collections.emptyList();
4     }
5     ...
6 }
```

In the example below a custom Null-Object `NullUser` will be used in order not to force the client in this case `AdministrationController` for an NPE check.

Bad code

```
1 public class AdministrationController {
2     private Session session;
3
4     public boolean hasRights() {
5         User user = session.user();
6
7         return user != null && user.isSuperUser(); // NPE check required
8     }
9 }
```

Bad code

```
1 public class User {
2     private final int id;
3     private final boolean isSuperUser;
4
5     public User(int id, boolean isSuperUser) {
6         this.id = id;
7         this.isSuperUser = isSuperUser;
8     }
9
10    public int id() {
11        return this.id;
12    }
13
14    public boolean isSuperUser () {
15        return this.isSuperUser;
16    }
17 }
```

Good code

```
1 public class AdministrationController {
2     private Session session;
3
4     public boolean hasRights() {
5         return session.user().isSuperUser(); // NPE check not required
6     }
7 }
```

Good code

```
1 public class NullUser extends User {
2     @Override
3     public boolean isSuperUser () {
4         return false;
5     }
6 }
```

Good code

```
1 public class Session {
2     private User user;
3
4     public Session() {
5         this.user = new NullUser(); // Initialized with NullUser and not with null
6     }
7
8     public User user() {
9         return user;
10    }
11 }
```

3.15 Avoid `null` as method parameter

Never allow `null` as a method parameter. It is a bad practice to pass a `null` argument to methods as a valid argument.

Suppose you implement one method which retrieves a list of all files in the case of passing `null` and based on a file extension a list of files based on that.

It looks like a convenient alternative to these two methods:

- `public List<File> findAllFiles() {}`
- `public List<File> findByExtension(String extension) {}`

Putting this in one method and controlling the behaviour with a `null` parameter, is a poor approach as each method should do just one think. A preferred way would be splitting this one method in two methods with meaningful method names.

A more flexible solution would be using a filter like the solution bellow or better using a Java Predicate.

Bad code

```

1 public List<File> findFiles(String extension) {
2   if (extension == null) {
3         // find all files
4   } else {
5         // find files by file extension
6   }
7 }
```

Good code

```

1 interface FileFilter {
2   boolean accept(File file);
3 }
```

Good code

```

1 public class AllFileFilter implements FileFilter {
2   @Override
3   public boolean accept(File file) {
4     return true;
5   }
6 }
```

Good code

```
1 public List<File> findFiles(FileFilter filter) {  
2     List<File> foundFiles = new ArrayList<>();  
3  
4     for (File file: files) {  
5         if (filter.accept(file)) { // find files by filter  
6             foundFiles.add(file);  
7         }  
8     }  
9  
10    return foundFiles;  
11}
```

More generic craftsman code

```
1 public List<File> files(Predicate<File> filter) {  
2     return files.stream()  
3             .filter(filter)  
4             .toList();  
5 }
```

3.16 Prefer enhanced loops over for loops

For loops which only uses the index variable to access an element of the list or array, replace them with the foreach syntax. This is less prone to errors.

Bad code

```
1 List<String> cars = List.of("BMW", "VW", "Porsche");
2
3 for (Iterator<String> i = cars.iterator(); i.hasNext();) {
4     System.out.println(i.next());
5 }
```

Bad code

```
1 for (int i = 0; i < cars.size(); i++) {
2     System.out.println(cars.get(i));
3 }
```

Good code

```
1 for (String car : cars) {
2     System.out.println(car);
3 }
```

Good code

```
1 cars.forEach(System.out::println); // Lambdas
```

Good code

```
1 cars.stream().forEach(System.out::println); // Streams
```

3.17 Code against interfaces not implementations

Declarations should use interfaces rather than specific implementation classes. With this you can change the implementation of this interface easily if desired.

Bad code

```
1 // Bad - uses class as type
2 ArrayList<String> bikes = new ArrayList<>();
```

Good code

```
1 // Good - uses interface as type
2 List<String> bikes = new ArrayList<>();
```

Bad code

```
1 // Bad - uses class as type
2 public void buy(ArrayList<String> bikes) {
3 ...
4};
```

Good code

```
1 // Good - uses interface as type
2 public void buy(Collection<String> bikes) {
3 ...
4};
```

3.18 Use existing exceptions

The Java API offers many exception classes, so it is not necessary to declare a separate exception class for each case.

Exception	Description
IllegalArgumentException	Non-null parameter value is inappropriate
IllegalStateException	Object state is inappropriate for method invocation
NullPointerException	Parameter value is null where prohibited
IndexOutOfBoundsException	Index parameter value is out of range
ConcurrentModificationException	Concurrent modification of an object has been detected where it is prohibited
UnsupportedOperationException	Object does not support method

3.19 Validate method parameter

Validate method arguments to ensure that they are within the scope of the intended design of the method. This practice ensures that operations on the parameters of the method provide valid results.

Methods that accept arguments beyond a confidence boundary must perform a validation of their arguments. This precaution applies to all public methods of a class. Other methods, including private methods, should validate arguments that are both untrusted and not validated if these arguments from a public method can propagate through its arguments.

Good code

```
1 public void concat(String source, List<String> concats, int startIndex) {  
2     if (source == null || concats == null) {  
3         throw new IllegalArgumentException("source cannot be null");  
4     }  
5     if (startIndex >= concats.size() || startIndex < 0) {  
6         throw new IllegalArgumentException("startIndex exceeds bounds of concats");  
7     }  
8     ...  
9 }
```

- Objects.requireNonNull(obj)
- Objects.requireNonNull(obj, message)
- Objects.requireNonNull(obj, messageSupplier)
- Objects.requireNonNullElse(obj, defaultObj)
- Objects.requireNonNullElseGet(obj, supplier)
- Objects.checkIndex(index, length)
- Objects.checkFromIndexSize(fromIndex, size, length)
- Objects.checkFromToIndex(fromIndex, toIndex, length)

3.20 Prevent NullPointerException for String comparison

Place string literals on the left-hand side of an `equals()` or `equalsIgnoreCase()` method. This prevents NPE from being raised, as a string literal cannot be `null`.

Bad code

```
1 private static final String COMPARE_VALUE = "Bad programmer";
2
3 public boolean compareIt(String input) {
4     return input.equals(COMPARE_VALUE);
5 }
```

Good code

```
1 private static final String COMPARE_VALUE = "Rockstar programmer";
2
3 // equals has a check for NPE
4 public boolean compareIt(String input) {
5     return COMPARE_VALUE.equals(input);
6 }
```

Refactor step by step

```
1 String firstName = null;
2
3 boolean e1 = firstName.equals("Bad programmer"); // Raise a NPE
4 boolean e2 = firstName != null && firstName.equals("Better programmer"); // NPE check could be removed
5 boolean e3 = "Good programmer".equals(firstName); // No NPE
```

3.21 Safely cast long to int

Use overflow save methods defined in the `Math` class, which were added to Java 8. They either return a mathematically correct value or throws a `ArithmaticException` if the result overflows. Otherwise, mathematical operations which exceed the integer ranges provided by their primitive integer data types, will provide incorrect results.

According to the [Java Language Specification \(JLS\)](#)⁹:

- The integer operators do **not** indicate overflow or underflow in any way.
- An integer operator can throw an exception for the following reasons:
 - Any integer operator can throw a `NullPointerException` if unboxing conversion of a null reference is required.
 - The integer divide operator `/` and the integer remainder operator `%` can throw an `ArithmaticException` if the right-hand operand is zero.
 - The increment and decrement operators `++` and `--` can throw an `OutOfMemoryError` if boxing conversion is required and there is not enough memory available to perform the conversion.

Bad code

```
1 long foo = 10L;
2 int bar = (int) foo;
```

Good code

```
1 long foo = 10L;
2 int bar = Math.toIntExact(foo);
```

When you use `Math.toIntExact` the method returns the value of the `long` argument, throwing an `ArithmaticException` if the value overflows an `int`.

Math.toIntExact in Java 15

```
1 public static int toIntExact(long value) {
2     if ((int)value != value) {
3         throw new ArithmaticException("integer overflow");
4     }
5     return (int)value;
6 }
```

Several other methods exists int the `Math` class to be save against overflows:

- `Math.incrementExact(long)`
- `Math.subtractExact(long, long)`
- `Math.decrementExact(long)`
- `Math.negateExact(long)`
- `Math.subtractExact(int, int)`

⁹<https://docs.oracle.com/javase/specs/jls/se15/html/jls-4.html#jls-4.2.2>

3.22 Convert integers to floating point for floating-point math operations

Incorrect conversions between integers and floating-point values can lead to unexpected results. These unexpected results may include an overflow or other unusual behaviour. When arithmetic is performed on integers, the result will always be an integer. It discards information about possible fractional remainders.

Prevent precision losses when converting primitive integers into floating point values by assigning that result to a `long`, `double`, or `float` with automatic type conversion. If you have started with an `int` or `long`, the result will likely not what you expect.

Bad code

```
1 short a = 111;
2 int b = 1978;
3
4 float d = a / 8;    // 13.0
5 double e = b / 20; // 98.0
6
7 float f = 7 / 8;          // 0.0
8 long g = 1_000 * 3_600 * 24 * 365; // 1471228928
9 long h = Integer.MAX_VALUE + 1;      // -2147483648
10 long i = Integer.MIN_VALUE - 1;     // 2147483647
```

Good code

```
1 short a = 111;
2 int b = 1978;
3
4 float d = a / 8.0f;    // 13.875
5 double e = b / 20.0d; // 98.9
6
7 float f = 7 / 8.0f;          // 0.0875
8 long g = 1_000L * 3_600 * 24 * 365; // 31536000000
9 long h = Integer.MAX_VALUE + 1L;      // 2147483648
10 long i = Integer.MIN_VALUE - 1L;     // -2147483649
```

3.23 Use Generics in favour of raw types

Before Java 5, all code used raw types. Mixing generic typed code and raw typed code is allowed but should be avoided. Mixing allows developers to maintain compatibility between non-generic legacy code and newer generic code. Using raw types with generic code will cause the compiler to show *unchecked* warnings, but the code can still be compiled. If generic and non-generic types are used correctly together, these warnings can be ignored. However, these warnings can also indicate potentially unsafe operations.

If you have the code under your control, the use of generic types is recommended over the use of raw types. This forces type security and forces proper type verification.

Advantages using Generics:

- Stronger type-checking ensures type safety at compile time.
 - Error when setting an item of the wrong type.
 - The type of an element is known at removal.
- Avoidance of `ClassCastException` that might be thrown at runtime.
- Removal of casts makes the code clearer, which means you can use less code.
- Enabling implementing generic algorithms, that work on collections of different types.
- Self-documenting and easier to read.

Bad code

```
1 List cars = new ArrayList();
```

Good code

```
1 List<String> cars = new ArrayList<>();
```

Bad code

```
1 List ids = new ArrayList();
2 ids.add(new Integer(0));
3 Integer id = (Integer) ids.get(0); // possible ClassCastException at runtime
```

Good code

```
1 List<Integer> ids = new ArrayList<>();
2 ids.add(new Integer(0));
3 Integer id = ids.get(0);
```

3.24 Prefer enums over int constants

Prefer using enums over `public static final int` constants. This was the standard way to represent an enumerated type, before Java 5.

Disadvantages without enums:

- Not type safe
 - An int enum type can be passed in as any other int value method parameter even if this variant does not exist.
- No namespace
 - You must prefix the constants of an int enum with a character string like in this case GENRE, to avoid collisions with other int enum types.
 - The meaning of ints may be lost. The meaning of enumerations not because they are part of an enumeration that is named.
- Limited functionality
 - Enumerations can have methods and fields. Therefore, they are more powerful, meaningful and more flexible than ints.
- Printed values are uninformative
 - As these are only ints, when you print them out you will only get a number, which does not tell you what it stands for or even what kind of number it is.

Bad code

```

1 public static final int GENRE_ROMANTIC = 0;
2 public static final int GENRE_DRAMA = 1;
3 public static final int GENRE_COMEDY = 2;
4
5 public static final int FORMAT_AVI = 0;
6 public static final int FORMAT_FLV = 1;
7 public static final int FORMAT_MP4 = 2;
```

Bad code

```
1 public List<String> findFilmsByGenre(int genre) {}; // Not type safe
```

Good code

```

1 public enum Genre { ROMANTIC, DRAMA, COMEDY }
2 public enum Format { AVI, FLV, MP3 }
```

Good code

```
1 public List<String> findFilmsByGenre(Genre genre) {}; // Type safe
```

Enums are very flexible

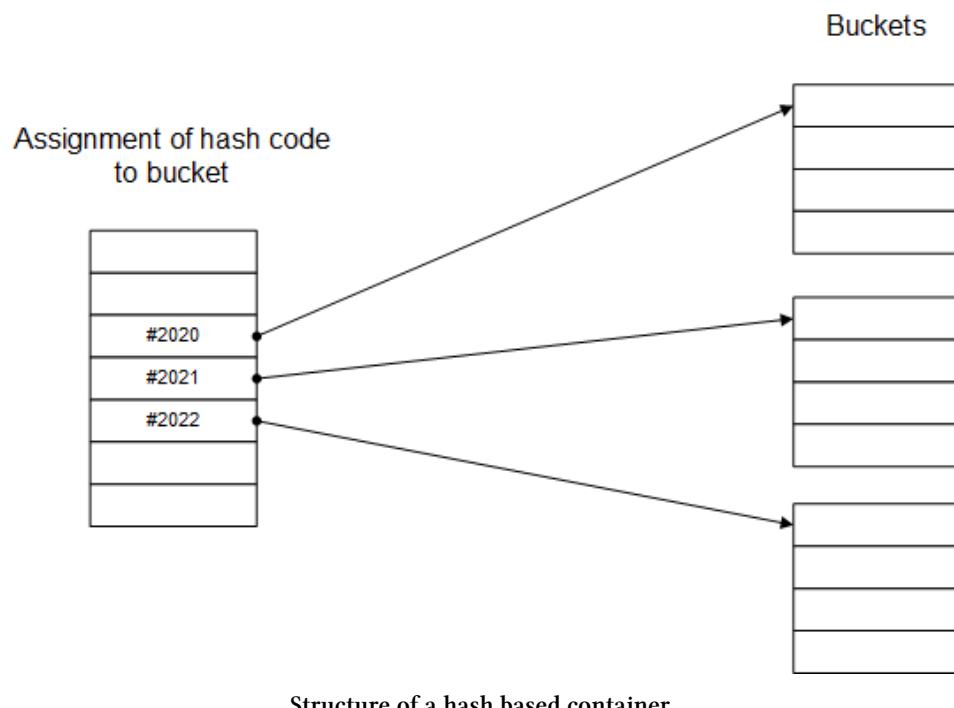
```
1  public enum JobPosition {
2
3      JUNIOR(2, 1.2f) {
4          @Override
5          public float pay(float hours) {
6              return (factor() * hours) * 0.8f;
7          }
8      },
9      SENIOR(3, 1.5f) {
10         @Override
11         public float pay(float hours) {
12             return (factor() * hours) * 0.95f;
13         }
14     },
15     PRINCIPAL(4, 1.7f) {
16         @Override
17         public float pay(float hours) {
18             return factor() * hours;
19         }
20     };
21
22     private final int level;
23     private final float factor;
24
25     JobPosition(int level, float factor) {
26         this.level = level;
27         this.factor = factor;
28     }
29
30     public int level() {
31         return this.level;
32     }
33
34     public float factor() {
35         return this.factor;
36     }
37
38     public abstract float pay(float hours);
39
40     public static JobPosition valueOf(int level) {
41         return Stream.of(JobPosition.values())
42             .filter(p -> p.level() == level)
43             .findFirst()
44             .orElseThrow(() ->
45                 new IllegalArgumentException(String.format("Unknown job position: %d", level)));
46     }
47 }
```

3.25 Be aware of the contract between `equals` and `hashCode`

Hash-based containers in Java:

- A hash-based container is organised in such a way that it creates different buckets in which the objects are stored sequentially.
- A bucket can be thought of as an array or list of object references.
- The access to the different buckets is done via an integral index and is therefore high performance; it is done in constant time, i.e., the access to the bucket always takes the same amount of time regardless of the number of elements in the container.
- Within a bucket, however, access to the elements is slow because it is sequential. The dependency here is linear, i.e., the larger the bucket, the longer it takes. Therefore, a hash-based container with many small buckets is cheaper than one with few large buckets.
- `hashCode` calculates a hash code for the object on which it is called.
- The hash code is an integral value that is used to store objects in a hash-based container or to find them in such a container.
- The hash-based containers in Java are `java.util.Hashtable`, `java.util.HashMap`, `java.util.HashSet` and their subclasses.

Structure of a hash based container:



Implementation of `equals` / `hashCode`:

- The default implementation of `hashCode` must be overwritten, if you do the same for `equals`.
- If two objects are equal according to the `equals` method, then calling the `hashCode` method on each of the two objects must produce the same result.
- It is not required that if two objects are unequal according to the `equals` method, the call to the `hashCode` method must return unique integer results for each of the two objects. However, the programmer should be aware that generating different integer results for unequal objects can improve hash table performance.
- If the same object is called more than once during execution, the `hashCode` method must consistently return the same integer, unless information used in `equals` comparisons on the object is changed.

Performance:

- The access to elements in a hash-based container is done by a quick identification of the bucket by means of index (= hash code) followed by a relatively slow sequential search within the hopefully small bucket. The advantage of hash-based containers is therefore the fast access to the bucket via index. If now the calculation of the hash code takes a long time, then the performance gain by the fast access via hash code is nullified. Therefore, hash code calculations should be as efficient as possible.
- The hash code calculation should therefore not only be fast, but also lead to a container with many small buckets.
- The goal is an implementation that is as fast as possible, which achieves an even distribution of the calculated hash codes in the interval of the possible integer values from -2147483648 to 2147483647.

Bad code

```

1  public final class Name {
2      private final String first;
3
4      public Name(String first) {
5          this.first = first;
6      }
7
8      @Override
9      public boolean equals(Object obj) {
10         if (this == obj) {
11             return true;
12         }
13         if (obj == null) {
14             return false;
15         }
16         if (getClass() != obj.getClass()) {
17             return false;
18         }
19         Name other = (Name) obj;
20         return Objects.equals(this.first, other.first);
21     }
22 }
```

Bad code

```
1 public static void main(String[] args) {
2     Set<Name> names = new HashSet<>();
3     names.add(new Name("Donald Duck"));
4     System.out.println(names.contains(new Name("Donald Duck"))); // prints false!!!
5 }
```

Good code

```
1 public final class Name {
2     private final String first;
3
4     public Name(String first) {
5         this.first = first;
6     }
7
8     @Override
9     public int hashCode() {
10        return Objects.hash(this.first);
11    }
12
13     @Override
14     public boolean equals(Object obj) {
15         if (this == obj) {
16             return true;
17         }
18         if (obj == null) {
19             return false;
20         }
21         if (getClass() != obj.getClass()) {
22             return false;
23         }
24         Name other = (Name) obj;
25         return Objects.equals(this.first, other.first);
26     }
27 }
```

3.26 Use text blocks for multi-line strings

A text block is a multi-line string literal that avoids the need for most escape sequences, automatically formats the string in a predictable way, and gives the developer control over format when desired.

You should use a text block when it improves the clarity of the code, particularly with multi-line strings.

Bad code

```
1 // ORIGINAL
2 String message = "'The time has come,' the Walrus said,\n" +
3     "'To talk of many things:\n" +
4     "Of shoes -- and ships -- and sealing-wax --\n" +
5     "Of cabbages -- and kings --\n" +
6     "And why the sea is boiling hot --\n" +
7     "And whether pigs have wings.'\n";
```

Good code

```
1 // BETTER
2 String message = """
3     'The time has come,' the Walrus said,
4     'To talk of many things:
5     Of shoes -- and ships -- and sealing-wax --
6     Of cabbages -- and kings --
7     And why the sea is boiling hot --
8     And whether pigs have wings.'
9 """;
```

Avoid aligning the opening and closing delimiters and the text block's left margin. This requires reindentation of the text block if the variable name or modifiers are changed.

Bad code

```
1 // ORIGINAL
2 String string = """
3     red
4     blue
5     """;
6
7 // ORIGINAL - after variable declaration changes
8 static String colors = """
9     red
10    blue
11    """;
```

Good code

```
1 // BETTER
2 String string = """
3     red
4     blue
5     """;
6
7 // BETTER - after variable declaration changes
8 static String colors = """
9     red
10    blue
11    """;
```

Either use only spaces or only tabs for the indentation of a text block. Mixing white space will lead to a result with irregular indentation.

Good code

```
1 // ORIGINAL
2     String colors = """
3     .....red
4     .....green
5     .....blue""";    // result: ".....red\ngreen\n.....blue"
6
7 // PROBABLY WHAT WAS INTENDED
8     String colors = """
9     .....red
10    .....green
11    .....blue""";    // result: "red\ngreen\nblue"
```



More information can be found here:
https://cr.openjdk.java.net/~jlaskey/Strings/TextBlocksGuide_v11.html#style-guidelines-for-text-blocks

3.27 Use always braces for the body of all statements

Use opening and closing braces for all statements like `if`, `for` and `while`. This should be done even if the body of the statement contains only one single line. You get better readability and are safe against misleading execution flows.¹⁰

Bad code

```
1 List<String> roles = new ArrayList<>();
2
3 if (isUser())
4     roles.add("USER");
5 else if(isAdmin())
6     roles.add("ADMIN");
```

This example above works as expected, but what if a developer extend the code as follows and do not add braces around the body. In this case the statement `roles.add("ADMIN")` will be always executed.

Bad code, not working as expected

```
1 List<String> roles = new ArrayList<>();
2
3 if (isUser())
4     roles.add("USER");
5 else if(isAdmin())
6     log.warn("Logged as admin!");
7     roles.add("ADMIN"); // Will be always executed
```

Good code

```
1 List<String> roles = new ArrayList<>();
2
3 if (isUser()) {
4     roles.add("USER");
5 } else if(isAdmin()) {
6     log.warn("Logged as admin!");
7     roles.add("ADMIN"); // Will be always executed
8 }
```

¹⁰Long, Fred: Java Coding Guidelines, Item 54

3.28 Pre calculate the length in loops

Bad code

```
1 // expensiveComputation() will be called n times
2 for (int i = 0; i < expensiveComputation(); i++) {
3     doSomething(i);
4 }
```

Good code

```
1 // expensiveComputation() will be called once
2 for (int i = 0, n = expensiveComputation(); i < n; i++) {
3     doSomething(i);
4 }
```

3.29 Avoid slow instantiation of String

Bad code

```
1 //slow instantiation
2 String slow = new String("Yet another string object");
```

Good code

```
1 //fast instantiation
2 String fast = "Yet another string object";
```

3.30 Use `StringBuilder` for concatenation

Strings are immutable, therefore concatenation doesn't simply append the new `String` to the end of the existing one. In each loop iteration the `String` is converted to an intermediate object type, the second `String` is appended, and then the intermediate object is converted back to a new `String`. This can lead to a poor performance as the number of iterations grows.

Better performance can be obtained by using a `StringBuilder` explicitly, which should be your default in almost all code. Try to avoid the `+` operator.

Bad code

```
1 public String concat() {
2     String result = "";
3
4     for (int i = 0; i < numItems(); i++) {
5         result += lineForItem(i); // String concatenation
6     }
7
8     return result;
9 }
```

Good code

```
1 public String concat() {
2     StringBuilder sb = new StringBuilder(numItems() * LINE_WIDTH);
3
4     for (int i = 0, n = numItems(); i < n; i++) {
5         sb.append(lineForItem(i));
6     }
7
8     return sb.toString();
9 }
```



`StringBuilder` is a drop-in replacement for `StringBuffer`, but with no guarantee of synchronization. Where possible, it is recommended using `StringBuilder` in preference to `StringBuffer` as it will be faster under most implementations.

3.31 Reduce lookups in collection containers

Reduce the number of lookups for an element within a collection if possible. In this example, an element is searched and in case of existsents the value will be used. The redundant operation can be reduced to one.

Bad code

```
1 Map<String, User> users = new HashMap<>();
2 ...
3
4 User user;
5
6 if (this.users.get(key) != null) { // call one
7     user = this.users.get(key);    // call two
8     String password = user.password();
9     ...
10 }
```

Good code

```
1 Map<String, User> users = new HashMap<>();
2 ...
3
4 User user;
5
6 if ((user = this.users.get(key)) != null) { // only one call
7     String password = user.password();
8     ...
9 }
```

3.32 Instantiate wrapper objects with `valueOf`

If a new instance is not required, this method should generally be used in preference to the constructor, as this method is likely to yield significantly better space and time performance by caching frequently requested values.

In the case of `Integer` and `Long` this method will always cache values in the range -128 to 127 inclusive and may cache other values outside of this range. The size of the cache can be controlled by the `-XX:AutoBoxCacheMax` option.

Bad code

```
1 new Boolean(true);
2 new Long(1);
3 new Integer(2);
```

Good code

```
1 Boolean.valueOf(true);
2 Long.valueOf(1);
3 Integer.valueOf(2);
```

Bad code

```
1 // 4 objects in memory
2 Boolean b1 = new Boolean(true);
3 Boolean b2 = new Boolean(false);
4 Boolean b3 = new Boolean(false);
5 Boolean b4 = new Boolean(false);
```

Good code

```
1 // 2 objects in memory
2 Boolean b1 = Boolean.TRUE;
3 Boolean b2 = Boolean.FALSE;
4 Boolean b3 = Boolean.FALSE;
5 Boolean b4 = Boolean.FALSE;
```

3.33 Use `entrySet` for iterating

When only the keys of a map are required in a loop, iterating the `keySet` makes sense. But when both the key and the value are needed, it's more efficient to iterate the `entrySet`, which will give access to both the key and value. The `Map.get(key)` lookup can be avoided.

Bad code

```
1 Map<String, String> table = new HashMap<>();
2
3 table.put("FirstKey", "FirstValue");
4 table.put("SecondKey", "SecondValue");
5
6 for (String key : table.keySet()) {
7     System.out.println(table.get(key));
8 }
```

Good code

```
1 Map<String, String> table = new HashMap<>();
2
3 table.put("FirstKey", "FirstValue");
4 table.put("SecondKey", "SecondValue");
5
6 for (Entry<String, String> entry: table.entrySet()) {
7     System.out.println(entry.getKey());
8     System.out.println(entry.getValue());
9 }
```

3.34 Use isEmpty() for String length

Bad code

```
1 if (text != null && !text.equals("")) {  
2     // actions  
3 }
```

Good code

```
1 if (text != null && !text.isEmpty()) {  
2     // actions  
3 }
```



More information can be found here:

<https://medium.com/javarevisited/micro-optimizations-in-java-string-equals-22be19fd8416>

3.35 Reduce the number of casts

Bad code

```
1 Integer io = new Integer(0);  
2 Object obj = (Object) io;  
3  
4 for (int i = 0; i < 100000; i++) {  
5     if (obj instanceof Integer) {  
6         byte x = ((Integer) obj).byteValue();  
7         double d = ((Integer) obj).doubleValue();  
8         float f = ((Integer) obj).floatValue();  
9     }  
10 }
```

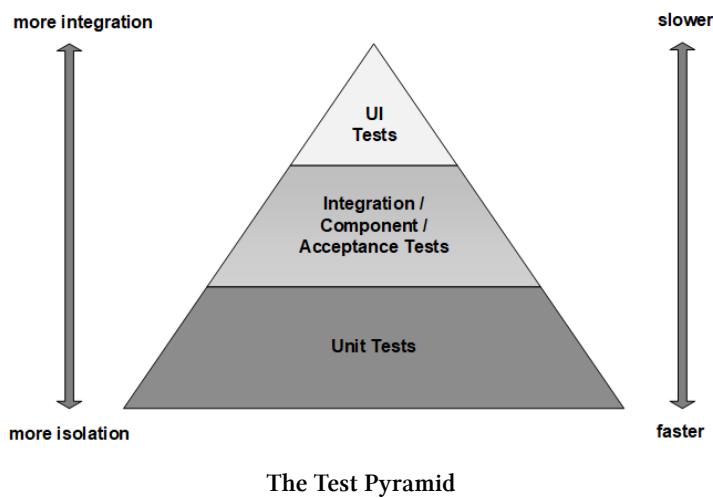
Good code

```
1 Integer io = new Integer(0);  
2 Object obj = (Object) io;  
3  
4 for (int i = 0; i < 100000; i++) {  
5     if (obj instanceof Integer) {  
6         Integer icast = (Integer) obj;  
7         byte x = icast.byteValue();  
8         double d = icast.doubleValue();  
9         float f = icast.floatValue();  
10    }  
11 }
```

4. Software Quality Assurance

4.1 Test Pyramid

Mike Cohn introduced the Test Pyramid in his book *Succeeding with Agile*. It is a great way to visualize different levels of testing. It also tells you how many tests to do on each level. The original test pyramid consists of three layers that your test suite should consist of *Unit Tests*, *Service Tests* and *User Interface Tests*.



According to this pyramid, write many small and fast unit tests. Write some more coarse-grained integration tests and very few high-level tests that test your application from top to bottom.

Be careful not to end up with a [Testing Pyramids & Ice-Cream Cones¹](#) that is a complete waste of time and maintenance.

Unit Tests

Unit tests focus on a single class and are the easiest, cheapest, and fastest to complete. Unit tests written at an early stage so that we get immediate feedback and know exactly where the bugs are.

Integration / Component / Acceptance Tests

Integration tests focus on the proper integration of different classes and modules. Compared to unit tests, they may require more specialised tools either for preparing the test environment or for interaction.

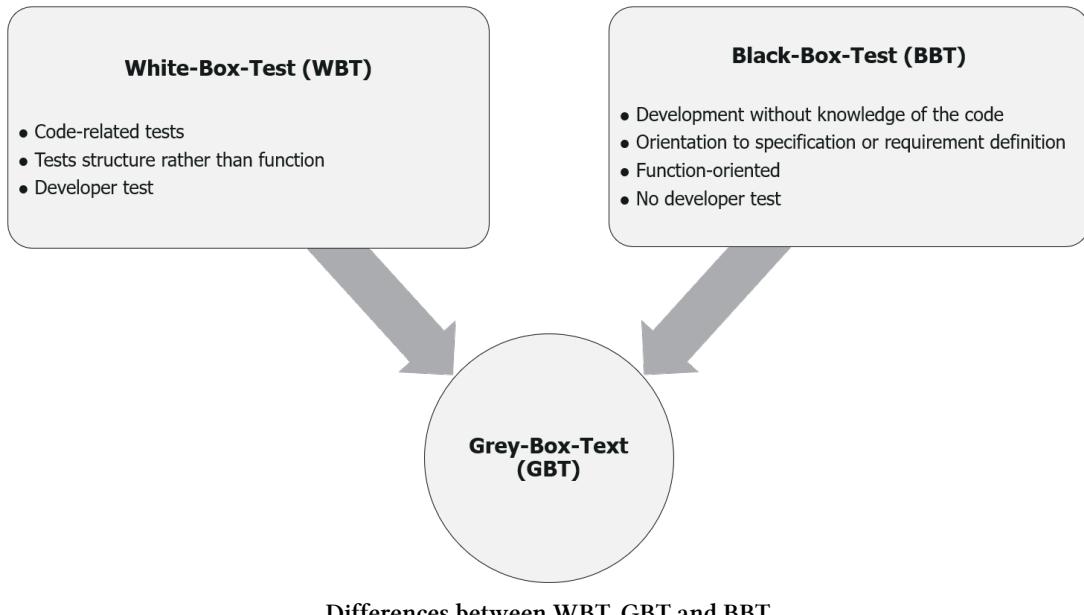
UI Tests

UI tests focus on the verification of a system from the client's point of view. The entire application is tested in a real scenario, such as communication with the database, network, hardware and other applications.

¹<https://alisterbscott.com/kb/testing-pyramids/>

4.2 Test Classification

Tests can be classification according to their information status. There are three main testing strategies like *White-Box-Test*, *Black-Box-Test* and *Grey-Box-Test*. WBT is performed by the development team after coding is complete by testing the internal behaviour of the code. BBT is performed by the professional test engineer by looking at the application. They do not have access to the logic and the flow of the code. GBT is the combination of WBT and BBT and is done without knowledge of the implementation.



Differences between WBT, GBT and BBT

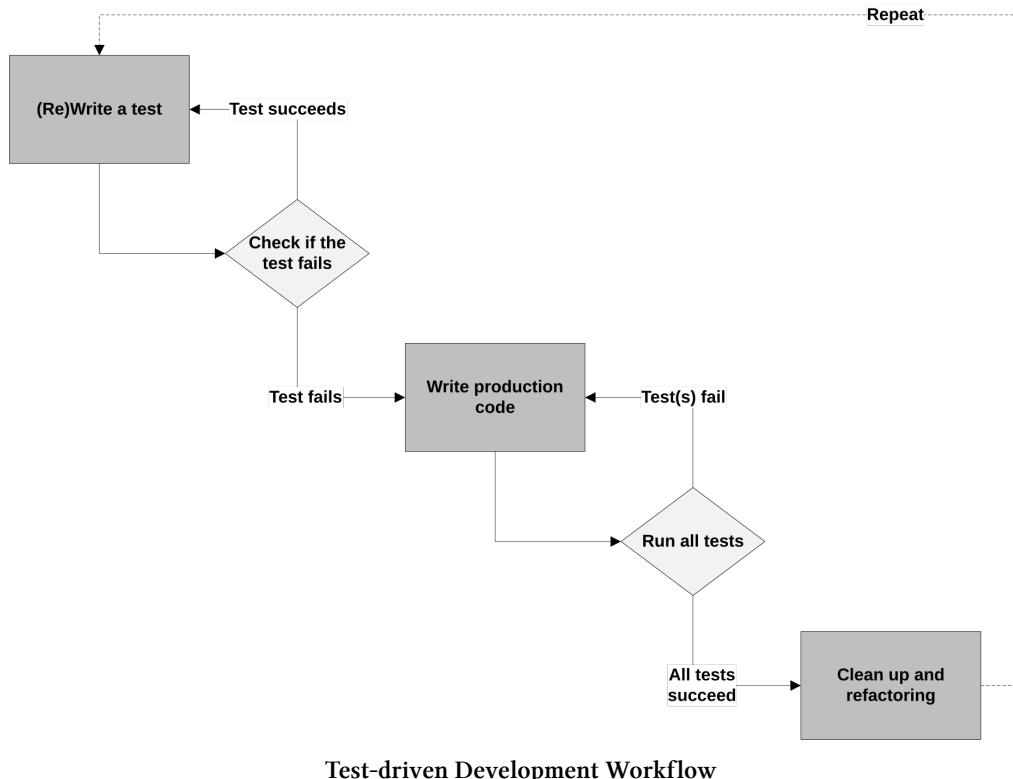
	WBT	BBT
Advantage	Testing of subcomponents and internal functionality Less organisational effort Automation through good tool support	Better verification of the overall system Testing of semantic properties with suitable specification Portability of systematically created test sequences to platform independent implementations
Disadvantage	Compliance with specification not verified Possible testing <i>around errors</i>	Greater organizational effort Additional functions added during implementation are only tested by chance Test sequences of an insufficient specification are unusable

4.3 Test-driven Development (TDD)

When we write a test... We are telling ourselves a story about how the operation will look from the outside – Kent Beck

What is test-driven Development?

- Test-driven programming
 - Motivate any behavioural change to the code through an automated test.
- Refactoring
 - Keep the code in *simple form* and develop the design step by step during the programming.
- Frequent integration
 - Integrate the code as often as necessary.



Why test-driven development?

- TDD ensures the quality and maintainability of software.
- Tests ensure that the existing functions are retained when extending and revising.
- Refactoring extends the productive life of a software.
- Code is often difficult to test afterwards.
- Test-First emphasizes the user view.
- Software development without testing is like climbing without rope and hooks.

Why Test-driven development improves the quality?

If you can't write a test for something you don't understand it

The practice of writing tests before implementation forces to deal more intensively with the problem to be solved and thus leads to a higher quality of the code, which is also more comprehensible for other developers. As a side-effect, TDD is also forced to deal with the question of what customers expect from the software to be written. TDD is therefore basically not a test method but a design method.

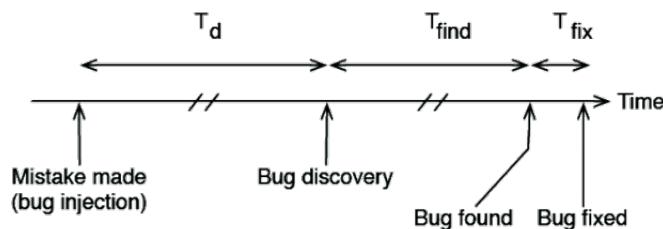
Without a regression test you can't clean the code

Without the possibility of quick testing, you often do not risk changing existing code. It is better to copy the code of a class, for example, into a new class, which is then extended with additional features. The result is an overloaded, confusing project with large amounts of superfluous code.

Fast feedback cycles save time and money

If you write the test first, the correctness of the implementation of a feature can be checked immediately. As a result, you always have certainty about when you have finished developing a feature.

The Physics of Debug Later Programming (DLP)



- As T_d increases, T_{find} increases dramatically
- T_{fix} is usually short, but can increase with T_d

Debug Later Programming

Without these rapid feedback cycles, you often get lost in the search for the causes of a bug. TDD helps to break down a big problem into smaller, manageable and testable problems.

You are going to test it anyway, spend the time to do it right

Regression tests are indispensable anyway. Without performing them regularly, testing large applications becomes almost impossible. Of course, regression tests can also be written afterwards, but then one does not benefit from the better design quality provided by TDD. Secondly, time tends to become limited as a deadline comes, so that subsequent testing often comes up short.

It makes my work more enjoyable

Every successful test gives you the feeling of having created something valuable: a piece of executable software that you can proudly show to other developers.

4.4 Unit testing with JUnit 5

4.4.1 Unit Tests

Benefits of unit tests:

- Changes to the code can be checked immediately
- Better code quality / stability or security
- Less hesitation before making changes to core components
- Shorter development times, despite additional tests to be implemented
- Easier refactoring
- Simplified documentation of the implementation

Requirements for unit tests:

- Easy implementation of the tests
- Fast and automated test execution
- Thorough analysis of the test results

4.4.2 JUnit 5

JUnit 5² is a simple Java unit testing framework helping testing on the JVM with a focus on Java 8 language features, extensibility, and a modern programming API.

What are the goals of JUnit 5?

- Reduce test creation effort to the absolute minimum
- Easy to learn and use
- Avoid redundant work
- Tests must
 - be repeatedly applicable
 - can be produced separately
 - be incremental
 - can be freely combined
 - be feasible also by others than the author
 - can also be evaluated by others than the author
- Enable the use of test data
 - Reusability of test data
 - Test data generation is usually more complex than the test itself

²<https://junit.org/junit5/>

What should we test?

The best thing about unit tests is that you can write them for all your production code classes, regardless of their functionality, complexity or internal structure. You can perform unit tests for controllers, services, repositories, domain classes or utility classes. Follow the rule of one test class per production class. This is a good start, but not a must. If you feel that you need to split your test into several test classes, then it's fine, do that.

A unit test class should test the public interface of the class. If you do this, all other methods will be tested automatically in an indirect way.

If the package structure of your test class is the same as the production class, `protected` and `package-private` methods are accessible from a test class and can be tested as well but testing these methods may be too far. Private methods cannot be tested anyway, because you cannot call them from another test class.

Your test suite should ensure that all your non-trivial code paths are tested. At the same time, they should not be too closely bound to your implementation.

How to structure tests?

All structure technics has the following steps in common:

- Setting up the test data
- Calling the method under test
- Assert that the expected results are returned

There are two common patterns, which can be applied to your test structure like **Arrange, Act, Assert³** and **Given, When, Then⁴**, which was developed as part of **Behavior-Driven Development⁵** (BDD).

What about project structure and naming conventions?

Nowadays all developers use standard tools for building their projects like **Maven⁶** or **Gradle⁷**. Thus, all production source code resides in `src/main/java` and all test source files are in `src/test/java`.

Typical project directory structure

```

1   src
2     └── main
3       ├── java
4       │   └── HelloWorld.java
5       └── test
6         └── java
7           └── HelloWorldTest.java

```

Test class names like above usually will be prefixed with `test`. This is a very common naming convention and should be followed consistently. It enables developer to understand at once which class is being tested. Furthermore, some tools relay on the convention.

³<https://xp123.com/articles/3a-arrange-act-assert/>

⁴<https://martinfowler.com/bliki/GivenWhenThen.html>

⁵<https://dannorth.net/introducing-bdd/>

⁶<https://maven.apache.org/>

⁷<https://gradle.org/>

4.4.3 First unit test

First simple JUnit 5 test

```
1 class SimpleTest {  
2  
3     private Collection<String> collection = new ArrayList<>();  
4  
5     @Test  
6     void testEmptyCollection() {  
7         assertTrue(collection.isEmpty());  
8     }  
9  
10    @Test  
11    void testOneItemCollection() {  
12        collection.add("First JUnit Test");  
13  
14        assertEquals(1, collection.size());  
15    }  
16 }
```

Execution order

```
1 setUp()  
2 testEmptyCollection()  
3  
4 setUp()  
5 testOneItemCollection()
```

Execution order⁸ of tests:

By default, test methods will be ordered using an algorithm that is deterministic but intentionally nonobvious. This ensures that subsequent runs of a test suite execute test methods in the same order, thereby allowing for repeatable builds. Even if unit tests should normally not depend on the order in which they are executed, there are times when it is necessary to enforce a certain order of execution of the test method. To specify the order in which test methods are executed, annotate your test class or test interface with `@TestMethodOrder` and provide the preferred `MethodOrder` implementation.



With JUnit 5 you can reduce the visibility of the test classes and methods. With JUnit 4 the classes and methods had to be `public`. In JUnit 5 test classes can have any visibility, it is recommended to use the default package visibility, which improves the test code base.

⁸<https://junit.org/junit5/docs/current/user-guide/#writing-tests-test-execution-order>

4.4.4 Assertions

Assertions⁹ is a collection of utility methods that support asserting conditions in tests. JUnit 5 contains many of the assertions of JUnit 4 and several interesting new assertions.

It also provides support for lambda expressions that can be used in assertions. The advantage of using lambda expressions for the assertion message is that it is evaluated lazily, which can reduce time and resources by avoiding the construction of complex messages.

The assertation result of a test provides specific value. A failed assertion will throw an `AssertionFailedError` or subclass thereof.

4.4.4.1 `assertEquals()` / `assertArrayEquals()`

Assert that expected and actual are equal.

Equality

```
1  @Test
2  void equality() {
3      Calculator calculator = new Calculator();
4
5      assertEquals(2, calculator.add(1, 1));
6      assertEquals(4, calculator.multiply(2, 2), "Optional failure message");
7
8      char[] expected = {'J', 'u', 'n', 'i', 't'};
9      char[] actual = "JUnit".toCharArray();
10
11     assertArrayEquals(expected, actual);
12 }
```

4.4.4.2 `assertSame()` / `assertNotSame()`

Assert that expected and actual refer to the same object.

Identity

```
1  @Test
2  void identity() {
3      User john = new User(1, "John", "Rambo");
4      User rocky = new User(2, "Rocky", "Balboa");
5
6      assertSame(john, john);
7      assertNotSame(john, rocky);
8 }
```

⁹<https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/Assertions.html>

4.4.4.3 `assertTrue()` / `assertFalse()` / `assertAll()`

Assert that the supplied condition is true or false. `assertAll()` assert that all supplied executables do not throw exceptions. It is a possibility to group assertions and have all the failed assertions reported together.

Truth

```

1  @Test
2  void truth() {
3      assertTrue('a' < 'b', () -> "Assertion messages can be lazily evaluated"
4          + "to avoid constructing complex messages unnecessarily.");
5  }
6
7  @Test
8  void groupedTruth() {
9      Person person = new Person("Jane", "Doe");
10
11     assertAll("person",
12         () -> assertEquals("Jane", person.firstName()),
13         () -> assertEquals("Doe", person.lastName())
14     );
15 }
```

4.4.4.4 `assertNull()` / `assertNotNull()`

Assert that actual is `null` or not.

Existence

```

1  @Test
2  void existence() {
3      assertNotNull(new Calculator());
4      assertNull(null);
5  }
```

4.4.4.5 `assertThrows()`

Assert that execution of the supplied executable throws an exception of the expected type and return the exception.

Exceptions

```

1  @Test
2  void exceptions() {
3      Calculator calculator = new Calculator();
4
5      Exception exception = assertThrows(ArithmaticException.class, () -> calculator.divide(1, 0));
6      assertEquals("/ by zero", exception.getMessage());
7  }
```

4.4.4.6 `assertTimeout()`

Assert that execution of the supplied executable completes before the given timeout is exceeded.

Timeout

```
1 @Test
2 void timeout() {
3     assertTimeout(Duration.ofSeconds(5), () -> TimeUnit.SECONDS.sleep(1));
4 }
```

4.4.4.7 `fail()`

Fail the test.

Failing

```
1 @Test
2 void failing() {
3     fail("a failing test");
4 }
```

4.4.5 Annotations

JUnit supports the following **annotations**¹⁰ for configuring tests:

@BeforeAll

Methods that are executed when the execution definition is started, before all tests.
Annotated method must be `static`.

@AfterAll

Methods that are executed after all tests, before closing the execution definition. For example, resources can be released here. Annotated method must be `static`.

@BeforeEach

Methods that are performed before each test.

@AfterEach

Methods that are performed after each test.

@Test

The actual test methods. Only methods with this annotation are executed as tests.

@Disabled

Temporary deactivation of test methods.

@Tag Used to declare tags for filtering, either at class or method level. Like **Categories**¹¹ in JUnit 4.

@DisplayName

Specification of user-defined names for test classes and methods.

@Timeout

Is used to fail a test if its execution exceeds a certain duration.

@Nested

Is used to signal that the annotated class is a nested. The nested test class must be an inner class, meaning a non-static class. The nested class can share the setup and state with an instance of its enclosing class. And, since inner classes cannot have static fields and methods, this prohibits the use of the `@BeforeAll` and `@AfterAll` annotations in nested tests.

@ParameterizedTest

Is used to signal that the annotated method is a parameterized test method. You have to pick up at least one source of arguments. There are several types of parameter sources you can pick from.

@TempDir

Can be used to annotate a non-private field in a test class or a parameter in a lifecycle method or test method of type `Path` or `File` that should be resolved into a temporary directory.

¹⁰<https://junit.org/junit5/docs/current/user-guide/#writing-tests-annotations>

¹¹<https://github.com/junit-team/junit4/wiki/Categories>

4.4.5.1 @Test

- `@Test`¹² marks the methods to be tested.
- The annotated methods must not be `private` or `static`.
- Must not return a value, thus the method should return `void`.
- Only with `@Test` annotated methods will be executed.

`@Test`

```
1 class TestAnnotationTest {  
2  
3     @Test  
4     void simple() {  
5         Collection<Object> collection = new ArrayList<>();  
6         assertTrue(collection.isEmpty());  
7     }  
8  
9     @Test  
10    void lambdaExpressions() {  
11        assertTrue(Stream.of(1, 2, 3)  
12                    .stream()  
13                    .mapToInt(i -> i)  
14                    .sum() > 5, () -> "Sum should be greater than 5");  
15    }  
16  
17    @Test  
18    void groupAssertions() {  
19        int[] numbers = {0, 1, 2, 3, 4};  
20  
21        assertEquals("numbers",  
22                      () -> assertEquals(numbers[0], 1),  
23                      () -> assertEquals(numbers[3], 3),  
24                      () -> assertEquals(numbers[4], 1)  
25                );  
26    }  
27 }
```

¹²<https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/Test.html>

4.4.5.2 @BeforeEach / @AfterEach

- `@BeforeEach`¹³ is used to specify that the annotated method should be executed before each test in the current test class.
- `@AfterEach`¹⁴ is used to specify that the annotated method should be executed after each test in the current test class.

@BeforeEach / @AfterEach

```

1  class BeforeAndAfterEachAnnotationsTest {
2      private Collection<String> collection;
3
4      @BeforeEach
5      void setUp() {
6          collection = new ArrayList<>();
7      }
8
9      @AfterEach
10     void tearDown() {
11         collection.clear();
12     }
13
14     @Test
15     void testEmptyCollection() {
16         assertTrue(collection.isEmpty());
17     }
18
19     @Test
20     void testOneItemCollection() {
21         collection.add("itemA");
22         assertEquals(1, collection.size());
23     }
24 }
```

Execution order

```

1  setUp()
2  testEmptyCollection()
3  tearDown()
4
5  setUp()
6  testOneItemCollection()
7  tearDown()
```

¹³<https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/BeforeEach.html>

¹⁴<https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/AfterEach.html>

4.4.5.3 @BeforeAll / @AfterAll

- `@BeforeAll`¹⁵ is used to specify that the annotated method should be executed before **all** tests in the current test class.
- `@AfterAll`¹⁶ is used to specify that the annotated method should be executed after **all** tests in the current test class.
- These annotated methods must be `static void` and are only executed **once** for a given test class.

@BeforeAll / @AfterAll

```

1  class BeforeAndAfterAllAnnotationsTest {
2      private Collection<String> collection;
3
4      @BeforeAll
5      static void oneTimeSetUp() {}
6      @AfterAll
7      static void oneTimeTearDown() {}
8
9      @BeforeEach
10     void setUp() {
11         collection = new ArrayList<>();
12     }
13     @AfterEach
14     void tearDown() {
15         collection.clear();
16     }
17
18     @Test
19     void testEmptyCollection() {
20         assertTrue(collection.isEmpty());
21     }
22     @Test
23     void testOneItemCollection() {
24         collection.add("itemA");
25         assertEquals(1, collection.size());
26     }
27 }
```

Execution order

```

1  oneTimeSetUp()
2
3  setUp()
4  testEmptyCollection()
5  tearDown()
6
7  setUp()
8  testOneItemCollection()
9  tearDown()
10
11 oneTimeTearDown()
```

¹⁵<https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/BeforeAll.html>

¹⁶<https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/AfterAll.html>

4.4.5.4 @Disabled

- `@Disabled`¹⁷ is used to specify that the annotated test class or test method is currently disabled and should not be executed.
- A reason can be declared to the annotation as well to provide more information why the test class or test method is disabled.
- Can be applied at the class level as well, which disable all test methods within that class.

Exclusion of test methods

```
1 class DisabledAnnotationTest {  
2  
3     @Disabled  
4     @Test  
5     void disabled() {  
6     }  
7 }
```

Exclusion of test methods with comment

```
1 class DisabledAnnotationTest {  
2  
3     @Disabled("not ready yet")  
4     @Test  
5     void disabledWithComment() {  
6     }  
7 }
```

Exclusion of test classes

```
1 @Disabled  
2 class DisabledAnnotationTest {  
3  
4     @Test  
5     void disabled() {  
6     }  
7  
8     @Test  
9     void disabledAsWell() {  
10    }  
11 }
```

¹⁷<https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/Disabled.html>

4.4.5.5 @DisplayName

- `@DisplayName`¹⁸ is used to declare a custom display name for the annotated test class or test method.
- Display names are used for reporting in IDEs and build tools and may contain spaces, special characters, and even emoji.

@DisplayName

```
1  @DisplayName("A special test case")
2  class DisplayNameAnnotationTest {
3
4      @Test
5      @DisplayName("Custom test name containing spaces")
6      void testWithDisplayNameContainingSpaces() {
7
8      }
9
10     @Test
11     @DisplayName("°□°")
12     void testWithDisplayNameContainingSpecialCharacters() {
13
14     }
15     @Test
16     @DisplayName("□")
17     void testWithDisplayNameContainingEmoji() {
18 }
```

4.4.5.6 @Tag

- `@Tag`¹⁹ marks test classes or test methods.
- These tags can later be used to filter the recognition and execution of tests.

@Tag

```
1  @Tag("fast")
2  @Tag("smoke-test")
3  class TagAnnotationTest {
4
5      @Test
6      @Tag("math")
7      void testingMathCalculation() {
8
9 }
```

¹⁸<https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/DisplayName.html>

¹⁹<https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/Tag.html>

4.4.5.7 @Timeout

- `@Timeout20` is used to define a timeout for a method or all testable methods within one class and its `@Nested` classes.
- Applying this annotation to a test class has the same effect as applying it to all testable methods.

`@Timeout`

```
1  class TimeoutAnnotationTest {  
2  
3      @BeforeEach  
4      @Timeout(5)  
5      void setUp() {  
6          // fails if execution time exceeds 5 seconds  
7      }  
8  
9      @Test  
10     @Timeout(value = 100, unit = TimeUnit.MILLISECONDS)  
11     void failsIfExecutionTimeExceeds100Milliseconds() {  
12         // fails if execution time exceeds 100 milliseconds  
13     }  
14 }  
15 }
```

²⁰<https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/Timeout.html>

4.4.6 Assumptions

Assumptions²¹ are used to perform tests only if certain conditions are met. This is typically used for external conditions that are necessary for the test to run properly. If the condition does not apply, the method is exited.

4.4.6.1 assumeFalse()

assumeFalse()

```
1 @Test
2 void falseAssumption() {
3     assumeFalse(5 < 1);
4
5     assertEquals(7, 5 + 2);
6 }
```

4.4.6.2 assumeTrue()

assumeTrue()

```
1 @Test
2 void trueAssumption() {
3     assumeTrue(5 > 1);
4
5     assertEquals(7, 5 + 2);
6 }
```

4.4.6.3 assumingThat()

assumingThat()

```
1 @Test
2 void assumptionThat() {
3     String message = "Just a assumptionThat test";
4
5     assumingThat(
6         "Just a assumptionThat test".equals(message),
7         () -> assertEquals(2 + 2, 4)
8     );
9 }
```

²¹<https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/Assumptions.html>

4.4.7 Parameterized Tests

Parameterised tests²² make it possible to run a test several times with different arguments. They are declared just like normal @Test methods but use the @ParameterizedTest annotation instead.

JUnit supports different sources as arguments:

- @ValueSource
- @EnumSource
- @MethodSource
- @CsvSource
- @CsvFileSource
- @ArgumentsSource
- @EnumSource
- @NullSource
- @EmptySource
- @NullAndEmptySource

4.4.7.1 @ValueSource

- @ValueSource²³ provides access to an array of values.
- Supported types are shorts, bytes, ints, longs, floats, doubles, chars, booleans, strings and classes.
- Only one of the supported types may be specified per @ValueSource declaration.

@ValueSource

```

1  @ParameterizedTest
2  @ValueSource(strings = { "otto", "level", "radar", "rotor", "kayak" })
3  void palindromes(String candidate) {
4      assertTrue(StringUtils.isPalindrome(candidate));
5  }
6
7  @ParameterizedTest
8  @ValueSource(ints = { 1, 2, 3 })
9  void testWithValueSource(int argument) {
10     assertTrue(argument > 0 && argument < 4);
11 }
```

²²<https://junit.org/junit5/docs/current/user-guide/#writing-tests-parameterized-tests>

²³<https://junit.org/junit5/docs/current/api/org.junit.jupiter.params/org/junit/jupiter/params/provider/ValueSource.html>

4.4.7.2 @MethodSource

- `@MethodSource`²⁴ provides access to values returned from factory methods of the class in which this annotation is declared or from static factory methods in external classes referenced by fully qualified method name.
- Each factory method must generate a stream of arguments.
- A stream is anything that JUnit can convert into a Stream, such as Stream, DoubleStream, LongStream, IntStream, Collection, Iterator, Iterable, an array of objects, or an array of primitives.
- Factory methods must be static and not declare any parameters.

`@MethodSource - Stream<String>`

```

1  @ParameterizedTest
2  @MethodSource("stringProvider")
3  void testWithExplicitLocalMethodSource(String argument) {
4      assertNotNull(argument);
5  }
6
7  static Stream<String> stringProvider() {
8      return Stream.of("JUnit 5", "rocks!");
9 }
```

`@MethodSource - IntStream`

```

1  @ParameterizedTest
2  @MethodSource("range")
3  void testWithRangeMethodSource(int argument) {
4      assertEquals(9, argument);
5  }
6
7  static IntStream range() {
8      return IntStream.range(0, 20).skip(10);
9 }
```

`@MethodSource - Stream<Arguments>`

```

1  @ParameterizedTest
2  @MethodSource("stringIntAndListProvider")
3  void testWithMultiArgMethodSource(String str, int num, List<String> list) {
4      assertEquals(5, str.length());
5      assertTrue(num >= 1 && num <= 2);
6      assertEquals(2, list.size());
7  }
8
9  static Stream<Arguments> stringIntAndListProvider() {
10     return Stream.of(
11         arguments("apple", 1, Arrays.asList("a", "b")),
12         arguments("lemon", 2, Arrays.asList("x", "y"))
13     );
14 }
```

²⁴<https://junit.org/junit5/docs/current/api/org.junit.jupiter.params/org/junit/jupiter/params/provider/MethodSource.html>

4.4.7.3 @CsvSource

- `@CsvSource`²⁵ reads comma-separated values (CSV) from one or more supplied CSV lines.

@CsvSource

```

1  @ParameterizedTest
2  @CsvSource({
3      "Bischofsmais,           1",
4      "Osternohe,             2",
5      "'Leogang, Austria', 0xF1",
6      "'Porte du Soleil, France',    4",
7  })
8  void testWithCsvSource(String bikepark, int rank) {
9      assertNotNull(bikepark);
10     assertEquals(0, rank);
11 }
```

4.4.7.4 @CsvFileSource

@CsvFileSource

```

1  @ParameterizedTest
2  @CsvFileSource(resources = "/two-column.csv", numLinesToSkip = 1)
3  void testWithCsvFileSource(String country, int reference) {
4      assertNotNull(country);
5      assertEquals(0, reference);
6 }
```

@CsvFileSource

```

1  #two-column.csv
2  Country, reference
3  Sweden, 1
4  Germany, 2
5  Australia, 3
6  "United States of America", 4
```

²⁵<https://junit.org/junit5/docs/current/api/org.junit.jupiter.params/org/junit/jupiter/params/provider/CsvSource.html>

4.4.7.5 @ArgumentsSource

@ArgumentsSource

```
1 @ParameterizedTest
2 @ArgumentsSource(BikeArgumentsProvider.class)
3 void testWithArgumentsSource(String argument) {
4     assertNotNull(argument);
5 }
```

ArgumentsProvider

```
1 public class BikeArgumentsProvider implements ArgumentsProvider {
2
3     @Override
4     public Stream<? extends Arguments> provideArguments(ExtensionContext context) {
5         return Stream.of("Specialized", "Cube").map(Arguments::of);
6     }
7 }
```

4.5 More on Unit Tests

4.5.1 Heuristics

Testing features, not methods

- Test cases are based on requirements and features, not on the methods to be tested.
- There is no 1:1 correspondence between test case method and tested method.
- Test cases should document how to use the classes correctly using examples.

Bad code

```
1 class EuroTest {  
2     @Test void newEuro() {...}  
3     @Test void getIntAmount() {...}  
4     @Test void getAmount() {...}  
5     @Test void plus() {...}  
6     @Test void minus() {...}  
7 }
```

Good code

```
1 class EuroTest {  
2     @Test void createFromInt() {...}  
3     @Test void createFromString() {...}  
4     @Test void rounding() {...}  
5     @Test void simpleAddition() {...}  
6 }
```

Testing at the edges

- Most algorithmic errors occur at the edges of the allowed value ranges.
- Can the value range be divided into several samples equivalent for the test? At least one sample per equivalence class!
- What is not tested may not work.

Implementation independence

- Tests are directed against the public class interface.
- Tests based on the innards of a class are extremely fragile.
- Wanting access to variables or private methods shows that the code still lacks a crucial design idea.

Orthogonal test cases

- Test cases are independent of each other if they refer to orthogonal aspects.

- Often a test can be extremely simplified by making assumptions that another test has already verified.
- If one gets into the trouble of having to adapt too many tests just to make a code change, the tests are not orthogonal.

Record results in the test

- Expected values are coded as constants, not calculated again in the test.
- If we reproduce application logic in the test, we also reproduce its errors.

Performance

Bad code

```

1 class EuroTest {
2
3     @Test
4     void multiYearInterest() {
5         double amount = 100.0;
6         double interest = 5.0;
7         double expectedInterest = amount * Math.pow((1 + interest / 100.0), 3.0);
8
9         assertEquals(expectedInterest, calculator.interest(amount, interest, 3), 0.001);
10    }
11 }
```

Good code

```

1 class EuroTest {
2
3     @Test
4     void multiYearInterest() {
5         double amount = 100.0;
6         double interest = 5.0;
7
8         assertEquals(115.76, calculator.interest(amount, interest, 3), 0.001);
9     }
10 }
```

Do not forget about exceptions and errors!

- Often the error cases are insufficiently tested. But exactly these are important to understand how the code behaves when it is not used correctly.

Remove redundancy in test code!

- The DRY Principle applies to test code as well.
- Redundancies should be avoided in order to reduce the impact on test code in the event of future changes to the production code.

Keep test cases short and understandable!

- Do not write test with big test methods.
- Split the test in multiple test methods.

Choose meaningful test case names!

- Naming is hard but try to communicate with the method test name what the test is doing.
- Be consistent and choose a naming approach which fits best.
- Test features not methods, this helps to divide tests in clean test cases with meaningful method names.

Do not trade test code as second citizen in your code base!

- Test code should be no less important than production code.
- Do not treat test code worse, treat it like production code and use the same coding standards.

Remove flaky tests!

- Since software tests serve as an early warning of potential regressions, they should always work reliably. A failed test should be cause for concern, and a broken build should immediately investigate why the test failed. It is a stop-the-world event.
- This approach can only work for tests that fail in a deterministic way. A test that sometimes fails and sometimes passes is unreliable and completely corrupts the entire test-suite. This can have negative effects on the team regarding tests.
- Developers no longer trust tests and soon ignore them. Even if non flaky tests fail, it is difficult to detect them in a number of broken tests. On the other hand, it is difficult to understand whether new failures are new or whether they come from existing flaky tests.

4.5.2 Naming of test methods

There are several ways to name the test methods. The most important thing is to be consistent and to define a convention within the team.

Approach 1:

Describe the facts of the test case.

Fact of the test as name

```
1 @Test void newAccount() {}
2 @Test void withdraw() {}
3 @Test void cannotWithdrawNegativeAmount() {}
4 @Test void cannotWithdrawUncoveredAmount() {}
```

Approach 2:

Describe the desired behaviour of the test case.

Desired behaviour as name

```
1 @Test newAccountShouldReturnCustomer() {}
2 @Test newAccountShouldHaveZeroBalance() {}
3 @Test withdrawShouldReduceBalanceByAmount() {}
4 @Test withdrawNegativeAmountShouldThrowException() {}
5 @Test withdrawNegativeAmountShouldNotChangeBalance() {}
```

4.5.3 Object Mother

An [Object Mother](#)²⁶ is a class which can be used in testing, which allows us to provide pre-configured objects for our tests. This example here is rather trivial but imagine this in a real application with classes having many attributes and/or compositions.

Most tests no longer instantiate objects themselves but go through a factory. In a future evolution, if a new mandatory attribute is added to a class, then all you must do is to modify the factory and all the tests will pass.

The combination of builders allows the objects generated by the Object Mother to be customised for the needs of the test. Otherwise, it would be necessary to multiply the factories for each need with a method taking in parameter all the necessary information for the tested case.

Classical approach

```
1 User user = new User("user", "password", "USER")
2 Authentication auth = new UsernamePasswordAuthenticationToken(user, null, user.authorities());
```

Object Mother Pattern

```
1 Authentication user = TestAuthentications.authenticatedUser();
2 Authentication admin = TestAuthentications.authenticatedAdmin();
3 Authentication tester = TestAuthentications.authenticated(new User("user", "password", "TESTER"));
```

Object Mother Pattern implementation

```
1 public final class TestAuthentications {
2
3     private static final User USER = new User("user", "password", "USER");
4     private static final User ADMIN = new User("admin", "password", "USER, ADMIN");
5
6     private TestAuthentications() {
7
8
9         public static Authentication authenticatedAdmin() {
10             return authenticated(ADMIN);
11         }
12
13         public static Authentication authenticatedUser() {
14             return authenticated(USER);
15         }
16
17         public static Authentication authenticated(User user) {
18             return new UsernamePasswordAuthenticationToken(user, null, user.authorities());
19         }
20     }
```

²⁶<https://martinfowler.com/bliki/ObjectMother.html>

4.5.4 Test Data Builder

The Object Mother and Builder patterns each bring their own set of advantages to our tests but can also be combined to further enhance the readability and maintainability of our tests. For complex object the Test Data Builder Pattern described in the book *Growing Object-Oriented Software, Guided by Tests* is a perfect solution for creating test objects.

Use of a Test Data Builder

```
1 Invoice anInvoice = new InvoiceBuilder().build();
```

Builder Pattern

```
1 public class InvoiceBuilder {
2
3     Recipient recipient = new RecipientBuilder().build();
4     InvoiceLines lines = new InvoiceLines(new InvoiceLineBuilder().build());
5     PoundsShillingsPence discount = PoundsShillingsPence.ZERO;
6
7     public InvoiceBuilder withRecipient(Recipient recipient) {
8         this.recipient = recipient;
9         return this;
10    }
11    public InvoiceBuilder withInvoiceLines(InvoiceLines lines) {
12        this.lines = lines;
13        return this;
14    }
15    public InvoiceBuilder withDiscount(PoundsShillingsPence discount) {
16        this.discount = discount;
17        return this;
18    }
19    public Invoice build() {
20        return new Invoice(recipient, lines, discount);
21    }
22 }
```

Combined Builder Pattern

```
1 Invoice invoiceWithNoPostcode = new InvoiceBuilder()
2     .withRecipient(new RecipientBuilder())
3     .withAddress(new AddressBuilder()
4         .withNoPostcode()
5         .build())
6     .build()
7     .build();
```

Combined Builder Pattern, shorter

```
1 Invoice invoice = new InvoiceBuilder()
2     .withRecipient(new RecipientBuilder().withAddress(new AddressBuilder().withNoPostcode())))
3     .build();
```

Combined Builder Pattern, even shorter

```
1 Invoice invoice = anInvoice()
2     .fromRecipient(aRecipient().withAddress(anAddress().withNoPostcode())))
3     .build();
```

Combined Builder Pattern, shortest

```
1 Invoice invoice = anInvoice()
2     .from(aRecipient().with(anAddress().withNoPostcode())))
3     .build();
```

4.5.5 F.I.R.S.T

Follow five rules that helps writing clean tests, which define the **F.I.R.S.T**²⁷ acronym.

Fast

- Tests should be fast.
- If tests are slow, they are run less frequently and therefore do not find problems early enough.

Independent

- Tests should not be interdependent but should be executable independently.
- A test should not set conditions for the next test and should be executable in any order.
- If tests are interdependent, the first one to fail triggers a cascade of further failed tests downstream.

Repeatable

- Tests should be repeatable in any environment.
- Tests should be executable in the production environment, in the QA environment and on the laptop on the train without a network.

Self-Validating

- Tests should have a boolean output.
- Either they are passed, or they fail.
- The manual study of log files should be avoided.
- If the tests do not validate themselves, failure can become subjective and a long manual evaluation may be required to execute the tests.

Timely

- Tests must be written in time.
- Unit tests should be written just before the production code that ensures that they are passed.
- If tests are developed after the production code, the production code may be difficult to test.

²⁷Robert, Martin: *Clean Code – A Handbook of Agile Software Craftsmanship*

4.6 Mocking with Mockito

Mockito²⁸ is a Java mocking framework for creating test double objects for unit tests of Java programs.

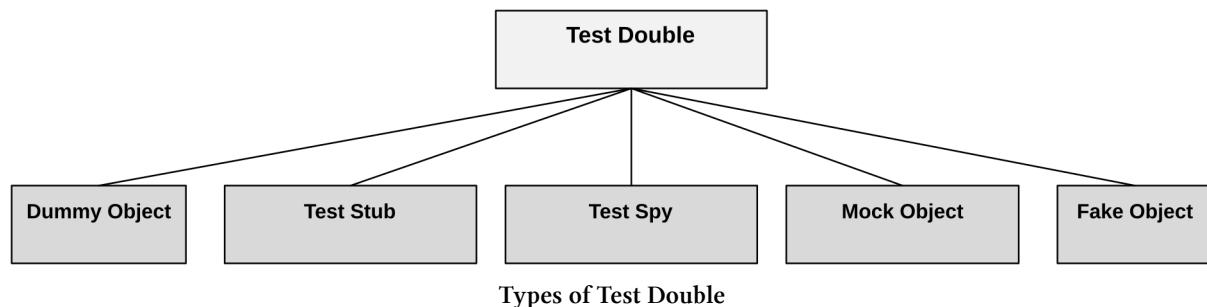
Why we need test double objects?

Most of the time the code we write depend on other dependencies. If we are using unit testing, often the code delegates some work to other methods in other classes or our test must depend on those methods. But we want the tests to be independent of all other dependencies.

Test double objects helps us to isolate a unit of code and test it alone. Replace DOCs (*Depended On Component*) and get control over the environment in which the SUT (*System Under Test*) is running and test the state and verify interactions.

4.6.1 Types of Test Double

Test Double is a generic term used by *Gerard Meszaros* in his book *xUnit Test Patterns* to mean any case where you replace a production object for testing purposes.



Test stub

used for providing the tested code with *indirect input*.

Mock object

used for verifying *indirect output* of the tested code, by first defining the expectations before the tested code is executed.

Test spy

used for verifying *indirect output* of the tested code, by asserting the expectations afterwards, without having defined the expectations before the tested code is executed. It helps in recording information about the indirect object created.

Fake object

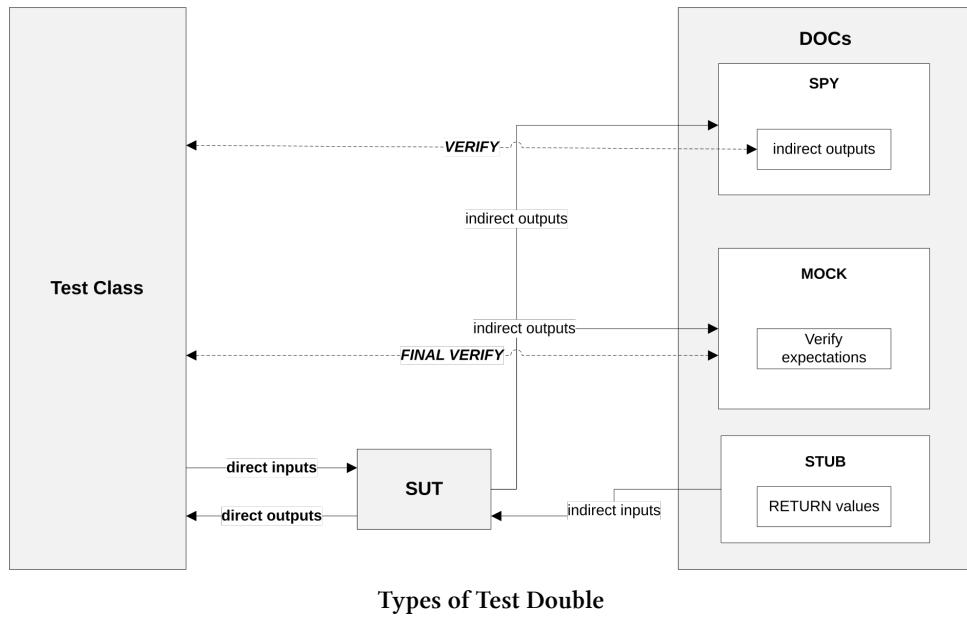
used as a simpler implementation, e.g. using an in-memory database in the tests instead of doing real database access.

Dummy object

used when a parameter is needed for the tested method but without needing to use the parameter.

²⁸<https://site.mockito.org/>

Dummies and stubs are used for **preparing the environment** for testing. Spies and mocks are for **verifying the correctness** of the communication.



There are [two styles²⁹](#) regarding mocking, the classical and the mockist TDD style. The classic TDD style is to use real objects when possible and a double when it is uncomfortable to use the real thing. However, a mocking TDD practitioner will always use a mock.

4.6.2 Activation

Activating Mockito can be done with JUnit 5 `@ExtendWith` annotation or programmatically, by invoking `MockitoAnnotations.openMocks()`.

`@ExtendWith`

```

1 @ExtendWith(MockitoExtension.class)
2 class MockitoAnnotationTest {
3     ...
4 }
```

`MockitoAnnotations.openMocks()`

```

1 class MockitoTest {
2
3     @BeforeEach
4     void init() {
5         MockitoAnnotations.openMocks(this);
6     }
7 }
```

²⁹<https://martinfowler.com/articles/mocksArentStubs.html#ClassicalAndMockistTesting>

4.6.3 Annotations

Mockito supports the following [annotations³⁰](#) for configuring mocked tests:

@Mock

Will create a new mock implementation for the given class. It will not create a real object.

@Spy

Create a wrapper around a real instance and spy on that real object.

@InjectMocks

Will inject the created mock to a given class instance.

@Captor

Is used to create an [ArgumentCaptor³¹](#) instance which is used to capture method argument values for further assertions.

4.6.3.1 @Mock

With `@Mock`, a mock instance of a class will be created.

- The instance is entirely instrumented to track interactions with it.
- This is not a real object and does not maintain the state changes.

Manually with `Mockito.mock()`

```

1  @Test
2  void whenNotUsingMockAnnotation() {
3      List mock = Mockito.mock(ArrayList.class);
4
5      mock.add("Java");
6      Mockito.verify(mock).add("Java");
7      assertEquals(0, mock.size());
8
9      Mockito.when(mock.size()).thenReturn(2020);
10     assertEquals(2020, mock.size());
11 }
```

@Mock Annotation

```

1  @Mock
2  List<String> mock;
3
4  @Test
5  void whenUsingMockAnnotation() {
6      mocked.add("Java");
7      Mockito.verify(mock).add("Java");
8      assertEquals(0, mock.size());
9
10     Mockito.when(mock.size()).thenReturn(2020);
11     assertEquals(2020, mockedList.size());
12 }
```

³⁰<https://javadoc.io/doc/org.mockito/mockito-core/latest/org/mockito/Mockito.html>

³¹<https://javadoc.io/doc/org.mockito/mockito-core/latest/org/mockito/ArgumentCaptor.html>

4.6.3.2 @Spy

With `@Spy`, a real instance of the class will be created.

- You can track every interaction with it.
- It maintains the state changes.

Manually with `Mockito.spy()`

```

1  @Test
2  void whenNotUsingSpyAnnotation() {
3      List<String> spy = Mockito.spy(new ArrayList<>());
4
5      spy.add("Java");
6      Mockito.verify(spy).add("Java");
7      assertEquals(1, spy.size());
8
9      Mockito.doReturn(2020).when(spy).size();
10     assertEquals(2020, spy.size());
11 }
```

`@Spy` Annotation

```

1  @Spy
2  List<String> spy = new ArrayList<>();
3
4  @Test
5  void whenUsingSpyAnnotation() {
6      spy.add("Java");
7      Mockito.verify(spy).add("Java");
8      assertEquals(2, spy.size());
9
10     Mockito.doReturn(2020).when(spy).size();
11     assertEquals(2020, spy.size());
12 }
```

4.6.3.3 @Captor

`@Captor`

```

1  @ExtendWith(MockitoExtension.class)
2  class MockitoCaptorTest {
3
4      @Mock
5      private List mock;
6
7      @Captor
8      private ArgumentCaptor captor;
9
10     @Test
11     void whenUseCaptorAnnotation() {
12         mock.add("Java");
13         Mockito.verify(mock).add(captor.capture());
14
15         assertEquals("Java", captor.getValue());
16     }
17 }
```

4.7 Code Coverage

Code coverage is a metric that measures at runtime which source code lines have been processed. The measurement is done by executing unit test cases and gives you a metric of the degree to which the source code has been tested.

This allows you to make a statement about how comprehensively the code has been tested. It improves the SW quality through better quality of the test cases and increases the efficiency of the test cases.

Test coverage should **never** be understood as metrics about the quality of the tests or the code! It is only an indicator for problem areas. Can therefore only make negative statements, not positive ones.

Measurement techniques:

Statement Coverage / Line Coverage

- Measures whether and how often a single line of code has been run through.
- Problem: If the line is a logical comparison, a single run is not representative.

Decision Coverage / Branch Coverage

- For case distinctions (if, while etc.) it is additionally checked that both cases (true and false) have occurred.

Path Coverage

- Path Coverage measures whether all possible combinations of programme flow paths have been run through.
- Problem: The number of possibilities increases exponentially with the number of decisions.

Function Coverage

- Measures whether they have been called based on the functions.

Race Coverage

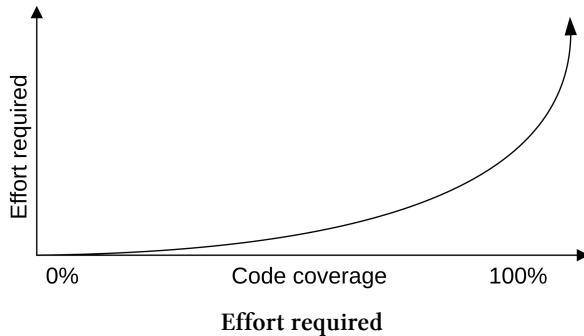
- Concentrates on code points that run in parallel.

Most code coverage tools on the market only support Statement and Decision Coverage, in some cases Path Coverage. Nevertheless, they already provide very valuable statements, e.g. code parts that were simply forgotten during testing and exceptions that have not been considered.

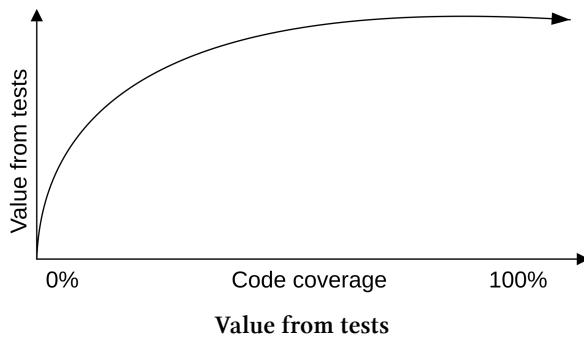
Challenges:

A 100% coverage is an illusion! Do not try to achieve 100% total code coverage. Achievable coverage depends strongly on the project type, whereas realistic values are in the range of 80%. Code coverage should be defined project-specifically and automatically checked during the build process.

Attaining 100% code coverage sounds good in theory but is almost always a waste of time. We will spend a lot of effort to get from 80% to 100%, as it is much more difficult than getting from 0% to 20%.



Increasing code coverage has lower revenues as well. Don't waste your time trying to achieve a very high level of test coverage, just to be better positioned with your project. Often stupid tests are written to artificially push them up. It is better to have tests that you can trust and test the features properly. Testing pure setters and getters make not always sense. These tests only cost time and money and do not really help the project.



Again, code coverage should not be used as a representation of the quality of a project!

Tooling:

EclEmma³²

Java based code coverage framework, with support of statement and decision coverage. Provide generation of text, HTML and XML reports and is part of Eclipse.

³²<http://www.eclemma.org/>

4.8 Static Code Analysis

Static code analysis is a static software test procedure. In contrast to dynamic test procedures, the software to be tested does not have to be executed.

Therefore, only the source code is needed. There is no need to run the system or a special test environment. It can be done by manual testing, but also automatically by a program. But should be anchored as early as possible in the development process.

Static code analysis is also used in compilers and IDEs for:

- Type check, e.g. *correct use of types*
- Style check, e.g. *use of code guidelines*
- Bug finding, e.g. *determining the use of uninitialized variables*

Tooling:

SonarQube³³

Platform for static code analysis of the technical quality. Supports not only Java but also the analysis of other programming languages. Analyses the code for different quality ranges and displays the results graphically.

Checkstyle³⁴

Checkstyle is an open source program that supports the compliance of coding conventions in Java programs.

SpotBugs³⁵

Open source program that uses static code analysis to check Java programs for numerous bugs. Is the successor of FindBugs.

PMD³⁶

PMD is a static source code analyser. It finds common programming flaws like unused variables, empty catch blocks, unnecessary object creation, and so forth. Include CPD, a copy-paste detector.

UCDetector³⁷

Unnecessary Code Detector searches for non-referenced methods and classes and give advice on the visibility of methods and classes.

³³<http://www.sonarqube.org/>

³⁴<http://checkstyle.sourceforge.net/>

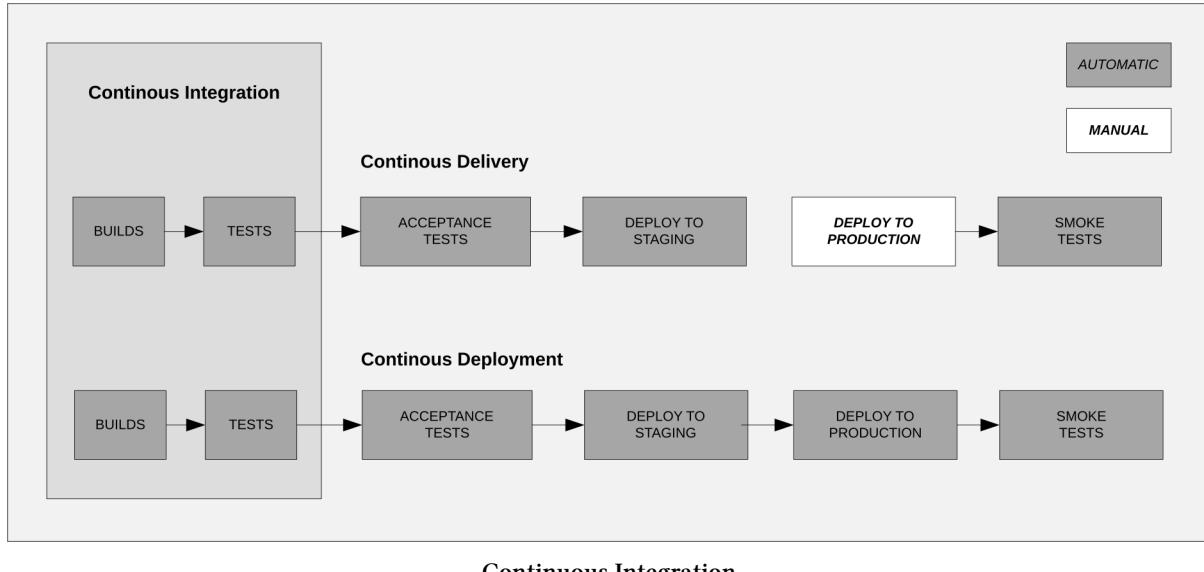
³⁵<https://spotbugs.github.io/>

³⁶<https://pmd.github.io/>

³⁷<http://www.ucdetector.org/>

4.9 Continuous Integration

Continuous integration describes a process of regular, complete rebuilding, testing and distribution of an application. There are cloud providers like [CircleCI³⁸](#) or [Travis³⁹](#) or local servers like [Jenkins⁴⁰](#), [GitLab CI⁴¹](#), [TeamCity⁴²](#) and [Bamboo⁴³](#).



Idea:

- Early and more frequent check-in of changes into the version management system.
- Replacement of large changes with incremental, functional small changes.
- When changes are checked in to the version management system, the entire system is rebuilt and automatically tested.
- Try to give the developer feedback on his changes as soon as possible.
- Improve quality by replacing the traditional approach to testing with continuous testing and rapid feedback.

Parts and steps:

- Compilation
- Test execution (Unit tests, Integration tests, Acceptance tests, etc.)
- Integration (Database, Third-party systems)
- Static analysis (Code & Architecture)
- Automatic deployment
- Generation of documentation

³⁸<https://circleci.com/>

³⁹<https://travis-ci.org/>

⁴⁰<https://www.jenkins.io/>

⁴¹<https://docs.gitlab.com/ee/ci/>

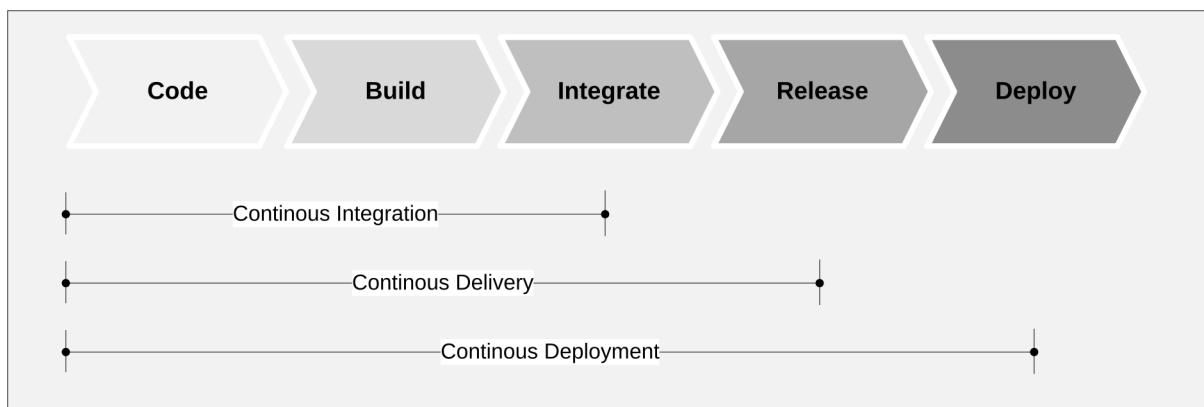
⁴²<https://www.jetbrains.com/teamcity/>

⁴³<https://www.atlassian.com/software/bamboo>

Addressed risks:

- Late troubleshooting
- Lack of team coordination
 - *Your changes don't match mine*
 - *Didn't you fix this two months ago?*
- Poor code quality
 - *Why do three different classes do the same thing?*
 - *The code of Team XY looks completely different*
- Lack of transparency / visibility
 - *What tests aren't running?*
 - *What does build 1.2.3 contain?*
 - *Where do we stand with code coverage?*
- Software not available
 - *I'm all right*
 - *It's working*
 - *I need another build to test*
 - *Tomorrow the boss is coming, we need a demo*

4.9.1 Differences between CI, CD and CD



Continuous Integration

- With CI, code changes are tested immediately after check-in, so that the newly included changes can be tested continuously.
- The basic idea is to test the code frequently during the development process in order to detect and fix potential problems early on.
- With CI, most of the work is done by automated tests and a build server.
- Since CI provides constant feedback on the software status, code problems that arise are usually less complex and much easier to solve.

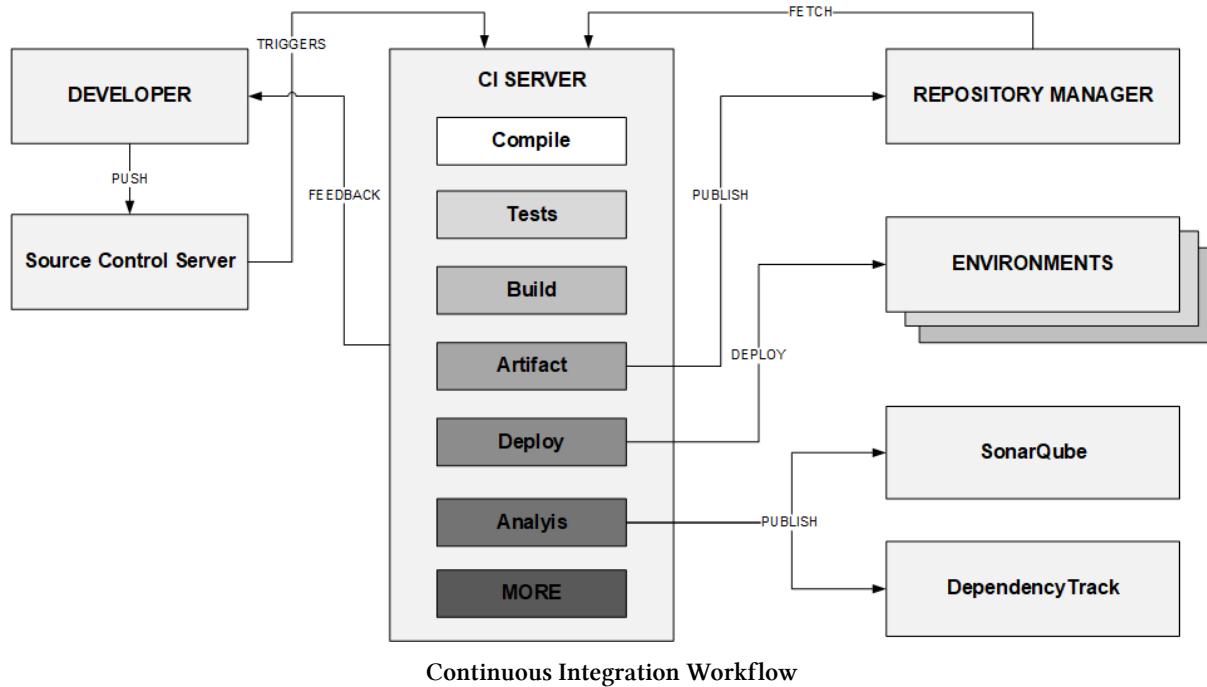
Continuous Delivery

- Continuous Delivery is an extension of the CI. It involves the Continuous Delivery of developed code to a staging environment as soon as the developer has developed it.
- It includes Continuous Integration, test automation and deployment automation processes that enable fast and reliable software development and deployment with minimal manual effort.
- The core idea is to deliver the code to QA, customers or any user base so that it can be continuously and regularly reviewed.
- This makes it possible to identify problems early in the development cycle and ensures that they can be resolved early.

Continuous Deployment

- Continuous Deployment is the next logical step after Continuous Delivery.
- It is the process of delivering code directly into production once it has been developed.
- With Continuous Deployment, all changes that pass the automated tests are automatically transferred to production.
- Successful implementation requires the automation of Continuous Integration, as well as the automation of Continuous Delivery into the staging environment. Finally, it also requires the ability to automatically deploy to production.

4.9.2 CI Workflow



Continuous Integration Workflow

In your CI workflow you can integrate many useful tools depending on what your goal is. In this case these following tools are part of the CI workflow.

Nexus⁴⁴

Manage binaries and build artifacts across your software supply chain.

SonarQube⁴⁵

Platform for static code analysis of the technical quality. Supports not only Java but also the analysis of other programming languages. Analyses the code for different quality ranges and displays the results graphically.

Renovate⁴⁶

Automated dependency updates for multi-platform and multi-languages.

DependencyTrack⁴⁷

Platform that allows organizations to identify and reduce risk from the use of third-party and open source components.

⁴⁴<https://www.sonatype.com/nexus>

⁴⁵<http://www.sonarqube.org/>

⁴⁶<https://github.com/renovatebot/renovate>

⁴⁷<https://dependencytrack.org/>

4.9.3 Preconditions

Common code base

- To be able to integrate meaningfully within a team, a version management system must exist in which all developers can continuously integrate their changes.

Automated compiling

- Every integration must pass through uniformly defined tests before the changes are integrated.
- This requires automated compiling.

Continuous test development

- Each change should be developed with an associated test if possible.
- Newly developed tests should become part of the automated test suite.
- Use code coverage to document and control test coverage.

Frequent integration

- Integrate changes into the common code base as often as possible.
- The risk of failed integrations is minimised with small increments.
- Developers work is often secured in the common code base.

Short test cycles

- Test cycle before integration should be short to encourage frequent integration.

Mirrored production environment

- The changes should be tested in a replica of the real production environment.

Easy access

- Even non-developers need easy access to the results of software development.

Automated reporting

- When was the last successful integration?
- What changes have been introduced since the last delivery?
- What is the quality of the version?

Automated distribution

4.9.4 Advantages and Disadvantages

Advantages:

- Integration problems are constantly being identified and resolved, not just before a milestone
- Early warnings for mismatched components
- Quick detection of errors through immediate execution of unit tests
- Constant availability of an executable stand for demo, test or distribution purposes
- Fast feedback on the check-in of incorrect or incomplete code
- Early and frequent testing, errors are found earlier
- Tests take place in parallel with development
- Correction of errors at the earliest, most favourable time
- Ability to react immediately to key metrics
- Review of *coding standards* and *best practices*

Disadvantages:

- Setting up the infrastructure
- Costs for hardware and software

4.9.5 Best Practices

- Use of a version management system
- Committing changes to a main branch should automatically trigger a build
- Check in early and often, prefer Trunk Based Development⁴⁸
- Do not check in a code that is not running
- Fix build errors immediately
- Tests should run as part of the CI process, with test failures causing a build failure
- Tackling problems early and failing quickly
- Code quality checks should run as part of the CI process
- Act and react based on metrics
- Create artifacts for each build
- Use configuration as code, the CI server configuration should be maintained as code
- Provide fast builds, so developer run these tests locally often

⁴⁸<https://trunkbaseddevelopment.com/>

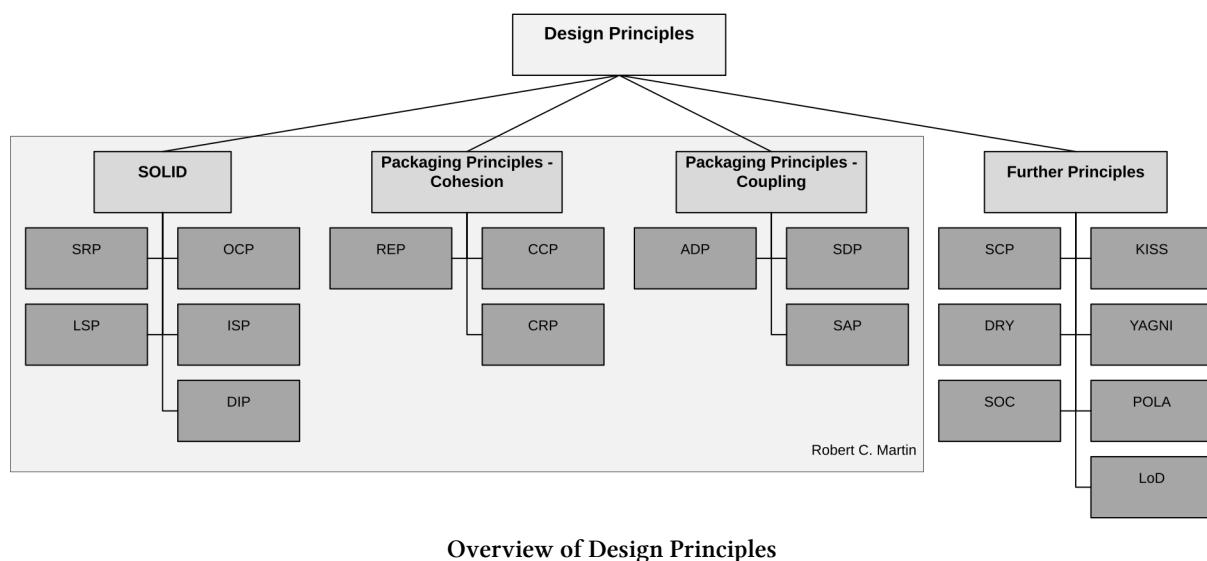
5. Design Principles

5.1 Goal of Design Principles

Code must be KISS, DRY and YAGNI!

- Design principles¹ should lead to good object-oriented design.
- Code and architecture smells are based on the violation of accepted design principles.
- Provide important information on how to fix code and architecture smells.
- If the violated design principle can be identified, it provides a first indication of what a better structure for the system might look like.
- Have been published and propagated by Robert C. Martin, Bertrand Meyer and Barbara Liskov, among others.

5.2 Overview of Design Principles



SOLID Principles

- The acronym **SOLID** was created by Robert C. Martin for a group of these principles.
- According to Robert C. Martin, applying these principles leads to a higher maintainability and thus to a longer service life of software.
- Deal with the question of how to combine classes into modules and packages.
- Lead to high cohesion within modules and low coupling between modules.

¹<http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod>

S	SRP	Single Responsibility Principle	<i>A class should have one, and only one, reason to change.</i>
O	OCP	Open Closed Principle	<i>You should be able to extend a classes behavior, without modifying it.</i>
L	LSP	Liskov Substitution Principle	<i>Derived classes must be substitutable for their base classes.</i>
I	ISP	Interface Segregation Principle	<i>Make fine grained interfaces that are client specific.</i>
D	DIP	Dependency Inversion Principle	<i>Depend on abstractions, not on concretions.</i>

SOLID Design Principles

Packaging Principles - Cohesion

REP	Release Reuse Equivalency Principle	<i>The granule of reuse is the granule of release.</i>
CCP	Common Closure Principle	<i>Classes that change together are packaged together.</i>
CRP	Common Reuse Principle	<i>Classes that are used together are packaged together.</i>

Packaging Principles - Cohesion

Packaging Principles - Coupling

ADP	Acyclic Dependencies Principle	<i>The dependency graph of packages must have no cycles.</i>
SDP	Stable Dependencies Principle	<i>Depend in the direction of stability.</i>
SAP	Stable Abstractions Principles	<i>Abstractness increases with stability.</i>

Packaging Principles - Coupling

Further Design Principles

SCP	Speaking Code Principle	<i>Writing code that speaks for itself and that does not need a comment.</i>
KISS	Keep It Simple (and) Stupid	<i>Simplicity should be a key goal in design, and unnecessary complexity should be avoided.</i>
DRY	Don't Repeat Yourself	<i>Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.</i>
YAGNI	You Ain't Gonna Need It!	<i>Always implement things when you actually need them, never when you just foresee that you need them.</i>
SOC	Separation Of Concerns	<i>Process of separating a program into distinct features that overlap in functionality as little as possible.</i>
POLA	Principle Of Least Astonishment	<i>If something has a hight astonishment factor, it may be necessary to redesign it.</i>
LoD	Law of Demeter	<i>Only talk to your immediate friends.</i>

Further Design Principles

5.3 SOLID Principles

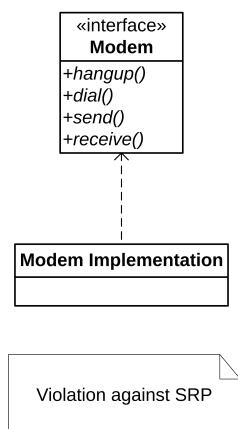
5.3.1 Single Responsibility Principle

A class should have one, and only one, reason to change.

- Each class has only one defined task to fulfil.
- A class should only have functions that directly contribute to the fulfilment of this task.
- One of the easiest ways to write classes that never need to be changed is to write classes that serve only one purpose.
- Each class should implement a coherent set of related functions.
- Having multiple responsibilities in a class leads to tight coupling and results in fragile designs.
- Following the principle of single responsibility is to ask yourself again and again whether each method and process of a class is directly related to the name of that class.
- For methods that do not match the name of the class, move these methods to other classes.

5.3.1.1 Example: Modem

Noncompliant Code Example

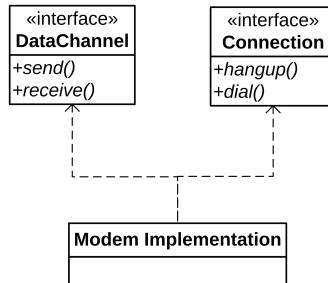


Bad code

```

1 public interface Modem {
2     // Connection management responsibilities
3     void dial(String phoneNumber);
4     void hangup();
5
6     // Data management responsibilities
7     void send(char c);
8     char receive();
9 }
  
```

Compliant Solution



No violation against SRP

Good code

```

1 public interface DataChannel {
2     void send(char c);
3     char receive();
4 }
  
```

Good code

```

1 public interface Connection {
2     void dial(String phoneNumber);
3     void hangup();
4 }
  
```

Good code

```

1 public class Modem implements Connection, DataChannel {
2     private final Connection connection;
3     private final DataChannel dataChannel;
4
5     public Modem(Connection connection, DataChannel dataChannel) {
6         this.connection = connection;
7         this.dataChannel = dataChannel;
8     }
9
10    @Override
11    public void dial(String phoneNumber) {
12        this.connection.dial(phoneNumber);
13    }
14
15    @Override
16    public void hangup() {
17        this.connection.hangup();
18    }
19
20    @Override
21    public void send(char c) {
22        this.dataChannel.send(c);
23    }
  
```

```
24  
25     @Override  
26     public char receive() {  
27         return this.dataChannel.receive();  
28     }  
29 }
```

5.3.1.2 Example: Book

Noncompliant Code Example

Bad code

```
1 public final class Book {
2
3     private final String title;
4     private final String author;
5     private final String content;
6
7     public Book(String title, String author, String content) {
8         this.title = title;
9         this.author = author;
10        this.content = content;
11    }
12
13    public String title() {
14        return this.title;
15    }
16
17    public String author() {
18        return this.author;
19    }
20
21    public String content() {
22        return this.content;
23    }
24
25    public void exportToPdf() {
26        // logic
27    }
28
29    public void exportToEpub() {
30        // logic
31    }
32 }
```

Compliant Solution

Good code

```
1 public final class Book {
2
3     private final String title;
4     private final String author;
5     private final String content;
6
7     public Book(String title, String author, String content) {
8         this.title = title;
9         this.author = author;
10        this.content = content;
11    }
12
13    public String title() {
14        return this.title;
```

```
15     }
16
17     public String author() {
18         return this.author;
19     }
20
21     public String content() {
22         return this.content;
23     }
24 }
```

Good code

```
1 public interface BookExporter {
2     void export(Book book);
3 }
```

Good code

```
1 public class PdfExporter implements BookExporter {
2
3     @Override
4     public void export(Book book) {
5         // logic
6     }
7 }
```

Good code

```
1 public class EpubExporter implements BookExporter {
2
3     @Override
4     public void export(Book book) {
5         // logic
6     }
7 }
```

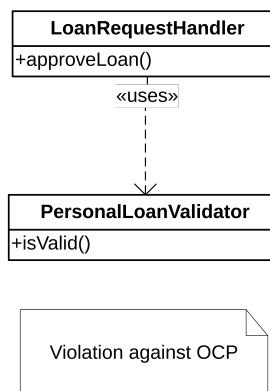
5.3.2 Open Closed Principle

You should be able to extend a classes behavior, without modifying it.

- Classes that follow the OCP have two essential characteristics:
 - **Open to extensions**, the behaviour of such class can be extended to meet new requirements of an existing or even new application.
 - **Closed against subsequent modification**, no longer need to be modified to serve as a basis for new requirements.
- Functions that perform different actions due to type switches are good examples of OCP violation. Such functions are never closed against changes, because adding a new type requires changing the source code of the function.
- It is more elegant and robust to extend a class group by adding a class than to modify the existing source code.
- By using e.g. abstract base classes, software components can be created which have a fixed unchangeable implementation, but whose behaviour can be changed indefinitely by inheritance and polymorphism.

5.3.2.1 Example: [LoanRequestHandler](#)

Noncompliant Code Example



Bad code

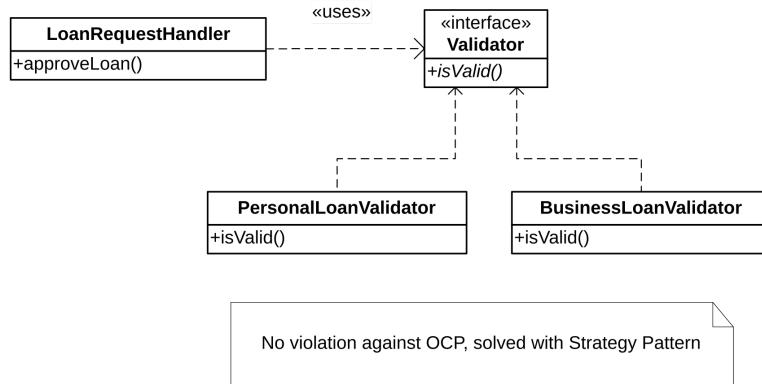
```

1  public class LoanRequestHandler {
2      private final int balance;
3      private final int period;
4
5      public LoanRequestHandler(int balance, int period) {
6          this.balance = balance;
7          this.period = period;
8      }
9
10     public void approveLoan(PersonalLoanValidator validator) {
11         if(validator.isValid(balance)) {
12             System.out.println("Loan approved...");
13         } else {
14             System.out.println("Sorry not enough balance...");
15         }
16     }
17 }
  
```

Bad code

```

1 public class PersonalLoanValidator {
2     public boolean isValid(int balance) {
3         return balance > 1000 ? true : false;
4     }
5 }
```

Compliant Solution**Good code**

```

1 public class LoanRequestHandler {
2     private final int balance;
3     private final int period;
4
5     public LoanRequestHandler(int balance, int period) {
6         this.balance = balance;
7         this.period = period;
8     }
9
10    public void approveLoan(Validator validator) {
11        if (validator.isValid(balance)) {
12            System.out.println("Loan approved...");
13        } else {
14            System.out.println("Sorry not enough balance...");
15        }
16    }
17 }
```

Good code

```

1 public interface Validator {
2     boolean isValid(int balance);
3 }
```

Good code

```
1 public class PersonalLoanValidator implements Validator {  
2     public boolean isValid(int balance) {  
3         return balance > 1000 ? true : false;  
4     }  
5 }
```

Good code

```
1 public class BusinessLoanValidator implements Validator {  
2     public boolean isValid(int balance) {  
3         return balance > 5000 ? true : false;  
4     }  
5 }
```

5.3.2.2 Example: Shape

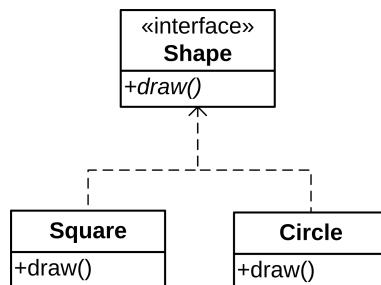
Noncompliant Code Example

Bad code

```

1  public void draw(Shape[] shapes) {
2      for (Shape shape : shapes) {
3          switch (shape.type()) {
4              case Shape.SQUARE:
5                  draw((Square) shape);
6                  break;
7              case Shape.CIRCLE:
8                  draw((Circle) shape);
9                  break;
10         }
11     }
12 }
13
14 private void draw(Circle circle) {
15     // logic for circle
16 }
17
18 private void draw(Square square) {
19     // logic for square
20 }
```

Compliant Solution



Good code

```

1  public interface Shape {
2      void draw();
3 }
```

Good code

```

1  public class Square implements Shape {
2      @Override
3      public void draw() {}
4 }
```

Good code

```
1 public class Circle implements Shape {  
2     @Override  
3     public void draw() {}  
4 }
```

Good code

```
1 public void draw(Shape[] shapes) {  
2     for (Shape shape : shapes) {  
3         shape.draw();  
4     }  
5 }
```

5.3.2.3 Example: HumanResourceDepartment

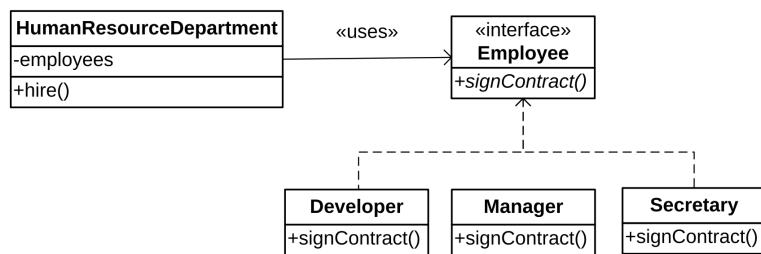
Noncompliant Code Example

Bad code

```

1  public class HumanResourceDepartment {
2
3      private final List<Developer> developers;
4      private final List<Manager> managers;
5
6      public HumanResourceDepartment() {
7          this.developers = new ArrayList<>();
8          this.managers = new ArrayList<>();
9      }
10
11     public void hire(Developer developer) {
12         developer.signContract();
13         this.developers.add(developer);
14     }
15
16     public void hire(Manager manager) {
17         manager.signContract();
18         this.managers.add(manager);
19     }
20 }
```

Compliant Solution



Good code

```

1  interface Employee {
2      void signContract();
3 }
```

Good code

```

1  class Developer implements Employee {
2      public void signContract() {}
3 }
```

Good code

```
1 class Manager implements Employee {  
2     public void signContract() {}  
3 }
```

Good code

```
1 class Secretary implements Employee {  
2     public void signContract() {}  
3 }
```

Good code

```
1 public class HumanResourceDepartment {  
2     private final List<Employee> employees;  
3  
4     public HumanResourceDepartment() {  
5         this.employees = new ArrayList<>();  
6     }  
7  
8     public void hire(Employee employee) {  
9         employee.signContract();  
10        this.employees.add(employee);  
11    }  
12 }
```

5.3.2.4 Example: Calculator

Noncompliant Code Example

Bad code

```
1 public class Calculator {
2     public double calculate(double x, double y, String operator) {
3         double result;
4
5         if ("plus".equals(operator)) {
6             result = x + y;
7         } else if ("minus".equals(operator)) {
8             result = x - y;
9         } else if ("multiply".equals(operator)) {
10            result = x * y;
11        } else if ("divide".equals(operator)) {
12            result = x / y;
13        } else {
14            throw new IllegalArgumentException(String.format("Operator %s not supported!", operator));
15        }
16
17        return result;
18    }
19 }
```

Compliant Solution

Good code

```
1 public enum Operation {
2     PLUS((x, y) -> x + y),
3     MINUS((x, y) -> x - y),
4     MULTIPLY((x, y) -> x * y),
5     DIVIDE((x, y) -> x / y);
6
7     private final DoubleBinaryOperator operator;
8
9     Operation(DoubleBinaryOperator operator) {
10         this.operator = operator;
11     }
12
13     public double apply(double x, double y) {
14         return this.operator.applyAsDouble(x, y);
15     }
16 }
```

Good code

```
1 public class Calculator {  
2     public double calculate(double x, double y, Operation operation) {  
3         return operation.apply(x, y);  
4     }  
5 }
```

5.3.2.5 Example: FileParser

Noncompliant Code Example

Bad code

```
1 public class FileParser {  
2  
3     public String parse(String filePath) {  
4         if (filePath.indexOf(".xml") > 1) {  
5             return parseXML(filePath);  
6         } else if (filePath.indexOf(".json") > 1) {  
7             return parseJson(filePath);  
8         }  
9  
10        return null;  
11    }  
12  
13    private String parseXML(String filePath) {  
14        // logic  
15        return null;  
16    }  
17  
18    private String parseJson(String filePath) {  
19        // logic  
20        return null;  
21    }  
22}
```

Compliant Solution

Good code

```
1 public interface FileParser {  
2     String parse(String file);  
3     boolean supports(String extension);  
4 }
```

Good code

```
1 public class JsonParser implements FileParser {  
2  
3     @Override  
4     public String parse(String file) {  
5         return String.format("Parser %s", JsonParser.class.getSimpleName());  
6     }  
7     @Override  
8     public boolean supports(String extension) {  
9         return "json".equalsIgnoreCase(extension);  
10    }  
11}
```

Good code

```
1 public class XmlParser implements FileParser {  
2  
3     @Override  
4     public String parse(String file) {  
5         return String.format("Parser %s", XmlParser.class.getSimpleName());  
6     }  
7     @Override  
8     public boolean supports(String extension) {  
9         return "xml".equalsIgnoreCase(extension);  
10    }  
11}
```

Good code

```
1 public final class FileParserFactory {  
2  
3     private FileParserFactory() {  
4     }  
5  
6     public static FileParser newInstance(String extension) {  
7         ServiceLoader<FileParser> loader = ServiceLoader.load(FileParser.class);  
8  
9         for (FileParser parser : loader) {  
10             if (parser.supports(extension)) {  
11                 return parser;  
12             }  
13         }  
14         return null;  
15     }  
16 }
```

Good code

```
1 // META-INF/services/swcs.dp.ocp.parser.after.FileParser  
2 swcs.dp.ocp.parser.after.JsonParser  
3 swcs.dp.ocp.parser.after.XmlParser
```

Good code

```
1 public class Client {  
2  
3     public static void main(String[] args) {  
4         FileParser parser = FileParserFactory.newInstance("xml");  
5         System.out.println(parser.parse("test.xml"));  
6     }  
7 }
```

5.3.3 Liskov Substitution Principle

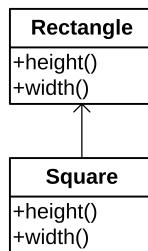
Derived classes must be substitutable for their base classes.

Let $q(x)$ be a property provable about objects x of type T . Then $q(y)$ should be true for objects y of type S , where S is a subtype of T .

- In a clean OOD, the derivation relationship between superclass and subclass represents a so-called **is-a** relationship, i.e. an object of the subclass is compatible in type with a superclass object and can be used wherever a superclass object is required.
- This ensures that superclass type operations applied to a subclass object are performed correctly.
- In this case, it is always safe to replace a superclass-type object with a subclass type object.
- When inheriting, do not only pay attention to the data, but also consider the behaviour of the methods of a class.

5.3.3.1 Example: Rectangle

Noncompliant Code Example



Bad code

```

1 public class Rectangle {
2
3     private double height;
4     private double width;
5
6     public double height() {
7         return this.height;
8     }
9
10    public double width() {
11        return this.width;
12    }
13
14    public void height(double height) {
15        this.height = height;
16    }
17
18    public void width(double width) {
19        this.width = width;
20    }
21
  
```

```
22     public double area() {  
23         return this.width * this.height;  
24     }  
25 }
```

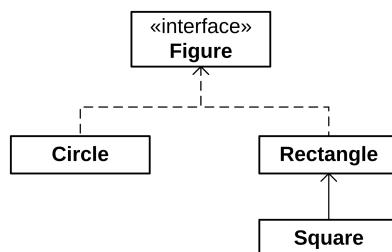
Bad code

```
1  public class Square extends Rectangle {  
2  
3      @Override  
4      public double area() {  
5          return this.height() * this.height();  
6      }  
7  }
```

Bad code

```
1  public void do(Rectangle rectangle) {  
2      rectangle.width(5);  
3      rectangle.height(4);  
4  
5      if (rectangle.area() != 20) {  
6          throw new IllegalStateException("Error in area calculation!");  
7      }  
8  }
```

Compliant Solution



Good code

```
1  public interface Figure {  
2      double area();  
3  }
```

Good code

```
1 public class Circle implements Figure {
2     private final double radius;
3
4     public Circle(double radius) {
5         this.radius = radius;
6     }
7     @Override
8     public double area() {
9         return Math.PI * (this.radius * this.radius);
10    }
11 }
```

Good code

```
1 public class Rectangle implements Figure {
2     private final double length;
3     private final double width;
4
5     public Rectangle(double length, double width) {
6         this.length = length;
7         this.width = width;
8     }
9     @Override
10    public double area() {
11        return this.length * this.width;
12    }
13 }
```

Good code

```
1 public class Square extends Rectangle {
2     public Square(double side) {
3         super(side, side);
4     }
5 }
```

Good code

```
1 public void do(Rectangle rectangle) {
2     // Rectangle double and width was set through constructor
3
4     if (rectangle.area() != 20) {
5         throw new IllegalStateException("Error in area calculation!");
6     }
7 }
```

5.3.3.2 Example: Coupon

Noncompliant Code Example

Bad code

```
1 public class Coupon {
2
3     private final BigDecimal regularPrice;
4
5     public Coupon(BigDecimal regularPrice) {
6         this.regularPrice = regularPrice;
7     }
8
9     public BigDecimal regularPrice() {
10        return this.regularPrice;
11    }
12
13    public BigDecimal afterDiscount(BigDecimal discount) {
14        if (discount == null) {
15            return this.regularPrice;
16        }
17
18        if (this.regularPrice.compareTo(discount) > 0) {
19            return this.regularPrice.subtract(discount);
20        }
21
22        return BigDecimal.ZERO;
23    }
24 }
```

Bad code

```
1 public class PromotionCoupon extends Coupon {
2
3     private static final double PERCENTAGE_DISCOUNT = 0.9;
4
5     public PromotionCoupon(BigDecimal regularPrice) {
6         super(regularPrice);
7     }
8
9     @Override
10    public BigDecimal afterDiscount(BigDecimal discount) {
11        // strengthened pre-conditions and no check for null
12        if (regularPrice().compareTo(discount) < 0) {
13            throw new IllegalArgumentException("Discount can not be greater than the price!");
14        }
15
16        return super.afterDiscount(discount)
17            .multiply(BigDecimal.valueOf(PERCENTAGE_DISCOUNT));
18    }
19 }
```

Bad code

```
1 public class Client {
2
3     private static final BigDecimal REGULAR_PRICE = BigDecimal.valueOf(8.99);
4
5     public static void main(String[] args) {
6         testCoupon(new Coupon(REGULAR_PRICE), null);                                // Pay 8,990000
7         testCoupon(new PromotionCoupon(REGULAR_PRICE), null);                         // NPE
8
9         testCoupon(new Coupon(REGULAR_PRICE), BigDecimal.valueOf(5));                // Pay 3,990000
10        testCoupon(new PromotionCoupon(REGULAR_PRICE), BigDecimal.valueOf(5)); // Pay 3,591000
11
12        testCoupon(new Coupon(REGULAR_PRICE), BigDecimal.valueOf(10));           // You get it for free!
13        testCoupon(new PromotionCoupon(REGULAR_PRICE), BigDecimal.valueOf(10)); // IAE
14    }
15
16    private static void testCoupon(Coupon coupon, BigDecimal discount) {
17        BigDecimal price = coupon.afterDiscount(discount);
18
19        // Client code relies on Coupon implementation
20        if (BigDecimal.ZERO.compareTo(price) == 0) {
21            System.out.println("You get it for free!");
22        } else {
23            System.out.println(String.format("Pay %f", price));
24        }
25    }
26 }
```

Compliant Solution**Good code**

```
1 public interface Coupon {
2
3     BigDecimal regularPrice();
4
5     /**
6      * Returns a {@code BigDecimal} whose value is {@code (+this)},
7      * minus the provided discount.
8      *
9      * @param discount the discount to use.
10     * @return {@code this}, minus the discount. If the discount is
11     *         greater than the regular price {@code BigDecimal.ZERO} will be returned.
12     *         If the discount is {@code null} than this discount will not be subtracted.
13     */
14     BigDecimal afterDiscount(BigDecimal discount);
15 }
```

Good code

```
1 public class RegularCoupon implements Coupon {  
2  
3     private final BigDecimal regularPrice;  
4  
5     public RegularCoupon(BigDecimal regularPrice) {  
6         this.regularPrice = regularPrice;  
7     }  
8  
9     @Override  
10    public BigDecimal regularPrice() {  
11        return this.regularPrice;  
12    }  
13  
14    @Override  
15    public BigDecimal afterDiscount(BigDecimal discount) {  
16        if (discount == null) {  
17            return this.regularPrice;  
18        }  
19  
20        if (this.regularPrice.compareTo(discount) > 0) {  
21            return this.regularPrice.subtract(discount);  
22        }  
23  
24        return BigDecimal.ZERO;  
25    }  
26}
```

Good code

```
1 public final class PromotionCoupon extends RegularCoupon {  
2  
3     private static final double PERCENTAGE_DISCOUNT = 0.9;  
4  
5     public PromotionCoupon(BigDecimal regularPrice) {  
6         super(regularPrice);  
7     }  
8  
9     @Override  
10    public BigDecimal afterDiscount(BigDecimal discount) {  
11        return super.afterDiscount(discount)  
12            .multiply(BigDecimal.valueOf(PERCENTAGE_DISCOUNT));  
13    }  
14}
```

Good code

```
1  public class Client {
2
3      private static final BigDecimal REGULAR_PRICE = BigDecimal.valueOf(8.99);
4
5      public static void main(String[] args) {
6          testCoupon(new RegularCoupon(REGULAR_PRICE), null);                      // Pay 8,990000
7          testCoupon(new PromotionCoupon(REGULAR_PRICE), null);                     // Pay 8,091000
8
9          testCoupon(new RegularCoupon(REGULAR_PRICE), BigDecimal.valueOf(5));     // Pay 3,990000
10         testCoupon(new PromotionCoupon(REGULAR_PRICE), BigDecimal.valueOf(5));   // Pay 3,591000
11
12         testCoupon(new RegularCoupon(REGULAR_PRICE), BigDecimal.valueOf(10));    // You get it for free!
13         testCoupon(new PromotionCoupon(REGULAR_PRICE), BigDecimal.valueOf(10)); // You get it for free!
14     }
15
16     private static void testCoupon(Coupon coupon, BigDecimal discount) {
17         BigDecimal price = coupon.afterDiscount(discount);
18
19         if (BigDecimal.ZERO.compareTo(price) == 0) { // Client code relies on Coupon implementation
20             System.out.println("You get it for free!");
21         } else {
22             System.out.println(String.format("Pay %f", price));
23         }
24     }
25 }
```

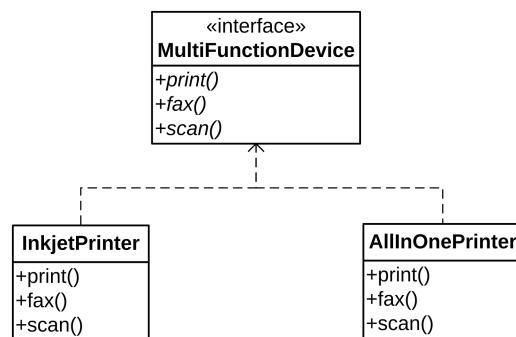
5.3.4 Interface Segregation Principle

Make fine grained interfaces that are client specific.

- Avoidance of *interface pollution* and *fat* interfaces.
- Instead of interfaces with all possible methods that clients might need, there should be separate interfaces that cover the specific needs of each client type.
- Methods of individual interfaces should have low coupling.
- Clients should not depend on methods that are not needed at all.
- Distribution should be made according to the requirements of the clients for the interfaces.
- Clients must therefore only operate with interfaces that can only do what they need.
- Enables a software to be divided into decoupled and therefore flexible classes.
- Future professional or technical requirements therefore only require minor changes to the software itself.

5.3.4.1 Example: `MultiFunctionDevice`

Noncompliant Code Example



Bad code

```

1 public interface MultiFunctionDevice {
2     void print();
3     void fax();
4     void scan();
5 }

```

Bad code

```

1 class AllInOnePrinter implements MultiFunctionDevice {
2     @Override
3     public void print() {
4     }
5
6     @Override
7     public void fax() {
8     }
9
10    @Override
11    public void scan() {
12    }
13 }

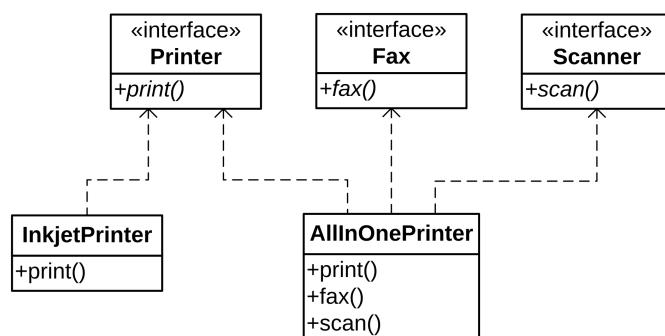
```

Bad code

```

1 class InkjetPrinter implements MultiFunctionDevice {
2
3     @Override
4     public void print() {
5
6     }
7
8     @Override
9     public void fax() {
10        throw new UnsupportedOperationException();
11    }
12
13     @Override
14     public void scan() {
15        throw new UnsupportedOperationException();
16    }

```

Compliant Solution**Good code**

```

1 public interface Printer {
2     void print();
3 }

```

Good code

```

1 public interface Fax {
2     void fax();
3 }

```

Good code

```

1 public interface Scanner {
2     void scan();
3 }

```

Good code

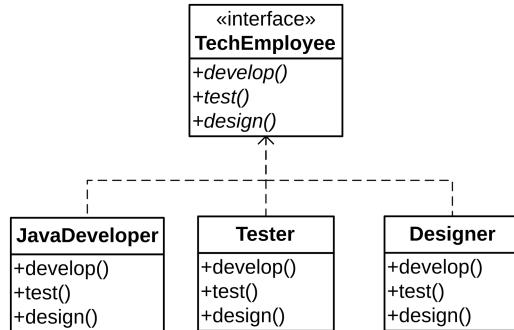
```
1 class InkjetPrinter implements Printer {  
2  
3     @Override  
4     public void print() {  
5         }  
6 }
```

Good code

```
1 class AllInOnePrinter implements Printer, Fax, Scanner {  
2  
3     @Override  
4     public void print() {  
5         }  
6  
7     @Override  
8     public void fax() {  
9         }  
10  
11    @Override  
12    public void scan() {  
13        }  
14 }
```

5.3.4.2 Example: TechEmployee

Noncompliant Code Example



Bad code

```
1 public interface TechEmployee {
2     void develop();
3     void test();
4     void design();
5 }
```

Bad code

```
1 public class JavaDeveloper implements TechEmployee {
2
3     @Override
4     public void develop() {
5         System.out.println("Yes, I love it!");
6     }
7
8     @Override
9     public void test() {
10        throw new UnsupportedOperationException();
11    }
12
13     @Override
14     public void design() {
15         throw new UnsupportedOperationException();
16     }
17 }
```

Bad code

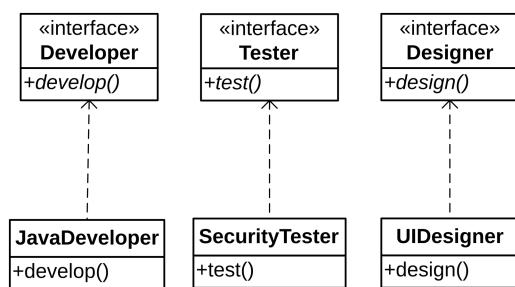
```

1  public class Tester implements TechEmployee {
2
3      @Override
4      public void develop() {
5          throw new UnsupportedOperationException();
6      }
7
8      @Override
9      public void test() {
10         System.out.println("Yes, I love it!");
11     }
12
13     @Override
14     public void design() {
15         throw new UnsupportedOperationException();
16     }
17 }
```

Bad code

```

1  public class Designer implements TechEmployee {
2
3      @Override
4      public void develop() {
5          throw new UnsupportedOperationException();
6      }
7
8      @Override
9      public void test() {
10         throw new UnsupportedOperationException();
11     }
12
13     @Override
14     public void design() {
15         System.out.println("Yes, I love it!");
16     }
17 }
```

Compliant Solution

Good code

```
1 public interface Developer {  
2     void develop();  
3 }
```

Good code

```
1 public interface Designer {  
2     void design();  
3 }
```

Good code

```
1 public interface Tester {  
2     void test();  
3 }
```

Good code

```
1 public class JavaDeveloper implements Developer {  
2  
3     @Override  
4     public void develop() {  
5         System.out.println("Yes, I love it!");  
6     }  
7 }
```

Good code

```
1 public class SecurityTester implements Tester {  
2  
3     @Override  
4     public void test() {  
5         System.out.println("Yes, I love it!");  
6     }  
7 }
```

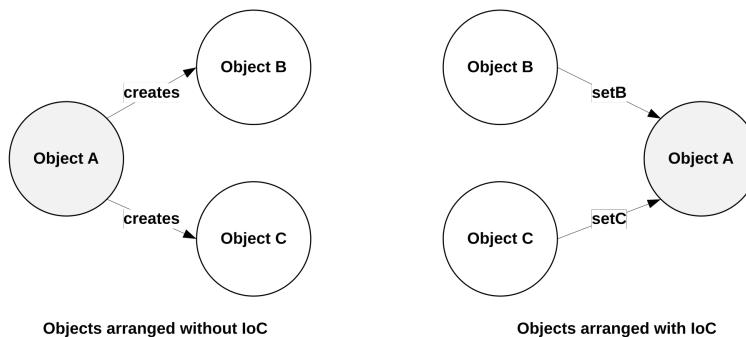
Good code

```
1 public class UIDesigner implements Designer {  
2  
3     @Override  
4     public void design() {  
5         System.out.println("Yes, I love it!");  
6     }  
7 }
```

5.3.5 Dependency Inversion Principle

Depend on abstractions, not on concretions.

- Dependencies should always be directed from more concrete modules of **lower** levels to abstract modules of **higher** levels.
- No class should instantiate foreign classes but should receive them as abstractions (e.g. interfaces) in the form of a parameter.
- This ensures that the dependency relationships always run in one direction, from the concrete to the abstract modules, from the derived classes to the base classes.
- **Hollywood principle:** *Don't call us, we'll call you!*
- Dependencies between objects are controlled from the outside.
 - Reduction of dependencies between the modules
 - Avoidance of cyclical dependencies
 - Decoupling of the component from their environment
 - Reduction of the necessary knowledge



5.3.5.1 Example: `UserService`

Noncompliant Code Example

Without DI the `UserRepository` would have to be created directly in the `UserService`.

Bad code

```

1 public class UserService {
2     private final UserRepository userRepository;
3
4     public UserService() {
5         this.userRepository = new UserRepositoryHibernate();
6     }
7 }
```

Compliant Solution

`UserRepository` is set by constructor injection when the object is created.

Good code

```

1  public class UserService {
2      private final UserRepository userRepository;
3
4      public UserService(UserRepository userRepository) {
5          this.userRepository = userRepository;
6      }
7  }

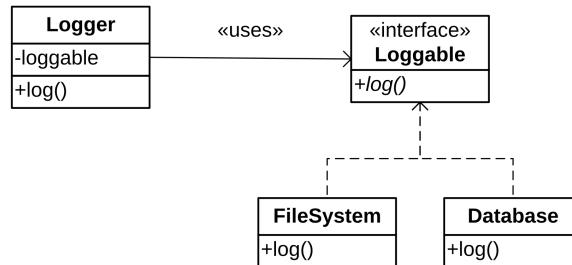
```

5.3.5.2 Example: Logger**Noncompliant Code Example****Bad code**

```

1  class Logger {
2      private final FileSystem fileSystem;
3
4      public Logger() {
5          this.fileSystem = new FileSystem();
6      }
7
8      public void log(String message) {
9          this.fileSystem.log(message);
10     }
11 }

```

Compliant Solution**Good code**

```

1  public interface Loggable {
2      void log(String message);
3  }

```

Good code

```
1 class FileSystem implements Loggable {  
2     @Override  
3     public void log(String message) {  
4         // file handling and writing  
5     }  
6 }
```

Good code

```
1 class Database implements Loggable {  
2     @Override  
3     public void log(String message) {  
4         // writing in database  
5     }  
6 }
```

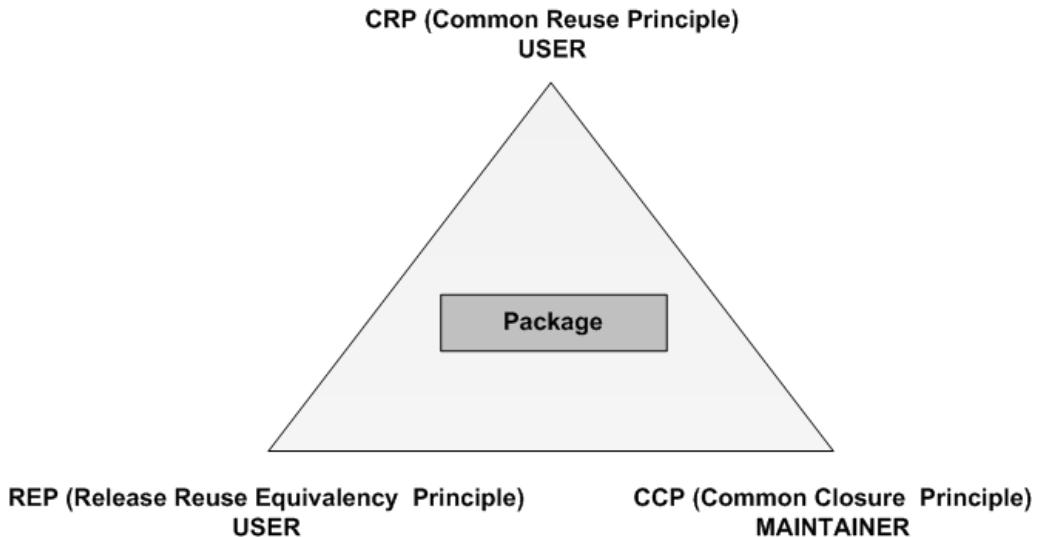
Good code

```
1 class Logger {  
2     private final Loggable loggable;  
3  
4     public Logger(Loggable loggable) {  
5         this.loggable = loggable;  
6     }  
7  
8     public void log(String message) {  
9         this.loggable.log(message);  
10    }  
11 }
```

Good code

```
1 class Client {  
2     public static void main(String[] args) {  
3         Logger fsLogger = new Logger(new FileSystem());  
4         Logger dbLogger = new Logger(new Database());  
5  
6         fsLogger.log("some text");  
7         dbLogger.log("other text");  
8     }  
9 }
```

5.4 Packaging Principles - Cohesion



5.4.1 Release Reuse Equivalency Principle

The granule of reuse is the granule of release.

- The elements of reuse are the elements of the release.
- Classes that are reused together should be in the same package so that they can be released together.
- Reducing the number of modules to be distributed, which in turn reduces the effort of software distribution.
- Avoidance of version conflicts during reuse, which can only arise from the simultaneous release of several packages.
- The users of classes, which in turn depend on other classes, want to rely on a collective release of new versions.

5.4.2 Common Closure Principle

Classes that change together are packaged together.

- All classes in a package should be closed to the same type of changes according to the open closed principle.
- Corresponds to the single responsibility principle applied to packages.
- Changes to the requirements of a software which require changes to a class of a module should only affect the classes of the module.
- Compliance with this principle allows the software to be broken down into packages in such a way that future changes can be implemented in only a few modules.
- This ensures that changes to the software can be made without major side effects and thus relatively inexpensively.

5.4.3 Common Reuse Principle

Classes that are used together are packaged together.

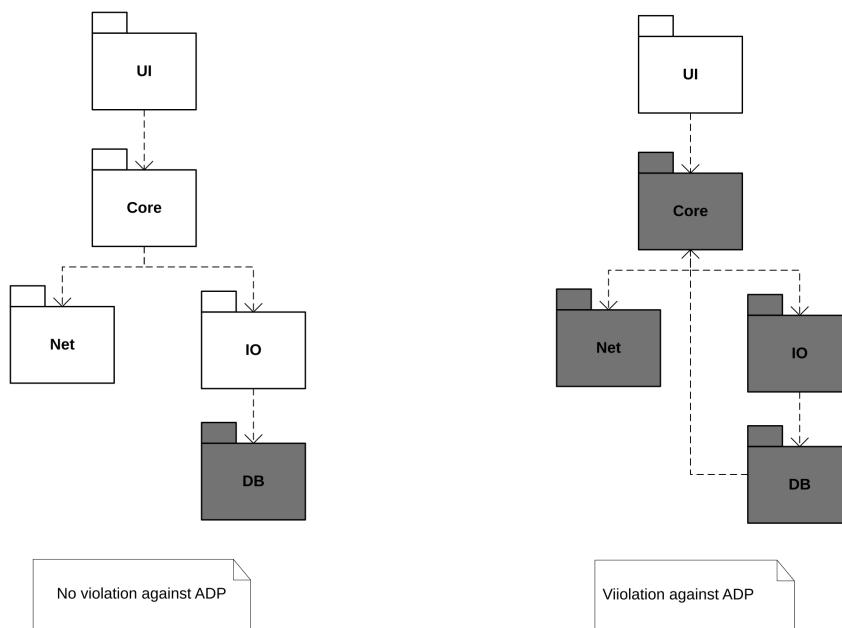
- Grouping of classes that are used together.
- Compliance with this principle ensures that units belonging together are subdivided.

5.5 Packaging Principles - Coupling

5.5.1 Acyclic Dependencies Principle

The dependency graph of packages must have no cycles.

- Classes and packages should not contain cyclic dependencies.
- Make testing, maintainability, reusability, builds and releases more difficult if cycles are not eliminated.



Breaking up such cycles is always possible, but there are basically two possibilities:

If the dependency from package A to package B is to be inverted.

Use of the DIP:

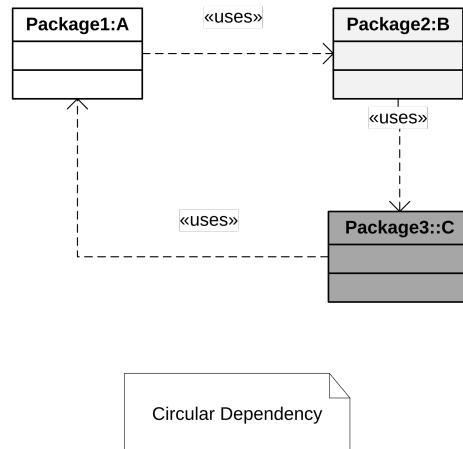
- Introduction of an interface in package A, which holds the methods required by B.
- Implementation of these interfaces in the corresponding classes of Package B.

Restructuring of the packages:

- Collecting all classes of a cycle in a separate package.
- Introducing one or more new packages with the classes on which the classes outside the cycle depend.

5.5.1.1 Example: Cyclic dependency

Noncompliant Code Example



swcs.dp.adp.before.p1

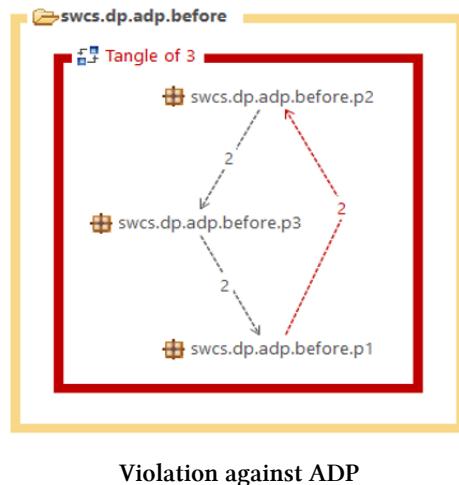
```
1 public class A {  
2     public A() {  
3         new B();  
4     }  
5 }
```

swcs.dp.adp.before.p2

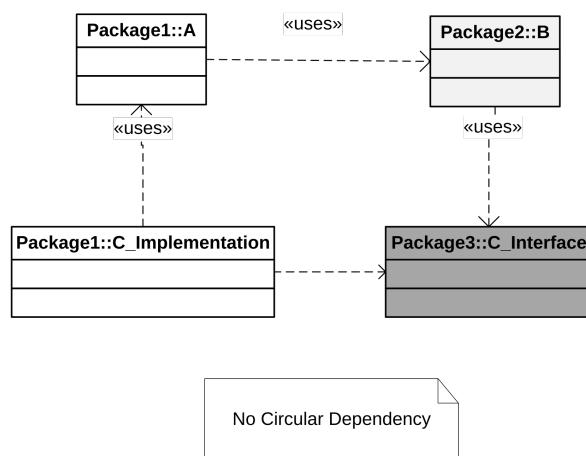
```
1 public class B {  
2     public B() {  
3         new C();  
4     }  
5 }
```

swcs.dp.adp.before.p3

```
1 public class C {  
2     public C() {  
3         new A();  
4     }  
5 }
```



Compliant Solution



swcs.dp.adp.after.p1

```

1 public class A {
2     private final B b;
3
4     public A(B b) {
5         this.b = b;
6     }
7 }
```

swcs.dp.adp.after.p1

```

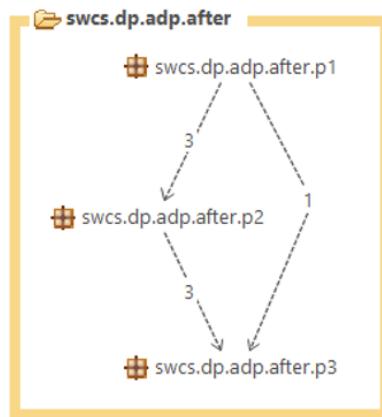
1 public class CImpl implements C {
2     private final A a;
3
4     public CImpl(A a) {
5         this.a = a;
6     }
7 }
```

swcs.dp.adp.after.p2

```
1 public class B {  
2     private final C c;  
3  
4     public B(C c) {  
5         this.c = c;  
6     }  
7 }
```

swcs.dp.adp.after.p3

```
1 public interface C {  
2 }
```

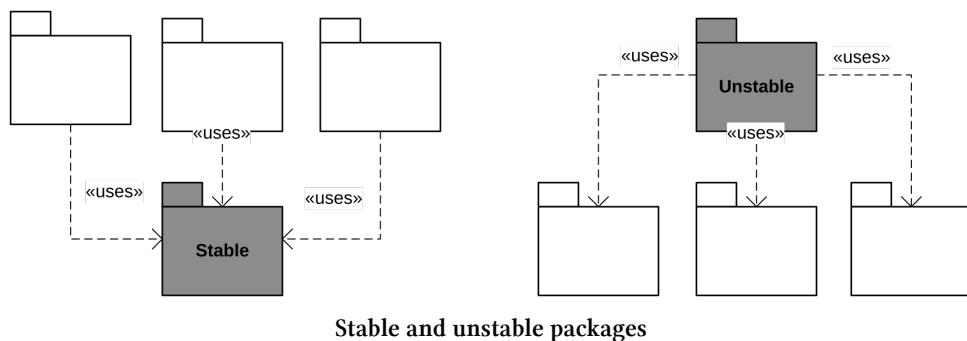


No violation against ADP

5.5.2 Stable Dependencies Principle

Depend in the direction of stability.

- The dependencies between packages should be in the same direction as the stability.
- A package should only depend on packages that are more stable than itself.
- The fewer dependencies a package has on others and the more dependencies other packages have on that package, the more stable it is.



Instability metric

The stability of a package is the ratio of outgoing coupling (c_e) to total coupling ($c_a + c_e$).

```
I = Instability, 0 ≤ I ≤ 1
I = 0, stable package
I = 1, unstable package
```

$$I = \frac{C_e}{C_a + C_e}$$

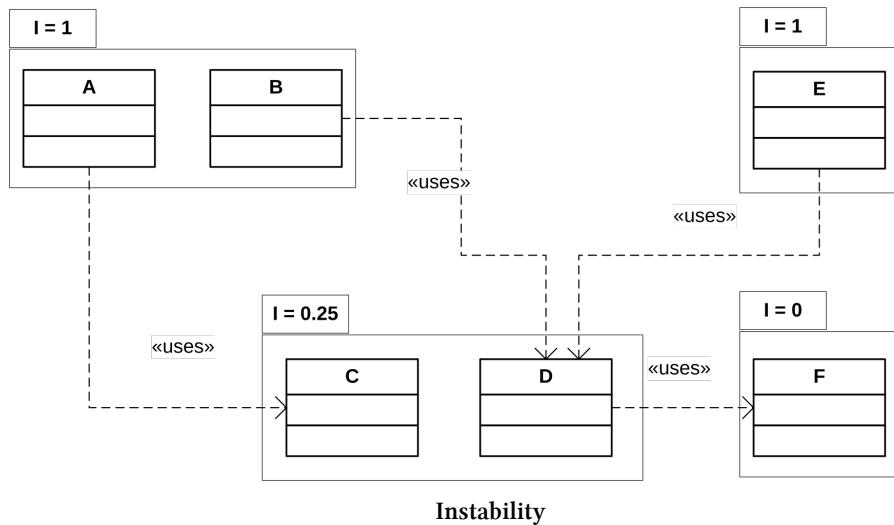
C_a = incoming dependencies (afferent couplings)

C_e = outgoing dependencies (efferent couplings)

An instability of 1 means that no other package depends on this package. This package depends on other packages and is as unstable as a package can be. Its lack of dependencies gives it no reason not to change. Furthermore, the packages it depends on can give it lots of reason to change.

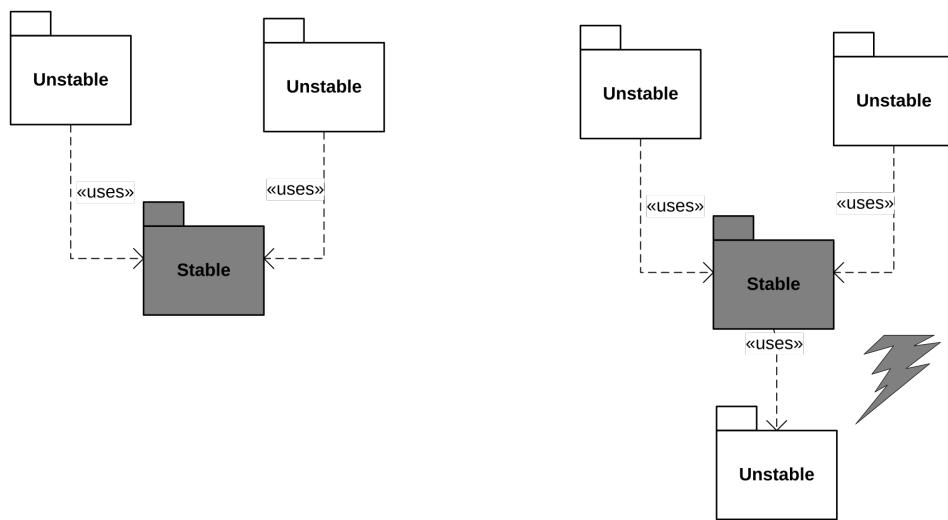
An instability of 0 means that the package depends on other packages, but does not depend on other packages at all. It is independent and is as stable as it can be. Its dependencies make it difficult to change. Further, it has no dependencies that could force it to change.

The SDP says that the metric of a package should be greater than the metrics of the packages it depends on. The metric of instability should decrease in the direction of dependency.



How do we achieve stability? The stable packages are not dependent on anything. A change from a dependent cannot spread to them and cause them to change. They are independent. Independent classes are classes that do not depend on anything else.

Another reason why a stable package gets stable is that many other classes depend on them. The more dependencies they have, the harder it is to make changes to them. If we change classes in the stable package, we would also have to change all the other classes that depend on them. So, there is a strong constraint that stops us from changing these classes. This fact not changing these classes increases their stability.



Stable package depend on unstable package

If all packages are maximally stable, the system would be unchangeable. This is not a desirable situation. We want to design our package structure so that some packages are unstable, and some are stable. The changeable packages should be at the top and depend on the stable packages at the bottom. If we place the unstable packages at the top and arrange them in this way, it does not violate the SDP.

5.5.3 Stable Abstractions Principles

Abstractness increases with stability.

- Establishes a relationship between stability and abstractness.
- Packages that are maximally stable should be maximally abstract.
- Unstable packages should be concrete.
- The abstractness of a package should be proportional to its stability.

Abstractness metric

The abstractness of a package is the ratio of abstract classes like abstract classes and interfaces in a package to the total number of classes in the package.

$A = \text{Abstractness}, 0 \leq A \leq 1$

$A = 0$, completely concrete package

$A = 1$, completely abstract package

$$A = \frac{N_a}{N_c}$$

N_a = number of abstract classes, interfaces

N_c = total number of classes

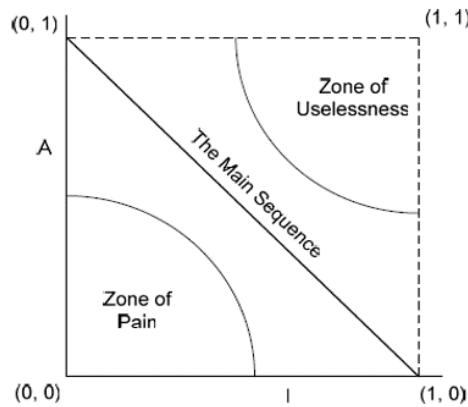
An abstractness of 0 means that the package has no abstract classes. An abstractness of 1 means that the package contains nothing but abstract classes such as interfaces and abstract classes.

Relationship between Instability (I) and Abstractness (A)

The relationship between I and A can be defined within a graph with A on the vertical axis and I on the horizontal axis. There are two *good* types of packages on this graph, the first **Interface packages** that are maximally stable and abstract at the top left at (0,1). The second the **Implementation packages** that are maximally unstable, and concrete are at the bottom right at (1,0).

However, not all packages can be classified in one of these two categories. Thus, packages have degrees of abstraction and stability. Therefore, we must accept that there is a place of points on the A/I graph that defines meaningful positions for packages. This can be done with the Main Sequence which is a line from (0,1) to (1,0). This line represents packages whose abstractness is *balanced* with stability. A packet on the Main Sequence is neither *too abstract* for its stability, nor is it *too unstable* for its abstractness. It has the *right* number of concrete and abstract classes in proportion to its efferent and afferent dependencies.

We can infer in which areas no packages should be, the **Zone of Pain** and the **Zone of Uselessness**.



The Main Sequence

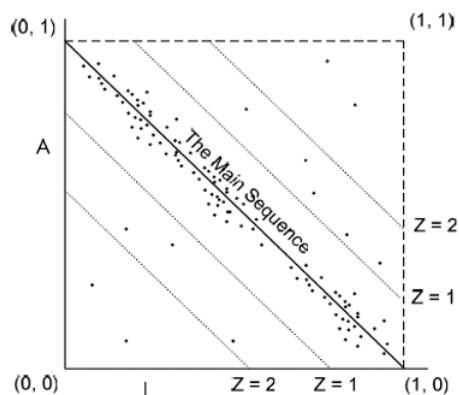
- Ideal line
- Even better when package at (0.1) or (1.0), hardly implemented in practice

Zone of Pain

- Stable non-abstract classes
- Difficult to change because stable, difficult to extend because not abstract
- Exceptions are utility classes such as `String`

Zone of Uselessness

- Unstable very abstract classes
- Are abstract but are nowhere extended



Distance from the Main Sequence

It is obvious that the most desirable positions for a package are at one of the two endpoints of the Main Sequence. However, most packages do not have such ideal attributes. The other packages have the best characteristics when they are on or near the ideal line, which can be defined with the distance to Main Sequence. For each package, the distance to the ideal line

between maximum stability and abstractness and maximum instability and concreteness can be calculated. The greater the distance, the worse the SAP is fulfilled.

$$D = \left| \frac{A + I - 1}{\sqrt{2}} \right|$$

or normalised

$$D' = |A + I - 1|$$

D = Distance from the ideal line, $0 \leq D \leq 0.707$

D' = Normalised distance from the ideal line, $0 \leq D' \leq 1$

A = Abstractness of a package

I = Instability of a package

5.6 Further Design Principles

5.6.1 Speaking Code Principle

- Code should communicate its purpose, even without comment and documentation.
- Comments are no substitute for bad code.
- Clear and expressive code with few comments is much better than messy and complex code with numerous comments.
- Instead of spending time writing comments that explain the chaos, spend time on cleaning up the chaos.
- Comment must be adjusted when the code is changed.

Bad code

```
1 if ((bike.type & TYP_FLAG) && bike.status == READY) { ... }
```

Good code

```
1 if (bike.isRentable()) { ... }
```

5.6.2 Keep It Simple (and) Stupid!

No one is in a position to manage large complexity. Not in software, not in any other aspect of life. Complexity is the killer of good software and therefore simplicity is the enabler. Simplicity is something that takes a lot of time and practice to recognize and produce. Of course, there are many rules that can be followed.

An example would be to have methods with only a few parameters. In the best case, none or only one. We can write "Keep It Simple (and) Stupid!".contains("Simple") and without reading any documentation, everyone will immediately understand what this does and why. The method does one thing and not more. There is no complicated context and no other arguments that can be passed to the method. There are no *special cases* or any caveats.

- Means that always the **simplest** solution of a problem should be chosen.
- Avoid code that is too complex and therefore too complicated.
- Finding simple solutions is a rule that helps to avoid various errors.
- A healthy rejection of non-simple solutions is a sign of searching for good code.
- The more difficult code is to explain, the more likely it is that it is more complicated than necessary and is not the most elegant solution.
- Make things as simple as possible, but not simpler!

5.6.3 Don't Repeat Yourself / Once and Only Once

- Refers to the avoidance of **duplicated code**, i.e. code fragments that are implemented in the same or very similar way in several places.
- Redundant existing source code is difficult to maintain, as consistency between the individual duplicates must be ensured.
- In systems that remain loyal to the DRY principle, however, changes need only be made in one place.

5.6.4 You Ain't Gonna Need It!

- Where no requirement, also no implementation!
- A program should only implement functionality when this functionality is needed.
- Contrary to this approach, in practice it is often attempted to prepare programs for possible future change requests and features through additional or more general code. “...*this is what THEY will demand soon anyway...*”
- Such code can be very annoying to read, understand, change existing functionality and costs time and money!

5.6.5 Separation Of Concerns

- A component should have exactly one task.
- Avoid mixing of several responsibilities in e.g. one class.
- Components become simpler, easier to understand, maintain and have better reusability.
- The complex of tasks of a unit should be self-contained (**high cohesion**).
- The unit should depend as little as possible on other units (**low coupling**).

6. Design Patterns of the Gang of Four

Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way Twice.

—Christopher Alexander 1977, A Pattern Language: Towns, Buildings, Construction

- Design patterns originally come from architecture.
- 1977 Christopher Alexander describes in his book *A Pattern Language. Towns, Buildings, Construction* the design patterns for landscape and urban planning, among other things.
- Kent and Cunningham drew their inspiration for design patterns in software development from Alexander (about 1987).
- Gamma, Vlissides, Johnson and Helm transferred the ideas to software technology and designed a pattern catalogue.
- The Gang of Four (GoF) published the book “Design Patterns” in 1995 and made design patterns popular.

Design patterns are solution patterns for recurring design problems in software design. They are based on experience, they are often only understandable and comprehensible with appropriate background knowledge and personal experience. They establish communication with forming a vocabulary (Pattern Language) that developers use to discuss design. The basics of object-oriented design like *Encapsulation*, *Responsibility*, *Polymorphism* and *Inheritance* must be understood.

Why are design patterns important?

Many problems repeat themselves in software development. Class hierarchies become inflexible over time and individual classes grow uncontrollably. New extensions are very difficult to implement, and the dependencies of a class increase so that the function can be fulfilled. The encapsulation of classes is becoming softened and more and more of the internal state is being exposed.

The wheel does not have to be reinvented every time, at least the schematic solutions are well documented. Since design patterns have existed for a long time, the solutions have been proven many times and the consequences are well understood. Furthermore, software design can be discussed on a higher level of abstraction and thus more efficiently, since the wording is already self-explanatory and known to the developers. Instead of looking for the solution to a problem, there is a sense of recognizing connections that suggest the use of a design pattern.

Classification of design patterns according to GoF

The 23 design patterns of the Gang of Four are divided into three groups. The categories describe the intention that is to be achieved with a design pattern.

Creational Patterns

How objects or even class hierarchies can be created?

Group	Pattern	Description
Creational	Singleton	Ensure a class only has one instance, and provide a global point of access to it.
	Builder	Separate the construction of a complex object from its representation so that the same construction process can create different representations.
	Factory Method	Define an interface for creating an object, but let the subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
	Abstract Factory	Provide an interface for creating families of related or dependent objects without specifying their concrete class.
	Prototype	Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

Structural Patterns

What structure must a composition of classes and objects have in order to solve a certain problem?

Group	Pattern	Description
Structural	Facade	Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
	Decorator	Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
	Adapter	Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
	Composite	Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
	Bridge	Decouple an abstraction from its implementation so that the two can vary independently.
	Flyweight	Use sharing to support large numbers of fine-grained objects efficiently.
	Proxy	Provide a surrogate or placeholder for another object to control access to it.

Behavioural Patterns

How do the objects work together at runtime? How is responsibility distributed among them?

Group	Pattern	Description
Behavioral	State	Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
	Template Method	Define a skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
	Strategy	Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Group	Pattern	Description
	Observer	Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
	Chain of Responsibility	Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.
	Command	Encapsulates a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
	Interpreter	Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.
	Iterator	Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
	Mediator	Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.
	Memento	Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.
	Visitor	Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.



With the introduction of lambdas in Java 8 some design patterns can be implemented in a [different way¹](#) with lambdas.

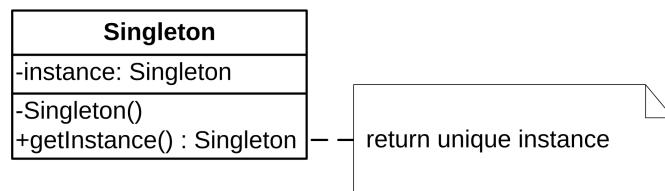
¹<https://github.com/mariofusco/from-gof-to-lambda>

6.1 Creational

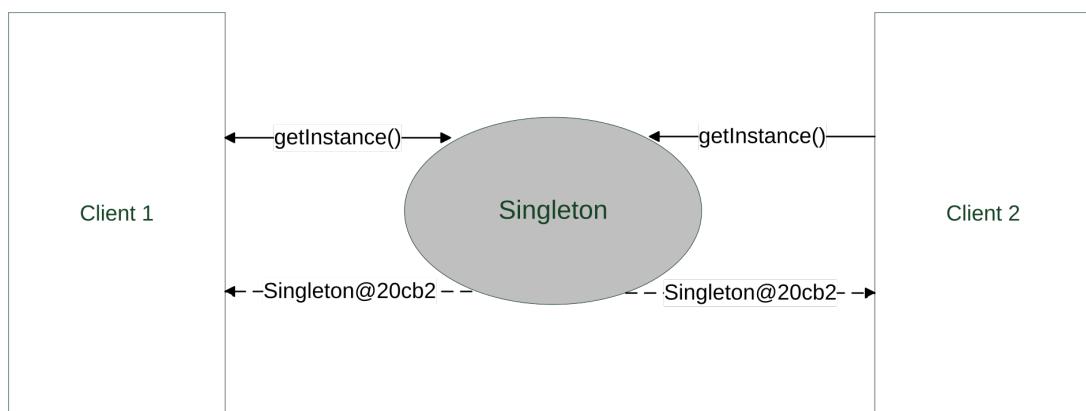
6.1.1 Singleton

Ensure a class only has one instance, and provide a global point of access to it.

- The Singleton design pattern ensures that there is only **one** instance of a class.
- There are three ways how singletons can be implemented.



Singletons are not created at each request. Only single instance is reused again and again.



6.1.1.1 Example: Lazy loading

Lazy loading

```

1  public class Singleton {
2      private static Singleton instance;
3
4      private Singleton() {}
5
6      public static synchronized Singleton getInstance() {
7          if (instance == null) {
8              instance = new Singleton();
9          }
10
11         return instance;
12     }
13 }
  
```

6.1.1.2 Example: Eager loading

Eager loading

```

1  public class Singleton {
2      private static Singleton instance = new Singleton();
3
4      private Singleton() {}
5
6      public static Singleton getInstance() {
7          return instance;
8      }
9 }
```

6.1.1.3 Example: Enum singleton

This approach provides serialisation for free and offers a guarantee against multiple instantiation. A single-element enum type is the best way to implement a singleton.²

Enum singleton, is the preferred approach

```

1  public enum Singleton {
2      INSTANCE
3  }
```

Advantages:

- **Easy to use.** A singleton class is written quickly and easily.
- Guarantee that only **one** instance exists in a JVM at a time.
- Compared to global variables there are several advantages:
 - **Access control.** The Singleton encapsulates its own creation and can therefore control exactly when and how access to the Singleton is allowed.
 - **Lazy-loading.** Singletons can only be created when they are really needed.

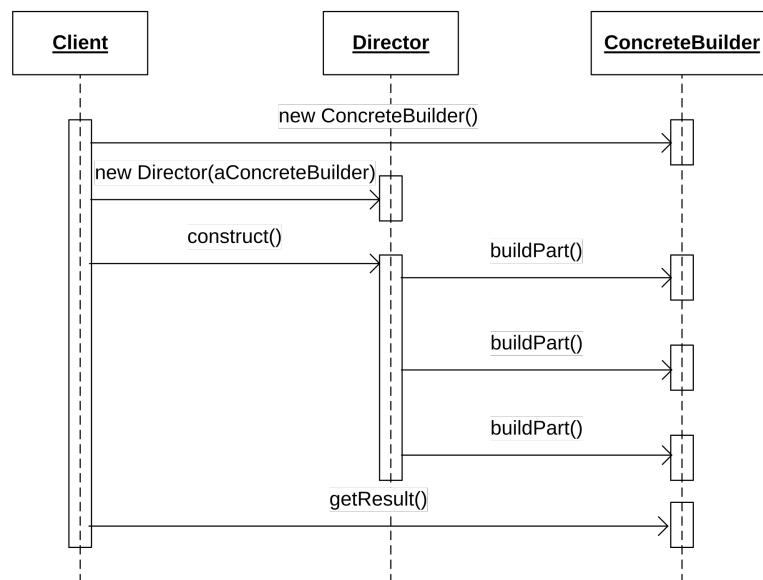
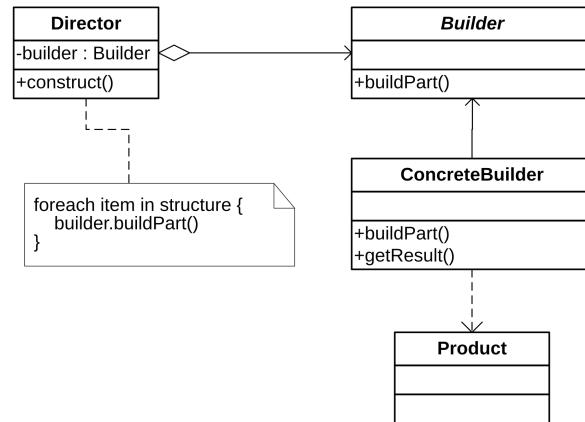
Disadvantages:

- **Problematic destruction.** To destroy objects in languages with garbage collection, an object must no longer be referenced. This is difficult to ensure with singletons. Due to the global availability, it happens very quickly that code parts still hold a reference to the singleton.
- Especially in multi-user applications, a Singleton can reduce the **performance** because it represents a bottleneck - especially in the synchronized form.
- **Uniqueness beyond physical limits.** Ensuring the uniqueness of a Singleton across physical boundaries (JVM) is difficult.

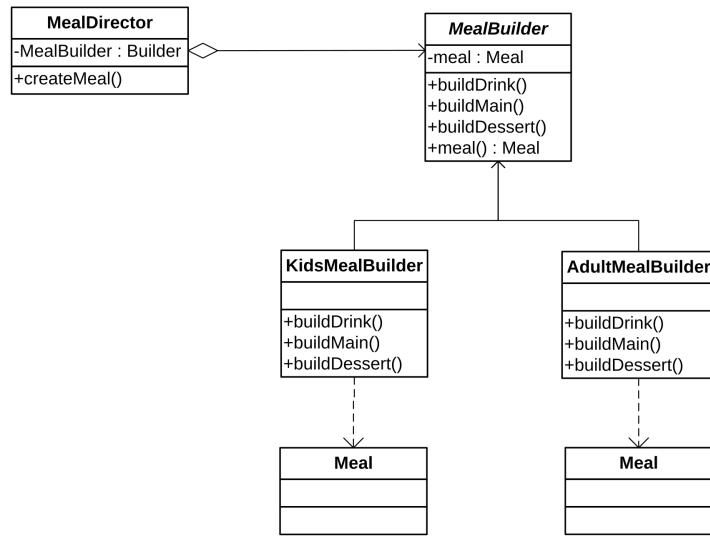
²Bloch, Joshua: Effective Java – Third Edition, Item 3

6.1.2 Builder

Separate the construction of a complex object from its representation so that the same construction process can create different representations.



6.1.2.1 Example: MealBuilder



MealBuilder

```

1 public abstract class MealBuilder {
2     private final Meal meal;
3
4     public MealBuilder() {
5         this.meal = new Meal();
6     }
7
8     public abstract void buildDrink();
9     public abstract void buildMain();
10    public abstract void buildDessert();
11
12    public Meal meal() {
13        return this.meal;
14    }
15 }
  
```

KidsMealBuilder

```

1 public class KidsMealBuilder extends MealBuilder {
2     public void buildDrink() {
3         // add drinks to the meal
4     }
5
6     public void buildMain() {
7         // add main part of the meal
8     }
9
10    public void buildDessert() {
11        // add dessert part to the meal
12    }
13 }
  
```

AdultMealBuilder

```
1 public class AdultMealBuilder extends MealBuilder {
2     public void buildDrink() {
3         // add drinks to the meal
4     }
5
6     public void buildMain() {
7         // add main part of the meal
8     }
9
10    public void buildDessert() {
11        // add dessert part to the meal
12    }
13 }
```

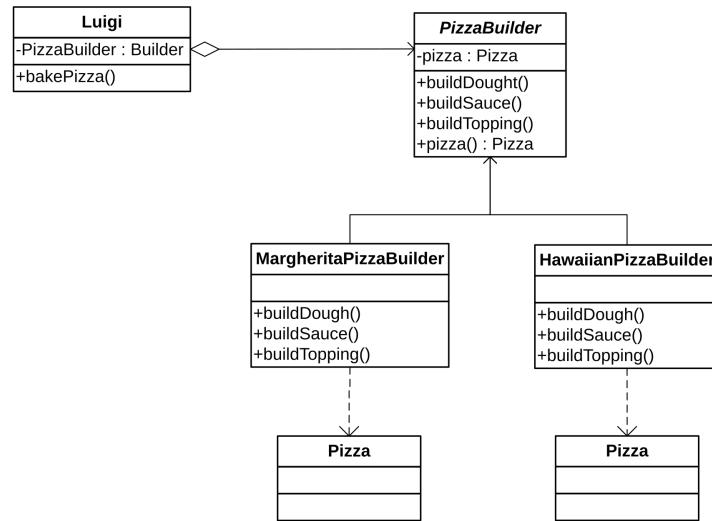
MealDirector

```
1 public class MealDirector {
2
3     public Meal createMeal(MealBuilder builder) {
4         builder.buildDrink();
5         builder.buildMain();
6         builder.buildDessert();
7
8         return builder.meal();
9     }
10 }
```

Client

```
1 public class Client {
2
3     public static void main(String[] args) {
4         MealDirector director = new MealDirector();
5         MealBuilder builder = null;
6         boolean isKid = true;
7
8         if (isKid) {
9             builder = new KidsMealBuilder();
10 } else {
11     builder = new AdultMealBuilder();
12 }
13
14     Meal meal = director.createMeal(builder);
15 }
16 }
```

6.1.2.2 Example: PizzaBuilder



PizzaBuilder

```

1 abstract class PizzaBuilder {
2     private final Pizza pizza;
3
4     public PizzaBuilder() {
5         this.pizza = new Pizza();
6     }
7
8     public Pizza pizza() {
9         return this.pizza;
10    }
11
12    public abstract void buildDough();
13
14    public abstract void buildSauce();
15
16    public abstract void buildTopping();
17}

```

HawaiianPizzaBuilder

```

1 class HawaiianPizzaBuilder extends PizzaBuilder {
2
3     @Override
4     public void buildDough() {
5         this.pizza().dough("cross");
6     }
7
8     @Override
9     public void buildSauce() {
10        this.pizza().sauce("mild");
11    }
12
13     @Override
14     public void buildTopping() {

```

```
15     this.pizza().topping("ham+pineapple");
16 }
17 }
```

MargheritaPizzaBuilder

```
1 class MargheritaPizzaBuilder extends PizzaBuilder {
2
3     @Override
4     public void buildDough() {
5         this.pizza().dough("cross");
6     }
7
8     @Override
9     public void buildSauce() {
10        this.pizza().sauce("mild");
11    }
12
13     @Override
14     public void buildTopping() {
15
16 }
```

Pizza

```
1 final class Pizza {
2     private String dough;
3     private String sauce;
4     private String topping;
5
6     public void dough(String dough) {
7         this.dough = dough;
8     }
9
10    public void sauce(String sauce) {
11        this.sauce = sauce;
12    }
13
14    public void topping(String topping) {
15        this.topping = topping;
16    }
17 }
```

Luigi

```

1  class Luigi {
2      private final PizzaBuilder pizzaBuilder;
3
4      public Luigi(PizzaBuilder pizzaBuilder) {
5          this.pizzaBuilder = pizzaBuilder;
6      }
7
8      public Pizza bakePizza() {
9          this.pizzaBuilder.buildDough();
10         this.pizzaBuilder.buildSauce();
11         this.pizzaBuilder.buildTopping();
12
13         return this.pizzaBuilder.pizza();
14     }
15 }
```

Client

```

1  public class Client {
2
3      public static void main(String[] args) {
4          Luigi luigi = new Luigi(new HawaiianPizzaBuilder());
5          Pizza pizza = luigi.bakePizza();
6      }
7 }
```

6.1.2.3 Example: Email**Email**

```

1  public class Email {
2
3      private final String subject;
4      private final String message;
5      private final String recipients;
6
7      private Email(String subject, String message, String recipients) {
8          this.subject = subject;
9          this.message = message;
10         this.recipients = recipients;
11     }
12
13     public String subject() {
14         return this.subject;
15     }
16
17     public String message() {
18         return this.message;
19     }
20
21     public String recipients() {
22         return this.recipients;
23     }
24 }
```

```
25     public void send() {
26 }
27
28     public static Builder builder() {
29         return new Builder();
30     }
31
32     public static final class Builder {
33
34         private String subject;
35         private String message;
36         private String signature;
37         private final Set<String> recipients;
38
39         public Builder() {
40             this.recipients = new HashSet<>();
41         }
42
43         public Builder withSubject(String subject) {
44             this.subject = subject;
45             return this;
46         }
47
48         public Builder withMessage(String message) {
49             this.message = message;
50             return this;
51         }
52
53         public Builder withSignature(String signature) {
54             this.signature = signature;
55             return this;
56         }
57
58         public Builder addRecipient(String recipient) {
59             this.recipients.add(recipient);
60             return this;
61         }
62
63         public Builder removeRecipient(String recipient) {
64             this.recipients.remove(recipient);
65             return this;
66         }
67
68         public Email build() {
69             return new Email(this.subject,
70                             String.format("%s%n%s", this.message, this.signature),
71                             String.join(".", this.recipients));
72         }
73     }
74 }
```

Client

```
1 public class Client {
2
3     public static void main(String[] args) {
4         Email email = Email.builder()
5             .addRecipient("bad@foo.com")
6             .addRecipient("coder@foo.com")
7             .withMessage("Your first Builder Pattern")
8             .withSignature("Clean Coder")
9             .build();
10
11         email.send();
12     }
13 }
```

6.1.2.4 Example: ImmutablePerson**ImmutablePerson**

```
1 public class ImmutablePerson {
2
3     private final String name;
4     private final String city;
5     private final List<String> favoriteDishes;
6
7     private ImmutablePerson(Builder builder) {
8         this.name = builder.name;
9         this.city = builder.city;
10        this.favoriteDishes = builder.favoriteDishes;
11    }
12
13     public String name() {
14         return this.name;
15     }
16
17     public String city() {
18         return this.city;
19     }
20
21     public List<String> favoriteDishes() {
22         return this.favoriteDishes != null ? new ArrayList<>(this.favoriteDishes) : null;
23     }
24
25     public Builder toBuilder() {
26         return new Builder(this);
27     }
28
29     public static Builder builder() {
30         return new Builder();
31     }
32
33     public static final class Builder {
34
35         private String name;
36         private String city;
37         private List<String> favoriteDishes;
```

```

38
39     public Builder() {
40 }
41
42     public Builder(ImmutablePerson person) {
43         this.name = person.name;
44         this.city = person.city;
45         this.favoriteDishes = person.favoriteDishes != null
46             ? new ArrayList<>(person.favoriteDishes) : null;
47     }
48
49     public Builder withName(String name) {
50         this.name = name;
51         return this;
52     }
53
54     public Builder withCity(String city) {
55         this.city = city;
56         return this;
57     }
58
59     public Builder addFavoriteDish(String dish) {
60         this.favoriteDishes.add(dish);
61         return this;
62     }
63
64     public Builder removeFavoriteDish(String dish) {
65         this.favoriteDishes.remove(dish);
66         return this;
67     }
68
69     public ImmutablePerson build() {
70         return new ImmutablePerson(this);
71     }
72 }
73 }
```

Client

```

1  public class Client {
2
3      public static void main(String[] args) {
4          ImmutablePerson hugo = ImmutablePerson.builder()
5              .withName("Hugo")
6              .withCity("Nuernberg")
7              .addFavoriteDish("Pasta")
8              .addFavoriteDish("Pancake")
9              .build();
10
11         ImmutablePerson mia = ImmutablePerson.builder()
12             .withName("Mia")
13             .withCity("Nuernberg")
14             .addFavoriteDish("Sausage")
15             .build();
16
17         ImmutablePerson hugoAfterMove = hugo.toBuilder()
18             .withCity("Munich")
```

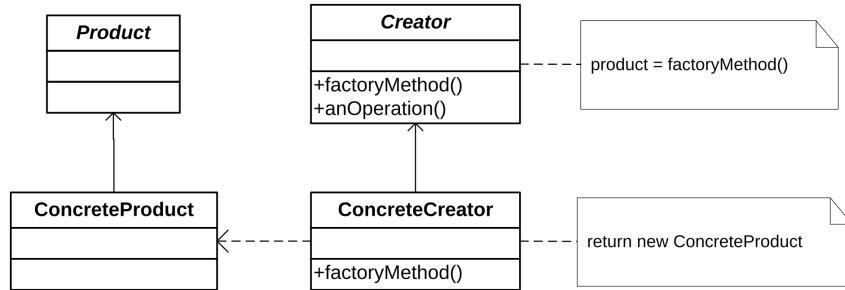
```
19         .build();  
20  
21     ImmutablePerson miaAfterMove = mia.toBuilder()  
22         .withCity("Munich")  
23         .build();  
24     }  
25 }
```

Advantages:

- **Different expressions** of complex products can be built.
- Further expressions can be easily added with **new concrete builder classes**.
- The logic for the construction is **separated**.
- A client does not need to know anything about the **generation process**.
- **Fine-grained control** over the generation process is possible. The product is built piece by piece under the director's control. Other generation patterns build the products in one piece.

6.1.3 Factory Method

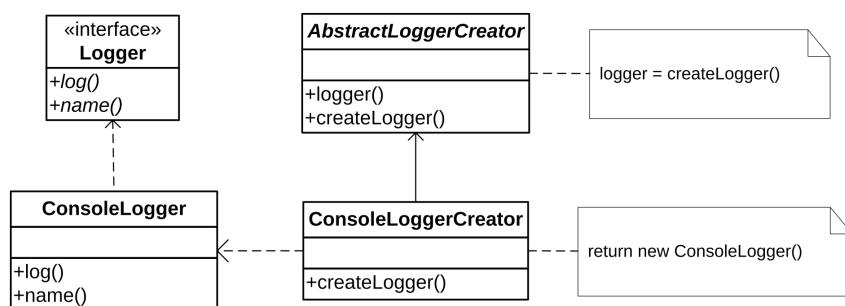
Define an interface for creating an object, but let the subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.



Use cases:

- Separation of object processing (**How?**) from concrete object creation (**What?**).
- Delegation of object instantiation to subclass.
- Cases in which an increasing number and shape of products can be expected. As well as scenarios in which all products have to go through a general manufacturing process, no matter what kind of product they are.
- If the products to be created are not known or should not be defined from the beginning.

6.1.3.1 Example: Logger



Logger

```

1 public interface Logger {
2     void log(String message);
3     String name();
4 }
  
```

ConsoleLogger

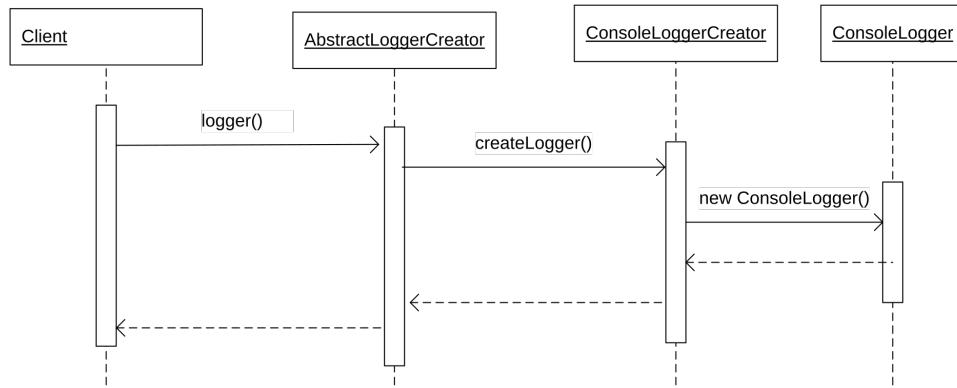
```
1 public class ConsoleLogger implements Logger {  
2  
3     @Override  
4     public void log(String message) {  
5         System.err.println(message);  
6     }  
7  
8     @Override  
9     public String name() {  
10        return getClass().getSimpleName();  
11    }  
12}
```

AbstractLoggerCreator

```
1 public abstract class AbstractLoggerCreator {  
2  
3     public Logger logger() {  
4         // depending on the subclass, we'll get a particular logger.  
5         Logger logger = createLogger();  
6  
7         // could do other operations on the logger here  
8         logger.log(String.format("Logger %s are used.", logger.name()));  
9  
10    return logger;  
11 }  
12  
13    public abstract Logger createLogger();  
14 }
```

ConsoleLoggerCreator

```
1 public class ConsoleLoggerCreator extends AbstractLoggerCreator {  
2  
3     @Override  
4     public Logger createLogger() {  
5         return new ConsoleLogger();  
6     }  
7 }
```

**Client**

```

1 public class Client {
2
3     public static void main(String[] args) {
4         AbstractLoggerCreator creator = new ConsoleLoggerCreator();
5         Logger logger = creator.logger();
6         logger.log("message");
7     }
8 }
```

6.1.3.2 Example: Department**Employee**

```

1 public interface Employee {
2     void paySalary();
3     void dismiss();
4 }
```

Programmer

```

1 public class Programmer implements Employee {
2
3     private final int id;
4
5     public Programmer(int id) {
6         this.id = id;
7     }
8
9     @Override
10    public void paySalary() {
11    }
12
13    @Override
14    public void dismiss() {
15    }
16
17 }
```

MarketingSpecialist

```
1 public class MarketingSpecialist implements Employee {
2
3     private final int id;
4
5     public MarketingSpecialist(int id) {
6         this.id = id;
7     }
8
9     @Override
10    public void paySalary() {
11    }
12
13    @Override
14    public void dismiss() {
15    }
16 }
```

Department

```
1 public abstract class Department {
2
3     public abstract Employee createEmployee(int id);
4
5     public void fire(int id) {
6         Employee employee = createEmployee(id);
7
8         employee.paySalary();
9         employee.dismiss();
10    }
11 }
```

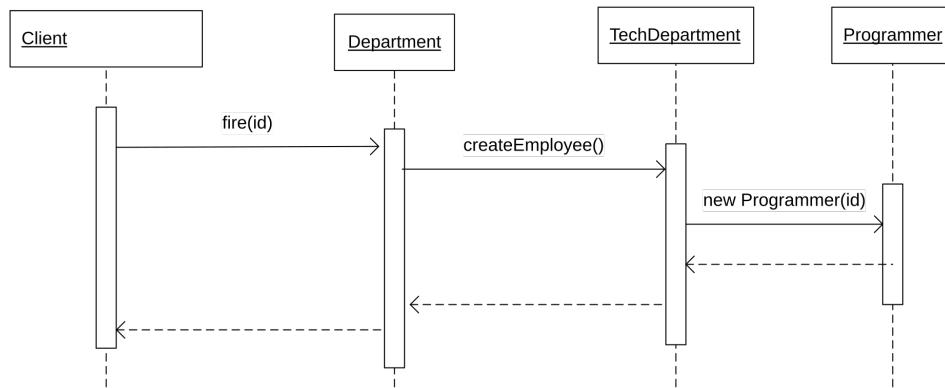
TechDepartment

```
1 public class TechDepartment extends Department {
2
3     @Override
4     public Employee createEmployee(int id) {
5         return new Programmer(id);
6     }
7 }
```

MarketingDepartment

```

1 public class MarketingDepartment extends Department {
2
3     @Override
4     public Employee createEmployee(int id) {
5         return new MarketingSpecialist(id);
6     }
7 }
```



Client

```

1 public class Client {
2
3     public static void main(String[] args) {
4         Department department = new TechDepartment();
5         department.fire(1);
6     }
7 }
```

Advantages:

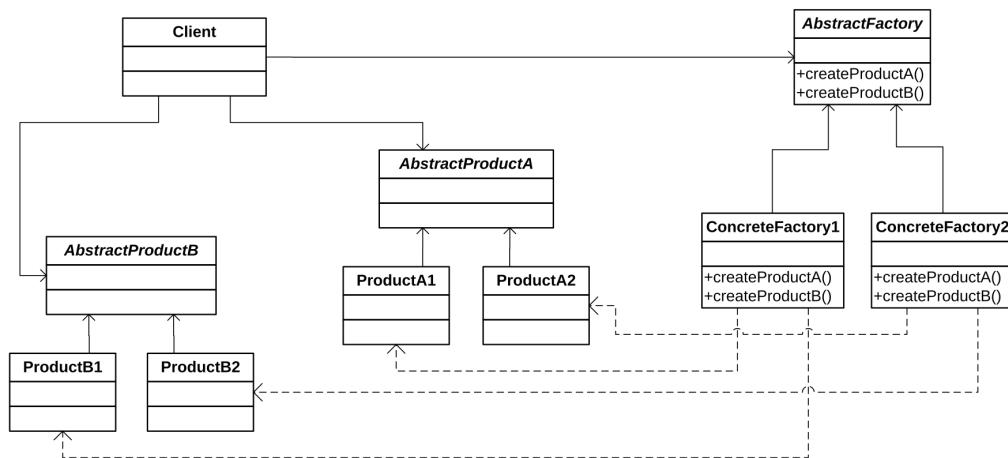
- Manufacturing process is **separated** from a concrete implementation.
- Different product implementations can go through the **same** production process.
- Reusability and consistency:
 - Encapsulation of the object creation code in its own class. This creates a uniform and central interface for the client. The product implementation is **decoupled** from its use.

Disadvantages:

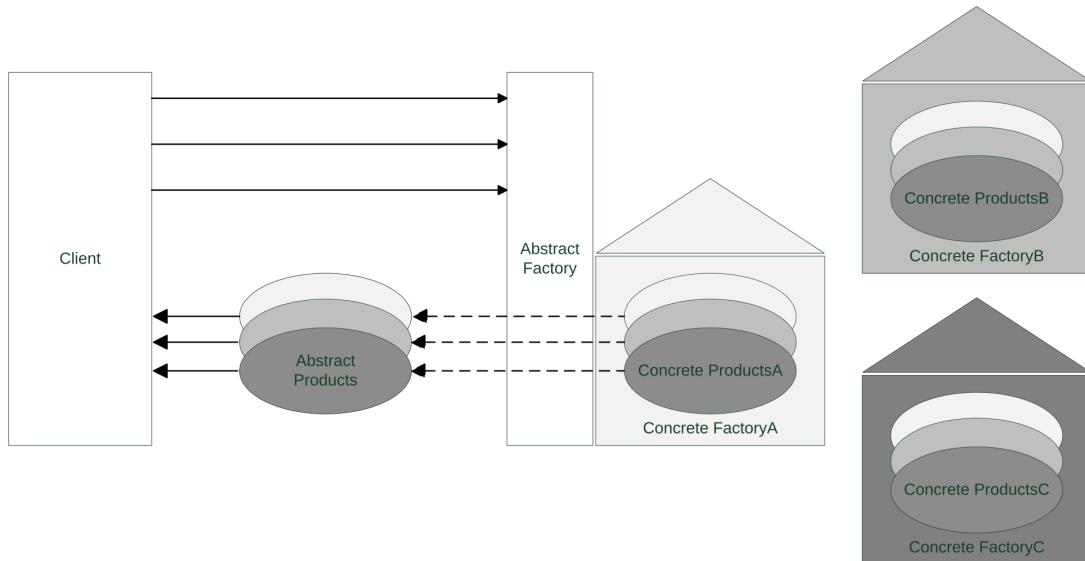
- If many simple objects are to be created, the effort is **disproportionately high**, because the creator always has to be derived.

6.1.4 Abstract Factory

Provide an interface for creating families of related or dependent objects without specifying their concrete class.



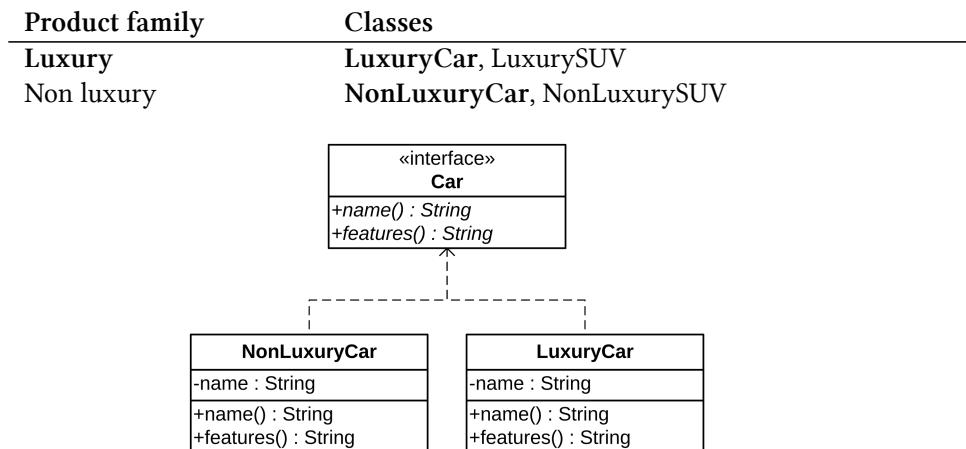
The Abstract Factory Design Pattern serves to define a **coherent family of products**.



Use cases:

- When different objects are created for a context and therefore always have to be created coherently.
- When a system must be configured with different sets of objects or should be able to do so.
- When a system should be independent of how certain objects are created.
- If a family of objects is provided but no statements can or should be made about the concrete implementations. Instead, interfaces are provided.
- Typical applications of abstract factories are e.g., GUI libraries that support different Look & Feels.

6.1.4.1 Example: Car



Car

```

1 public interface Car {
2     String name();
3     String features();
4 }
```

NonLuxuryCar

```

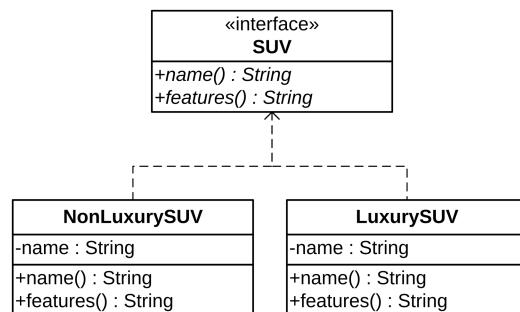
1 public class NonLuxuryCar implements Car {
2     private final String name;
3
4     public NonLuxuryCar(String name) {
5         this.name = name;
6     }
7
8     @Override
9     public String name() {
10        return this.name;
11    }
12
13    @Override
14    public String features() {
15        return "Non-Luxury Car Features ";
16    }
17 }
```

LuxuryCar

```

1  public class LuxuryCar implements Car {
2      private final String name;
3
4      public LuxuryCar(String name) {
5          this.name = name;
6      }
7
8      @Override
9      public String name() {
10         return this.name;
11     }
12
13     @Override
14     public String features() {
15         return "Luxury Car Features ";
16     }
17 }
```

Product family	Classes
Luxury	LuxuryCar, LuxurySUV
Non luxury	NonLuxuryCar, NonLuxurySUV

**Car**

```

1  public interface SUV {
2      String name();
3      String features();
4 }
```

NonLuxurySUV

```

1  public class NonLuxurySUV implements SUV {
2      private final String name;
3
4      public NonLuxurySUV(String name) {
5          this.name = name;
6      }
7
8      @Override
9      public String name() {
10         return this.name;
11     }
12 }
```

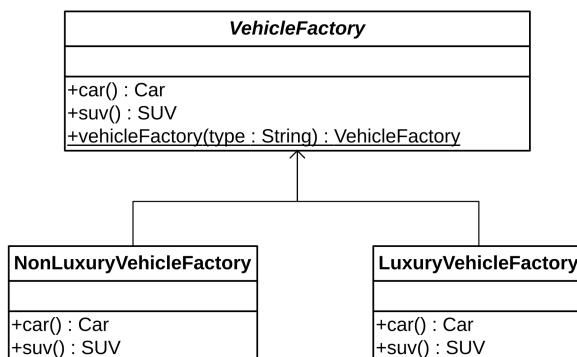
```

12
13     @Override
14     public String features() {
15         return "Non-Luxury SUV Features ";
16     }
17 }
```

LuxurySUV

```

1 public class LuxurySUV implements SUV {
2     private final String name;
3
4     public LuxurySUV(String name) {
5         this.name = name;
6     }
7
8     @Override
9     public String name() {
10        return this.name;
11    }
12
13     @Override
14     public String features() {
15         return "Luxury SUV Features ";
16     }
17 }
```

**VehicleFactory**

```

1 public abstract class VehicleFactory {
2     public static final String LUXURY_VEHICLE = "Luxury";
3     public static final String NON_LUXURY_VEHICLE = "Non-Luxury";
4
5     public abstract Car car();
6     public abstract SUV suv();
7
8     public static VehicleFactory vehicleFactory(String type) {
9         if (VehicleFactory.LUXURY_VEHICLE.equals(type)) {
10             return new LuxuryVehicleFactory();
11         }
12         if (VehicleFactory.NON_LUXURY_VEHICLE.equals(type)) {
13             return new NonLuxuryVehicleFactory();
14         }
15     }
16 }
```

```

15
16     return new LuxuryVehicleFactory();
17 }
18 }
```

NonLuxuryVehicleFactory

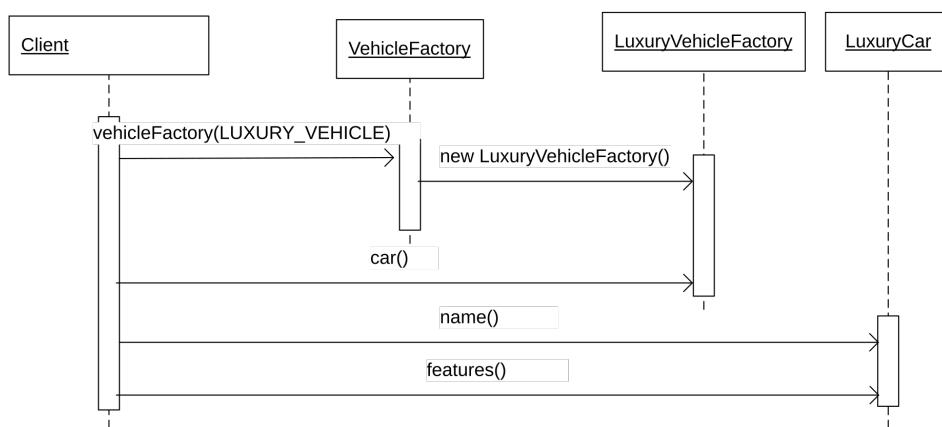
```

1 public class NonLuxuryVehicleFactory extends VehicleFactory {
2
3     @Override
4     public Car car() {
5         return new NonLuxuryCar("NL-C");
6     }
7
8     @Override
9     public SUV suv() {
10        return new NonLuxurySUV("NL-S");
11    }
12 }
```

LuxuryVehicleFactory

```

1 public class LuxuryVehicleFactory extends VehicleFactory {
2
3     @Override
4     public Car car() {
5         return new LuxuryCar("L-C");
6     }
7
8     @Override
9     public SUV suv() {
10        return new LuxurySUV("L-S");
11    }
12 }
```



Client

```
1 public class Client {
2
3     public static void main(String[] args) {
4         VehicleFactory vf = VehicleFactory.vehicleFactory(VehicleFactory.LUXURY_VEHICLE);
5
6         Car car = vf.car();
7         System.out.println("Name: " + car.name() + " Features: " + car.features());
8
9         SUV suv = vf.suv();
10        System.out.println("Name: " + suv.name() + " Features: " + suv.features());
11    }
12 }
```

Advantages:

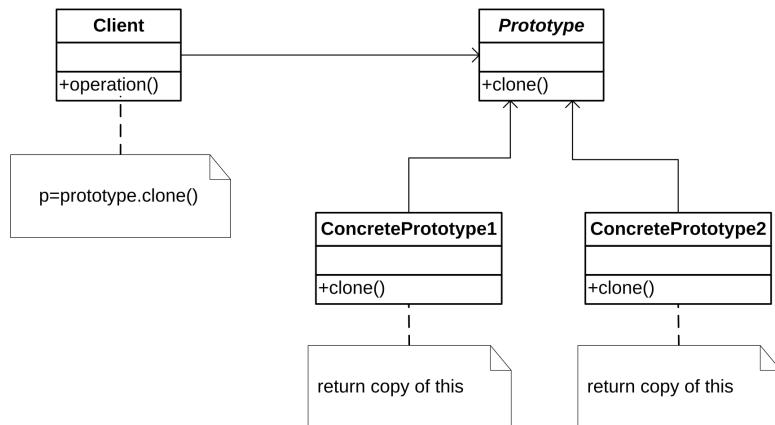
- By **shielding the concrete classes**, the client code becomes generally valid. No code is necessary for special cases.
- **Consistency**. It is ensured that only those objects that fit together reach the client.
- **Flexibility**. Entire object families can be exchanged, since the client only relies on abstractions (Abstract Factory, product interfaces).
- **Simple extension with new product families**. New product families can be integrated into the system very easily. All that is required is the reimplementations of the factory interface. Afterwards, the new factory only needs to be instantiated at a central point in the client.

Disadvantages:

- **Inflexibility regarding new family members**. If a new product is to be added to the product family, the interface of the Abstract Factory must be changed. However, this leads to the breaking of code of all concrete factories. The change effort is high.
- A new implementation with great similarity to an existing one still requires a new factory.

6.1.5 Prototype

Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.



Use cases:

- Creating a copy of an prototype object and making required modifications.

Java provides some ways to copy an object in order to create a new one. One is using the `java.lang.Cloneable` marker interface. You need to implement the `java.lang.Cloneable` interface and override the `clone` method which every object inherits from the `Object` class.

There are two categories regarding cloning, **shallow** and **deep cloning**.

A shallow copy copies all reference types or value types, but it does not copy the objects that the references refer to. The references in the new object point to the same objects that the references in the original object points to. The default `clone` method implementation in Java creates a clone as a shallow copy.

A deep copy copies the elements and everything directly or indirectly referenced by the elements.

6.1.5.1 Example: Person - Shallow copy

Person

```

1  public class Person implements Cloneable {
2
3      private String name;
4      private Address address;
5
6      public Person(String name, Address address) {
7          this.name = name;
8          this.address = address;
9      }
10
11     public String name() {
12         return this.name;
13     }
  
```

```
13     }
14
15     public void name(String name) {
16         this.name = name;
17     }
18
19     public Address address() {
20         return this.address;
21     }
22
23     public void address(Address address) {
24         this.address = address;
25     }
26
27     @Override
28     public Person clone() {
29         try {
30             return (Person) super.clone();
31         } catch (CloneNotSupportedException e) {
32             throw new AssertionError();
33         }
34     }
35 }
```

Address

```
1  public class Address implements Cloneable {
2
3      private String city;
4      private String street;
5
6      public Address(String city, String street) {
7          this.city = city;
8          this.street = street;
9      }
10
11     public String city() {
12         return this.city;
13     }
14
15     public void city(String city) {
16         this.city = city;
17     }
18
19     public String street() {
20         return this.street;
21     }
22
23     public void street(String street) {
24         this.street = street;
25     }
26
27     @Override
28     public Address clone() {
29         try {
30             return (Address) super.clone();
31         } catch (CloneNotSupportedException e) {
```

```
32         throw new AssertionError();
33     }
34 }
35 }
```

Client

```
1 public class Client {
2
3     public static void main(String[] args) {
4
5         Person orginal = new Person("Orginal name", new Address("Orginal city", "Orginal street"));
6         print("Orginal", orginal);
7
8         // Clone as a shallow copy
9         Person clone = orginal.clone();
10        print("Clone", clone);
11
12        clone.name("Modified name");
13        clone.address().city("Modified city");
14
15        print("Clone after modification", clone);
16        print("Orginal after clone modification", orginal);
17    }
18
19    private static void print(String object, Person person) {
20        System.out.println(String.format("%s: %s, %s",
21            object,
22            person.name(),
23            person.address().city()));
24    }
25 }
```

Output

```
1 Orginal: Orginal name, Orginal city
2 Clone: Orginal name, Orginal city
3 Clone after modification: Modified name, Modified city
4 Orginal after clone modification: Orginal name, Modified city
```

6.1.5.2 Example: Person - Deep copy

Address

```

1  public class Address implements Cloneable {
2
3      private String city;
4      private String street;
5
6      public Address(String city, String street) {
7          this.city = city;
8          this.street = street;
9      }
10
11     public String city() {
12         return this.city;
13     }
14
15     public void city(String city) {
16         this.city = city;
17     }
18
19     public String street() {
20         return this.street;
21     }
22
23     public void street(String street) {
24         this.street = street;
25     }
26
27     @Override
28     public Address clone() {
29         try {
30             return (Address) super.clone();
31         } catch (CloneNotSupportedException e) {
32             throw new AssertionError();
33         }
34     }
35 }
```

Person

```

1  public class Person implements Cloneable {
2
3      private String name;
4      private Address address;
5
6      public Person(String name, Address address) {
7          this.name = name;
8          this.address = address;
9      }
10
11     public String name() {
12         return this.name;
13     }
14
15     public void name(String name) {
```

```
16     this.name = name;
17 }
18
19     public Address address() {
20         return this.address;
21     }
22
23     public void address(Address address) {
24         this.address = address;
25     }
26
27     @Override
28     public Person clone() {
29         try {
30             Person person = (Person) super.clone();
31             person.address(person.address.clone());
32
33             return person;
34         } catch (CloneNotSupportedException e) {
35             throw new AssertionError();
36         }
37     }
38 }
```

Client

```
1 public class Client {
2
3     public static void main(String[] args) {
4
5         Person orginal = new Person("Orginal name", new Address("Orginal city", "Orginal street"));
6         print("Orginal", orginal);
7
8         // Clone as a deep copy
9         Person clone = orginal.clone();
10        print("Clone", clone);
11
12        clone.name("Modified name");
13        clone.address().city("Modified city");
14
15        print("Clone after modification", clone);
16        print("Orginal after clone modification", orginal);
17    }
18
19     private static void print(String object, Person person) {
20         System.out.println(String.format("%s: %s, %s",
21                                         object,
22                                         person.name(),
23                                         person.address().city()));
24     }
25 }
```

Output

```

1 Orginal: Orginal name, Orginal city
2 Clone: Orginal name, Orginal city
3 Clone after modification: Modified name, Modified city
4 Orginal after clone modification: Orginal name, Orginal city

```

6.1.5.3 Example: Person - Copy constructor / factory

According to Effective Java `clone` and `java.lang.Cloneable` is complex and the rules for overriding `clone` are tricky and difficult to get right.

Object's `clone` method is very tricky. It's based on field copies, and it's *extra-linguistic*. It creates an object without calling a constructor. There are no guarantees that it preserves the invariants established by the constructors. If that state is mutable, you don't have two independent objects. If you modify one, the other changes as well. And all of a sudden, you get random behavior. A better approach is providing a *copy constructor* or *copy factory*.³

Person

```

1 public class Person {
2
3     private String name;
4     private Address address;
5
6     // Copy constructor
7     public Person(Person person) {
8         this(person.name(), new Address(person.address()));
9     }
10
11    public Person(String name, Address address) {
12        this.name = name;
13        this.address = address;
14    }
15
16    public String name() {
17        return this.name;
18    }
19
20    public void name(String name) {
21        this.name = name;
22    }
23
24    public Address address() {
25        return this.address;
26    }
27
28    public void address(Address address) {
29        this.address = address;
30    }
31
32    // Copy factory
33    public static Person newInstance(Person person) {
34        return new Person(person.name(), new Address(person.address()));

```

³Bloch, Joshua: Effective Java – Third Edition, Item 13

```
35     }
36 }
```

Address

```
1  public class Address {
2
3      private String city;
4      private String street;
5
6      // Copy constructor
7      public Address(Address address) {
8          this(address.city(), address.street());
9      }
10
11     public Address(String city, String street) {
12         this.city = city;
13         this.street = street;
14     }
15
16     public String city() {
17         return this.city;
18     }
19
20     public void city(String city) {
21         this.city = city;
22     }
23
24     public String street() {
25         return this.street;
26     }
27
28     public void street(String street) {
29         this.street = street;
30     }
31
32     // Copy factory
33     public static Address newInstance(Address address) {
34         return new Address(address.city(), address.street());
35     }
36 }
```

Client

```
1  public class Client {
2
3      public static void main(String[] args) {
4
5          Person orginal = new Person("Orginal name", new Address("Orginal city", "Orginal street"));
6          print("Orginal", orginal);
7
8          // Clone as a deep copy with copy constructor
9          Person clone = new Person(orginal);
10         // Alternative as a deep copy with copy factory
11         // Person clone = Person.newInstance(orginal);
12         print("Clone", clone);
```

```
13     clone.name("Modified name");
14     clone.address().city("Modified city");
15
16     print("Clone after modification", clone);
17     print("Orginal after clone modification", orginal);
18 }
19
20
21 private static void print(String object, Person person) {
22     System.out.println(String.format("%s: %s, %s",
23         object,
24         person.name(),
25         person.address().city()));
26 }
27 }
```

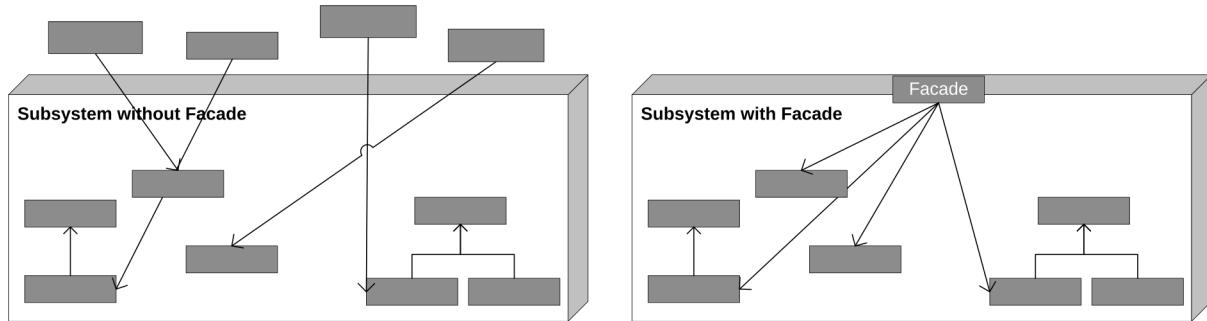
Output

```
1 Orginal: Orginal name, Orginal city
2 Clone: Orginal name, Orginal city
3 Clone after modification: Modified name, Modified city
4 Orginal after clone modification: Orginal name, Orginal city
```

6.2 Structural

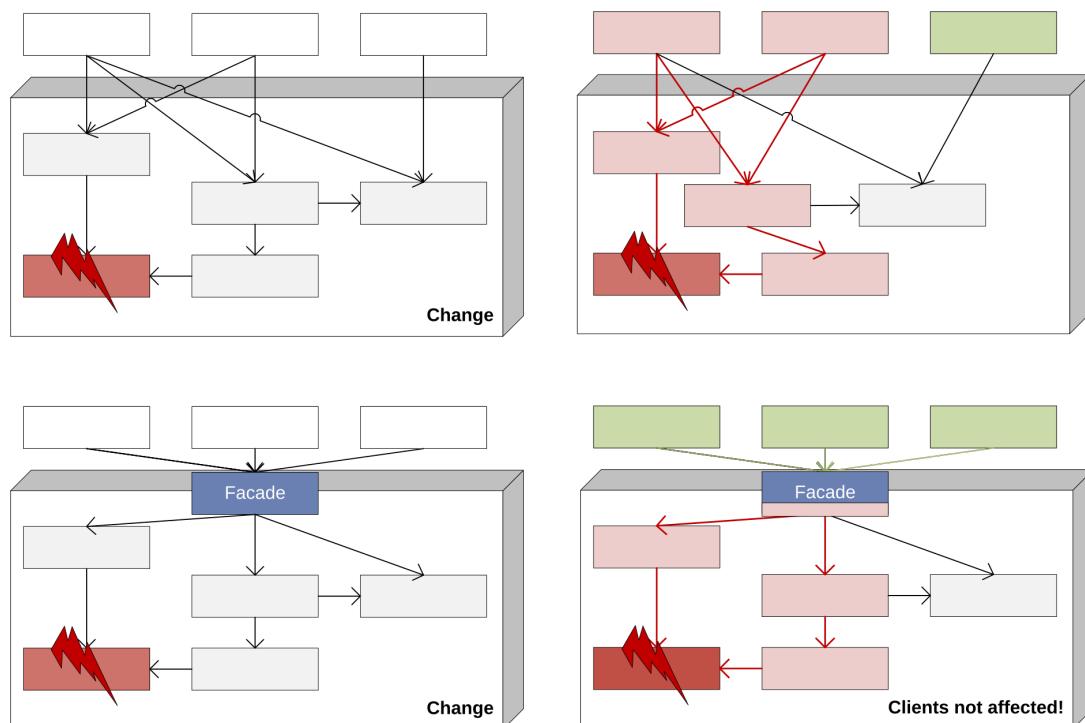
6.2.1 Facade

Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.



Use cases:

- Division of a complex system into several simple subsystems.
- Facade promotes loose coupling.
- Subdivision of a system into layers and the communication of these via facades.
- Reduction of dependencies between the client and the subsystem used.



6.2.1.1 Example: Travel

HotelBooker

```

1 public class HotelBooker {
2     public List<Hotel> hotels(LocalDate from, LocalDate to) {
3         // returns hotels available in the particular date range
4     }
5 }
```

FlightBooker

```

1 public class FlightBooker {
2     public List<Flight> flights(LocalDate from, LocalDate to) {
3         // returns flights available in the particular date range
4     }
5 }
```

TravelFacade

```

1 public class TravelFacade {
2     private HotelBooker hotelBooker;
3     private FlightBooker flightBooker;
4
5     public Travels availableTravels(LocalDate from, LocalDate to) {
6         List<Flight> flights = flightBooker.flights(from, to);
7         List<Hotel> hotels = hotelBooker.hotels(from, to);
8
9         // process
10    }
11 }
```

Client

```

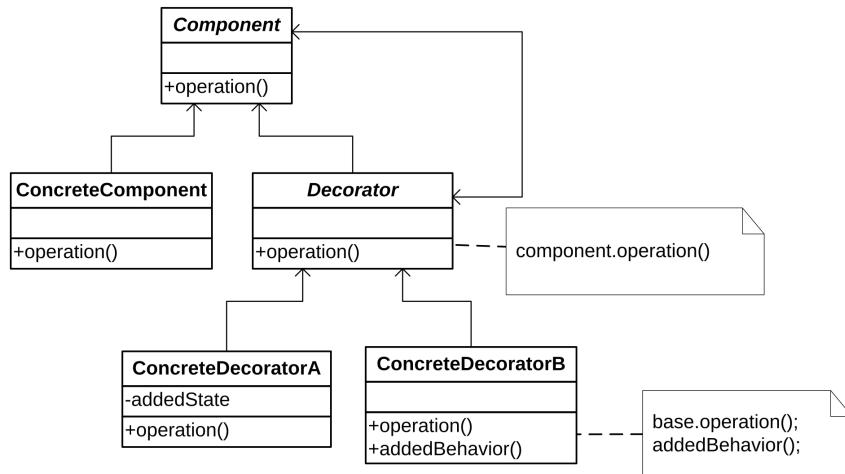
1 public class Client {
2     public static void main(String[] args) {
3         TravelFacade facade = new TravelFacade();
4         facade.availableTravels(from, to);
5     }
6 }
```

Advantages:

- **Simplified interface.** The client can use a complex system more easily by reducing the number of objects that the client needs to know.
- **Decoupling** the client from the subsystem. Since the client only works against the facade, it is independent of changes in the subsystem.
- **Maintenance** and modifications of the subsystem only mean changes within the system and at most the facade implementation. The interface of the facade to the outside remains unchanged.

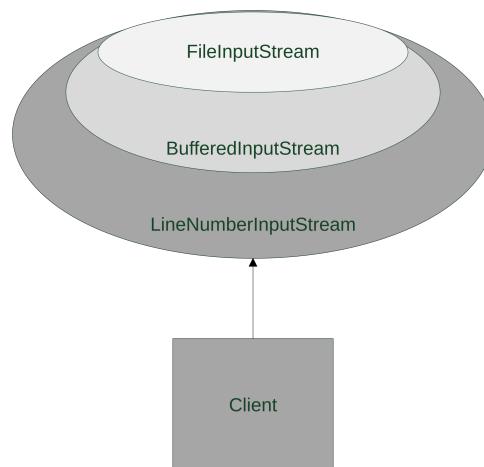
6.2.2 Decorator

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.



Use cases:

- Alternative to static subclassing (inheritance) of objects
- Dynamic adding or removing functionalities
- Extension of `final` classes, which are therefore not inheritable
- If functionality extension by means of static inheritance is impracticable
 - z. e.g., with an unmanageable number of classes, which would result if every possible combination of extensions were considered
- Practical examples:
 - Java IO API, `BufferedInputStream` extends the interface around a buffer
 - Different types of cars with different features (lowering, spoiler)
 - Dish with various side dishes
 - Different types of telephone (mobile phone, smartphone, classic telephone) with variable behaviour (vibration, ringing, silent, lights)



6.2.2.1 Example: Message

Message

```
1 public interface Message {  
2     void print(String text);  
3 }
```

DefaultMessage

```
1 public class DefaultMessage implements Message {  
2  
3     @Override  
4     public void print(String text) {  
5         System.out.println(text);  
6     }  
7 }
```

TimeStampMessage

```
1 public class TimeStampMessage implements Message {  
2  
3     private final Message message;  
4  
5     public TimeStampMessage(Message message) {  
6         this.message = message;  
7     }  
8  
9     @Override  
10    public void print(String text) {  
11        this.message.print(String.format("%s %s", LocalDateTime.now(), text));  
12    }  
13 }
```

UpperCaseMessage

```
1 public class UpperCaseMessage implements Message {  
2  
3     private final Message message;  
4  
5     public UpperCaseMessage(Message message) {  
6         this.message = message;  
7     }  
8  
9     @Override  
10    public void print(String text) {  
11        this.message.print(text.toUpperCase());  
12    }  
13 }
```

Client

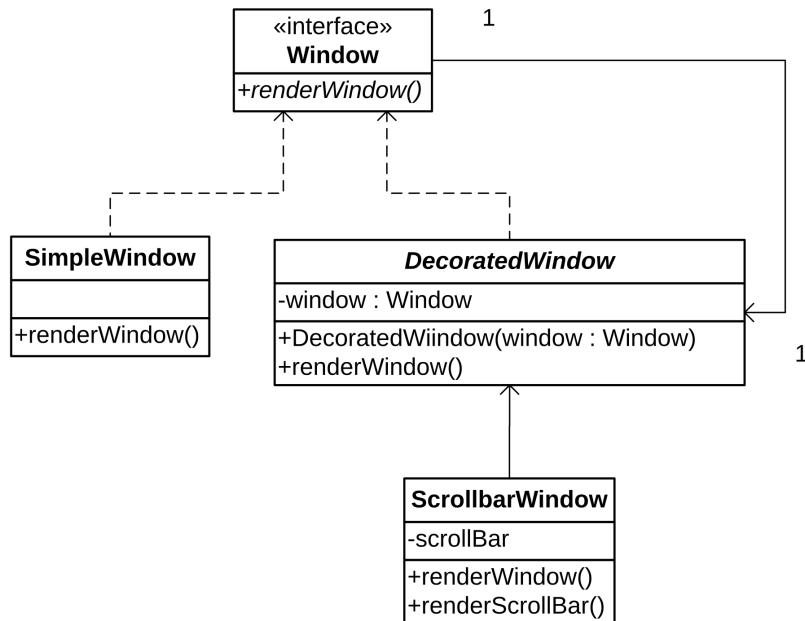
```

1 public class Client {
2
3     public static void main(String[] args) {
4         new DefaultMessage().print("DefaultMessage");
5         new UpperCaseMessage(new DefaultMessage()).print("DefaultMessage");
6         new TimeStampMessage(new DefaultMessage()).print("DefaultMessage");
7         new UpperCaseMessage(new TimeStampMessage(new DefaultMessage())).print("DefaultMessage");
8     }
9 }
```

Output

```

1 DefaultMessage
2 DEFAULTMESSAGE
3 2020-12-29T13:04:38.388325800 DefaultMessage
4 2020-12-29T13:04:38.416309600 DEFAULTMESSAGE
```

6.2.2.2 Example: Window**Window**

```

1 public interface Window {
2     void renderWindow();
3 }
```

SimpleWindow

```

1  public class SimpleWindow implements Window {
2      public void renderWindow() {
3          // implementation of rendering details
4      }
5 }
```

DecoratedWindow

```

1  public class DecoratedWindow implements Window {
2      private Window window = null;
3
4      public DecoratedWindow(Window window) {
5          this.window = window;
6      }
7
8      public void renderWindow() {
9          window.renderWindow();
10     }
11 }
```

ScrolledWindow

```

1  public class ScrolledWindow extends DecoratedWindow {
2      private Object scrollBar = null;
3
4      public ScrolledWindow(Window window) {
5          super(window);
6      }
7
8      public void renderWindow() {
9          super.renderWindow();
10         renderScrollBar();
11     }
12
13     private void renderScrollBar() {
14         scrollBar = new Object(); // prepare scrollbar
15         // render scrollbar
16     }
17 }
```

Client

```

1  public class Client {
2      public static void main(String[] args) {
3          Window window = new SimpleWindow();
4          window.renderWindow();
5
6          // at some point later maybe text size becomes larger
7          // than the window thus the scrolling behavior must be added
8
9          // decorate old window, now window object has additional behavior
10         window = new ScrolledWindow(window);
11     }
```

```
12     window.renderWindow();
13 }
14 }
```

Advantages:

- Decorator offers **more possibilities** than inheritance, as it is not static and therefore defines a behaviour of the subclass.
- **Slim, cohesive classes.** Each decorator represents exactly one function and nothing else. This increases class cohesion and makes the corresponding code easier to maintain and extend.
- Avoidance of long and confusing inheritance hierarchies.
- By combining a few Decorator objects, the functionality of the object can be extended considerably.

Disadvantages:

- Decorated objects are wrapped by the actual decorator. This makes it difficult to determine the identity of the objects.
- **High number** of objects. Each additional feature requires a new Decorator object. The number of many small, similar objects and their initialization code can quickly become confusing. Can be fixed by a factory.
- Decorator-Pattern is a **dynamic process**, so in general the runtime behaviour is worse than with classical inheritance.

6.2.3 Adapter

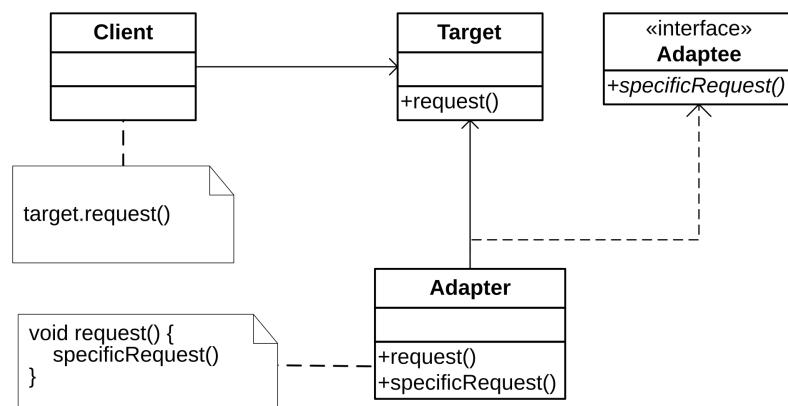
Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

The adapter pattern defines four roles:

- **Adaptee** which is to be adapted to the target interface.
- **Adapter** who performs the adaptation.
- **Target interface** to be used by the client.
- **Client** who uses the adaptee through the adapter.

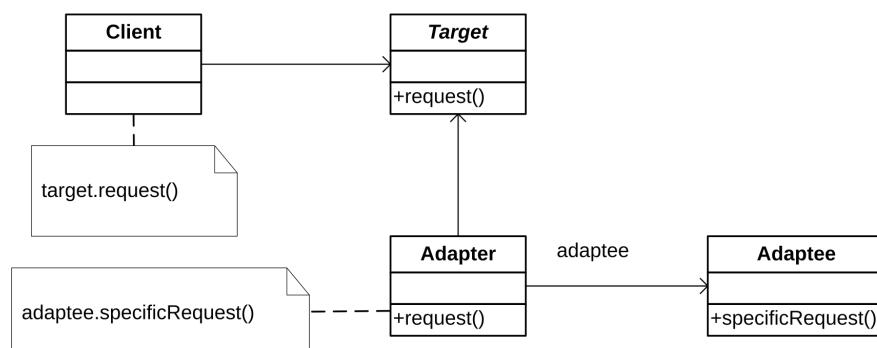
Class Adapter uses multiple inheritance:

In the case of the class adapter, the adapter inherits our adaptee class and implements the target interface. When a client object calls a method of the class adapter, the adapter internally calls a method of the adapter it inherits.

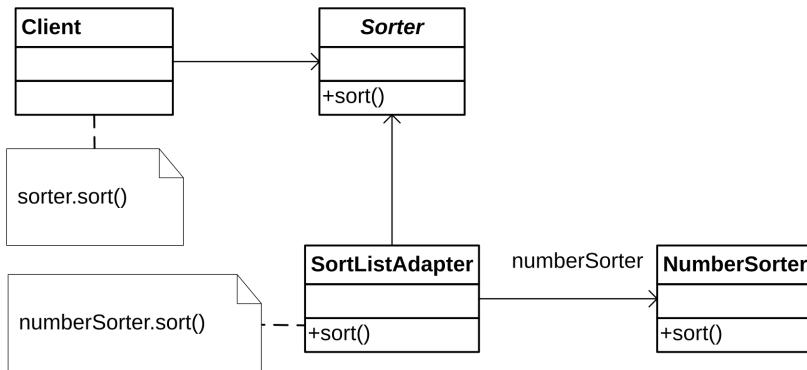


Object Adapter relies on object composition:

In the case of an object adapter, the adapter delegates to an instance of the adaptee. The adapter implements the target interface, but does not inherit the classes that need to be adapted. When a client calls a method of the object adapter, the object adapter calls a corresponding method on the instance of the adaptee it references.



6.2.3.1 Example: Sorter



Sorter - Target interface

```

1 public interface Sorter {
2     int[] sort(int[] numbers);
3 }
  
```

SortListAdapter - Adapter

```

1 public class SortListAdapter implements Sorter {
2
3     public int[] sort(int[] numbers) {
4         List<Integer> numberList = convertArrayToList(numbers);
5
6         NumberSorter numberSorter = new NumberSorter();
7         numberList = numberSorter.sort(numberList);
8
9         return convertListToArray(numberList);
10    }
11 }
  
```

NumberSorter - Adaptee

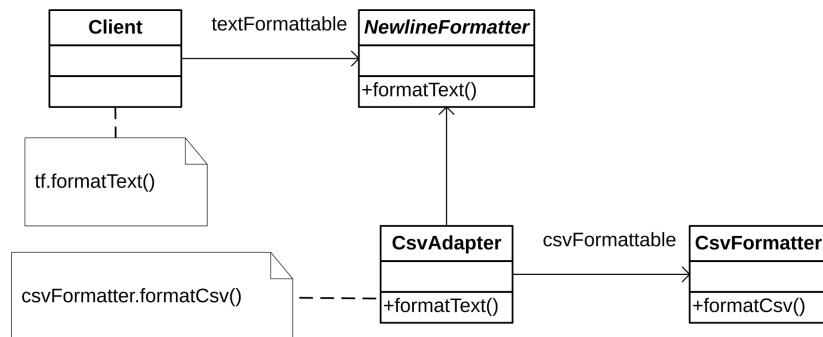
```

1 // A third party implementation of a number sorter that deals with lists, not arrays.
2 public class NumberSorter {
3     public List<Integer> sort(List<Integer> numbers) {
4         return sort(numbers);
5     }
6 }
  
```

Client

```

1 public class Client {
2     public static void main(String[] args) {
3         Sorter sorter = new SortListAdapter();
4         sorter.sort(new int[] { 34, 2, 4, 12, 1 });
5     }
6 }
```

6.2.3.2 Example: TextFormatter**TextFormattable**

```

1 public interface TextFormattable {
2     String formatText(String text, String separator);
3 }
```

NewlineFormatter

```

1 public class NewlineFormatter implements TextFormattable {
2
3     @Override
4     public String formatText(String text, String separator) {
5         if (text != null && separator != null) {
6             return Stream.of(text.split(Pattern.quote(separator)))
7                 .map(String::stripLeading)
8                 .collect(Collectors.joining(System.lineSeparator()));
9         }
10
11         return null;
12     }
13 }
```

CsvFormattable

```

1 public interface CsvFormattable {
2     String formatCsv(String text, String separator);
3 }
```

CsvFormatter

```
1 public class CsvFormatter implements CsvFormattable {  
2  
3     @Override  
4     public String formatCsv(String text, String separator) {  
5         if (text != null && separator != null) {  
6             return Stream.of(text.split(Pattern.quote(separator)))  
7                         .collect(Collectors.joining(", "));  
8         }  
9  
10        return null;  
11    }  
12}
```

CsvAdapter

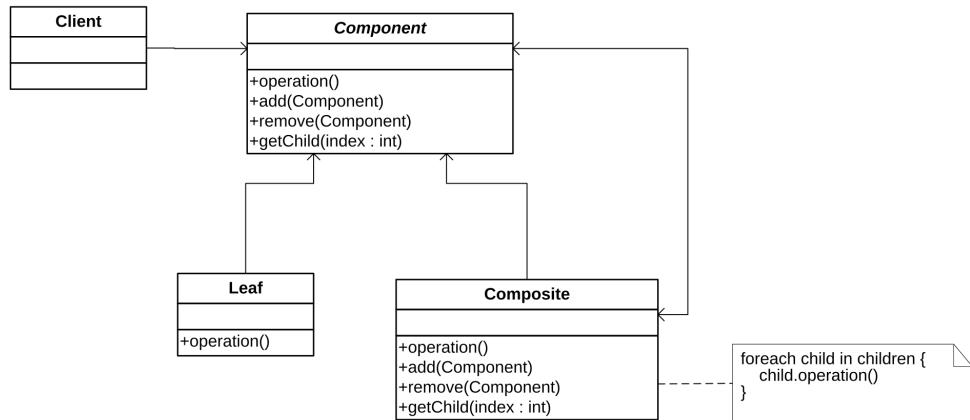
```
1 public class CsvAdapter implements TextFormattable {  
2  
3     private final CsvFormatter csvFormatter;  
4  
5     public CsvAdapter(CsvFormatter csvFormatter) {  
6         this.csvFormatter = csvFormatter;  
7     }  
8  
9     @Override  
10    public String formatText(String text, String separator) {  
11        return this.csvFormatter.formatCsv(text, separator);  
12    }  
13}
```

Client

```
1 public class Client {  
2  
3     public static void main(String[] args) {  
4         String text = "Adapter Pattern 1. Adapter Pattern 2. Adapter Pattern 3."  
5  
6         TextFormattable tf = new NewlineFormatter();  
7         System.out.println(tf.formatText(text, ".") );  
8  
9         tf = new CsvAdapter(new CsvFormatter());  
10        System.out.println(tf.formatText(text, ".") );  
11    }  
12}
```

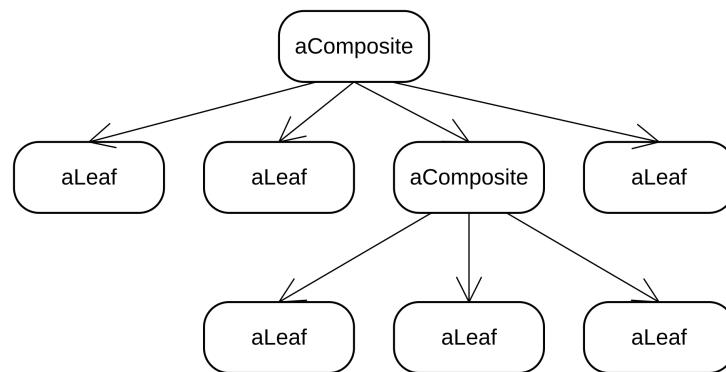
6.2.4 Composite

Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.



Use cases:

- Manipulation of a hierarchical collection of simple and complex objects.
- Modelling of a tree-like structure of objects
 - **File system.** Files and folders can be modelled with the composite pattern. Files represent the leafs and folders represent the composites, as they may contain other files and folders.
 - **Menus.** Menus consist of a root entry (composite) and commands (leafs).
 - Hierarchy of graphic objects e.g. in Java / AWT (Container & Component)



6.2.4.1 Example: Graphic

Component

```
1 public interface Component {  
2     void paint();  
3 }
```

Composite

```
1 public abstract class Composite implements Component {  
2  
3     private List<Component> components = new ArrayList<Component>();  
4  
5     @Override  
6     public void paint() {  
7         for (Component component : this.components) {  
8             component.paint();  
9         }  
10    }  
11  
12    public void add(Component component) {  
13        this.components.add(component);  
14    }  
15  
16    public void remove(Component component) {  
17        if (this.components.contains(component))  
18            this.components.remove(component);  
19    }  
20  
21    public Component get(int index) {  
22        return this.components.get(index);  
23    }  
24 }
```

Sheet

```
1 public class Sheet extends Composite {  
2     public void paint() {  
3         System.out.println(Sheet.class.getSimpleName());  
4         super.paint();  
5     }  
6 }
```

Row

```
1 public class Row extends Composite {  
2     public void paint() {  
3         System.out.println(String.format(" %s", Row.class.getSimpleName()));  
4         super.paint();  
5     }  
6 }
```

Column

```
1 public class Column extends Composite {  
2     public void paint() {  
3         System.out.println(String.format(" %s", Column.class.getSimpleName()));  
4         super.paint();  
5     }  
6 }
```

Client

```
1 public class Client {  
2     public static void main(String[] args) {  
3         Composite sheet = new Sheet();  
4  
5         Composite r1 = new Row();  
6         r1.add(new Column());  
7         r1.add(new Column());  
8         sheet.add(r1);  
9  
10        Composite r2 = new Row();  
11        r2.add(new Column());  
12        r2.add(new Column());  
13        sheet.add(r2);  
14  
15        sheet.paint();  
16    }  
17 }
```

Client output

```
1 Sheet  
2   Row  
3     Column  
4     Column  
5   Row  
6     Column  
7     Column
```

6.2.4.2 Example: Organization Chart

Employee

```

1 public interface Employee {
2     void draw();
3 }
```

DisciplinaryLeadership

```

1 public abstract class DisciplinaryLeadership implements Employee {
2
3     private final List<Employee> employees = new ArrayList<>();
4
5     @Override
6     public void draw() {
7         for (Employee employee : this.employees) {
8             employee.draw();
9         }
10    }
11
12    public void add(Employee employee) {
13        this.employees.add(employee);
14    }
15
16    public Employee get(int index) {
17        return this.employees.get(index);
18    }
19 }
```

CTO

```

1 public class CTO extends DisciplinaryLeadership {
2
3     @Override
4     public void draw() {
5         System.out.println(CTO.class.getSimpleName());
6         super.draw();
7     }
8 }
```

VP

```

1 public class VP extends DisciplinaryLeadership {
2
3     @Override
4     public void draw() {
5         System.out.println(String.format(" %s", VP.class.getSimpleName()));
6         super.draw();
7     }
8 }
```

Developer

```

1  public class Developer implements Employee {
2
3      @Override
4      public void draw() {
5          System.out.println(String.format(" %s", Developer.class.getSimpleName()));
6      }
7 }
```

Client

```

1  public class Client {
2      public static void main(String[] args) {
3
4          DisciplinaryLeadership cto = new CTO();
5          DisciplinaryLeadership vp1 = new VP();
6          DisciplinaryLeadership vp2 = new VP();
7
8          vp1.add(new Developer());
9          vp1.add(new Developer());
10
11         vp2.add(new Developer());
12         vp2.add(new Developer());
13         vp2.add(new Developer());
14
15         cto.add(vp1);
16         cto.add(vp2);
17
18         cto.draw();
19     }
20 }
```

Client output

```

1  CTO
2  VP
3  Developer
4  Developer
5  VP
6  Developer
7  Developer
8  Developer
```

Advantages:

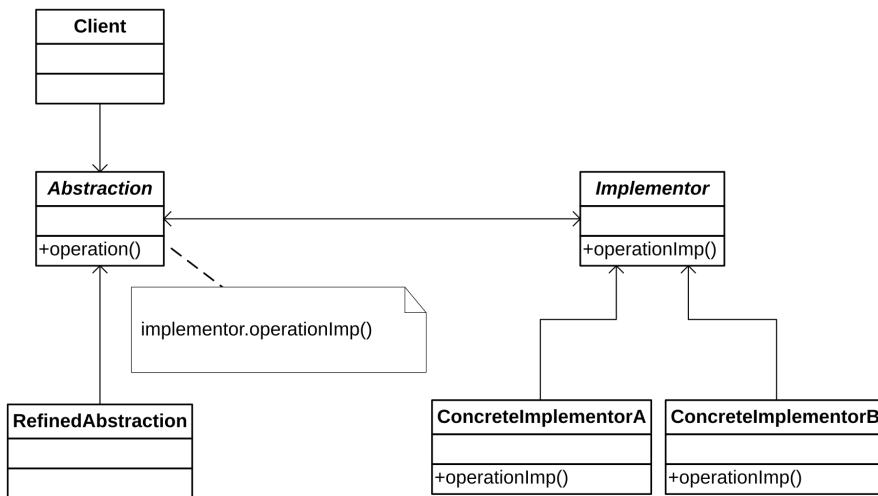
- Simple implementation of **nested** structures
- **Elegant working** with the tree structure. The client can easily traverse through the hierarchy, call operations and manage them, i.e., add new elements and delete existing ones.
- **Flexibility and expandability.** Easy addition of new elements (leafs or composites).

Disadvantages:

- **Generalisation of the draft.** If a particular compound element is to have only a fixed number of children, or only certain children, this can only be checked at runtime.

6.2.5 Bridge

Decouple an abstraction from its implementation so that the two can vary independently.



Use cases:

- Avoiding a permanent binding between an abstraction and its implementation.
- Combining different abstractions and implementations and extend them independently.
- Changes in the implementation of an abstraction should have no impact on clients.
- Sharing an implementation among multiple objects.
- Run-time binding of the implementation.

6.2.5.1 Example: Message

Message

```

1  public abstract class Message {
2
3      private final MessageSender messageSender;
4
5      public Message(MessageSender messageSender) {
6          this.messageSender = messageSender;
7      }
8
9      public MessageSender messageSender() {
10         return this.messageSender;
11     }
12
13     abstract void send();
14 }
```

EmailMessage

```
1 public class EmailMessage extends Message {  
2  
3     public EmailMessage(MessageSender messageSender) {  
4         super(messageSender);  
5     }  
6  
7     @Override  
8     public void send() {  
9         messageSender().sendMessage();  
10    }  
11 }
```

SmsMessage

```
1 public class SmsMessage extends Message {  
2  
3     public SmsMessage(MessageSender messageSender) {  
4         super(messageSender);  
5     }  
6  
7     @Override  
8     public void send() {  
9         messageSender().sendMessage();  
10    }  
11 }
```

MessageSender

```
1 public interface MessageSender {  
2  
3     void sendMessage();  
4 }
```

EmailMessageSender

```
1 public class EmailMessageSender implements MessageSender {  
2  
3     @Override  
4     public void sendMessage() {  
5         System.out.println("Sending email message...");  
6     }  
7 }
```

SmsMessageSender

```
1 public class SmsMessageSender implements MessageSender {  
2  
3     @Override  
4     public void sendMessage() {  
5         System.out.println("Sending sms message...");  
6     }  
7 }
```

Client

```
1 public class Client {  
2  
3     public static void main(String[] args) {  
4         sendSms();  
5         sendEmail();  
6     }  
7  
8     private static void sendSms() {  
9         MessageSender sender = new SmsMessageSender();  
10        Message message = new SmsMessage(sender);  
11        message.send();  
12    }  
13  
14    private static void sendEmail() {  
15        MessageSender sender = new EmailMessageSender();  
16        Message message = new EmailMessage(sender);  
17        message.send();  
18    }  
19 }
```

6.2.5.2 Example: Television

Font

```
1 public interface Television {  
2  
3     void on();  
4  
5     void off();  
6  
7     void channel(int channel);  
8 }
```

Samsung

```
1 public class Samsung implements Television {  
2  
3     @Override  
4     public void on() {  
5         System.out.println("On");  
6     }  
7  
8     @Override  
9     public void off() {  
10        System.out.println("Off");  
11    }  
12  
13     @Override  
14     public void channel(int channel) {  
15         System.out.println(channel);  
16     }  
17 }
```

Philips

```
1 public class Philips implements Television {  
2  
3     @Override  
4     public void on() {  
5         System.out.println("On");  
6     }  
7  
8     @Override  
9     public void off() {  
10        System.out.println("Off");  
11    }  
12  
13     @Override  
14     public void channel(int channel) {  
15         System.out.println(channel);  
16     }  
17 }
```

RemoteControl

```
1 public abstract class RemoteControl {  
2  
3     private final Television television;  
4  
5     public RemoteControl(Television television) {  
6         this.television = television;  
7     }  
8  
9     public void on() {  
10        this.television.on();  
11    }  
12  
13    public void off() {  
14        this.television.off();  
15    }  
16  
17    public void channel(int channel) {  
18        this.television.channel(channel);  
19    }  
20}
```

SmartRemoteControl

```
1 public class SmartRemoteControl extends RemoteControl {  
2  
3     private int channel;  
4  
5     public SmartRemoteControl(Television television) {  
6         super(television);  
7     }  
8  
9     public void nextChannel() {  
10        this.channel++;  
11        channel(this.channel);  
12    }  
13  
14    public void previousChannel() {  
15        this.channel--;  
16        channel(this.channel);  
17    }  
18}
```

Client

```
1 public class Client {  
2  
3     public static void main(String[] args) {  
4         useSamsung();  
5         usePhilips();  
6     }  
7  
8     private static void useSamsung() {  
9         Television television = new Samsung();  
10        SmartRemoteControl remoteControl = new SmartRemoteControl(television);
```

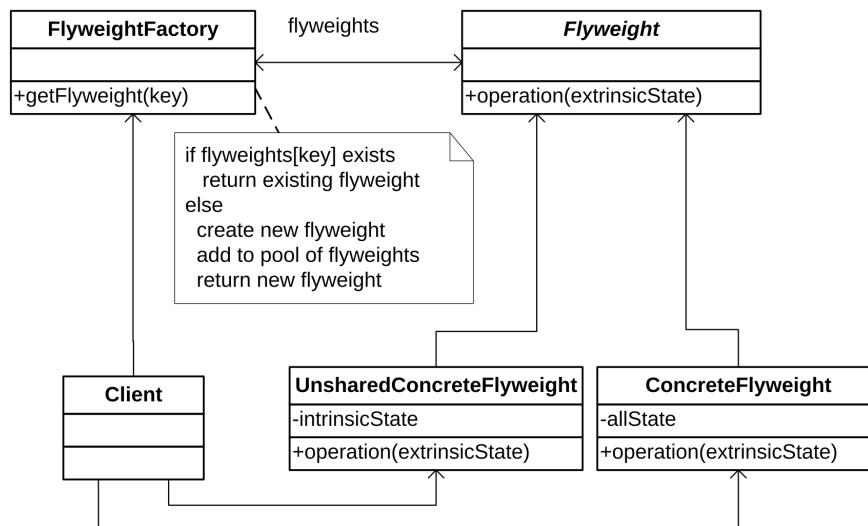
```
11     remoteControl.on();
12     remoteControl.nextChannel();
13     remoteControl.nextChannel();
14     remoteControl.previousChannel();
15     remoteControl.off();
16 }
17
18
19 private static void usePhilips() {
20     Television television = new Philips();
21     SmartRemoteControl remoteControl = new SmartRemoteControl(television);
22
23     remoteControl.on();
24     remoteControl.nextChannel();
25     remoteControl.nextChannel();
26     remoteControl.previousChannel();
27     remoteControl.off();
28 }
29 }
```

Advantages:

- Abstraction and implementation are **decoupled** and **independent**.
- Improved **extensibility**, you can extend the abstraction and implementation hierarchies independently.
- The implementation is also changed **dynamically at run time**.
- **Hiding** of implementation details from the client.

6.2.6 Flyweight

Use sharing to support large numbers of fine-grained objects efficiently.



Use cases:

- Need to create **large** number of objects, when memory cost is a constraint.
- When many of the object attributes can be made **external and shared**.
- Avoid unnecessary object **initialization**.
- Where the **identity** of the object is not a matter of concern and the required operations can be performed on the shared objects.

As defined by GoF, an object can have two states, the intrinsic and the extrinsic state:

Intrinsic state

Stored in the flyweight; it consists of information that's independent on the flyweights context, thereby making it shareable.

Extrinsic state

Depends on and varies with the flyweight's context and therefore can not be shared. Client objects are responsible for passing extrinsic state to the flyweight if necessary.

6.2.6.1 Example: Font

Font

```

1  public final class Font {
2
3      private final String name;
4      private final int size;
5
6      public Font(String name, int size) {
7          this.name = name;
8          this.size = size;
9      }
10
11     public String name() {
12         return this.name;
13     }
14
15     public int size() {
16         return this.size;
17     }
18
19     @Override
20     public int hashCode() {
21         return Objects.hash(this.name, this.size);
22     }
23
24     @Override
25     public boolean equals(Object obj) {
26         if (this == obj) {
27             return true;
28         }
29         if (obj == null) {
30             return false;
31         }
32         if (getClass() != obj.getClass()) {
33             return false;
34         }
35         Font other = (Font) obj;
36         return Objects.equals(this.name, other.name) && this.size == other.size;
37     }
38 }
```

FontFactory

```

1  public final class FontFactory {
2
3      private final Set<Font> fonts = new HashSet<>();
4
5      public Font of(String name, int size) {
6          for (Font font : this.fonts) {
7              if (font.name().equals(name) && font.size() == size) {
8                  return font;
9              }
10         }
11         Font font = new Font(name, size);
12 }
```

```
13     this.fonts.add(font);
14
15     return font;
16 }
17 }
```

Client

```
1 public class Client {
2
3     public static void main(String[] args) {
4         FontFactory factory = new FontFactory();
5
6         System.out.println(factory.of("Helvetica", 12));
7         System.out.println(factory.of("Arial ", 10));
8
9         // Will return same objects
10        System.out.println(factory.of("Helvetica", 12));
11        System.out.println(factory.of("Arial ", 10));
12    }
13 }
```

6.2.6.2 Example: City

Person

```
1  public final class Person {
2
3      private final String firstName;
4      private final String lastName;
5      private final String street;
6      private final City city;
7
8      public Person(String firstName, String lastName, String street, City city) {
9          this.firstName = firstName;
10         this.lastName = lastName;
11         this.street = street;
12         this.city = city;
13     }
14
15     public String firstName() {
16         return this.firstName;
17     }
18
19     public String lastName() {
20         return this.lastName;
21     }
22
23     public String street() {
24         return this.street;
25     }
26
27     public City city() {
28         return this.city;
29     }
30
31     @Override
32     public int hashCode() {
33         return Objects.hash(this.city, this.firstName, this.lastName, this.street);
34     }
35
36     @Override
37     public boolean equals(Object obj) {
38         if (this == obj) {
39             return true;
40         }
41         if (obj == null) {
42             return false;
43         }
44         if (getClass() != obj.getClass()) {
45             return false;
46         }
47
48         Person other = (Person) obj;
49         return Objects.equals(this.city, other.city)
50             && Objects.equals(this.firstName, other.firstName)
51             && Objects.equals(this.lastName, other.lastName)
52             && Objects.equals(this.street, other.street);
53     }
54 }
```

55 }

City

```
1  public final class City {
2
3      private final String country;
4      private final String countryCode;
5      private final String name;
6
7      public City(String country, String countryCode, String name) {
8          this.country = country;
9          this.countryCode = countryCode;
10         this.name = name;
11     }
12
13     public String country() {
14         return this.country;
15     }
16
17     public String countryCode() {
18         return this.countryCode;
19     }
20
21     public String name() {
22         return this.name;
23     }
24
25     @Override
26     public int hashCode() {
27         return Objects.hash(this.name, this.countryCode);
28     }
29
30     @Override
31     public boolean equals(Object obj) {
32         if (this == obj) {
33             return true;
34         }
35         if (obj == null) {
36             return false;
37         }
38         if (getClass() != obj.getClass()) {
39             return false;
40         }
41         City other = (City) obj;
42         return Objects.equals(this.name, other.name)
43             && Objects.equals(this.countryCode, other.countryCode);
44     }
45
46 }
```

CityFactory

```

1  public final class CityFactory {
2
3      private final Set<City> cities = new HashSet<>();
4
5      public City of(String country, String countryCode, String name) {
6          for (City city : this.cities) {
7              if (city.country().equals(country)
8                  && city.countryCode().equals(countryCode)
9                  && city.name().equals(name)) {
10                  return city;
11              }
12          }
13
14          City city = new City(country, countryCode, name);
15          this.cities.add(city);
16
17          return city;
18      }
19  }

```

Employee

```

1  public class Client {
2
3      public static void main(String[] args) {
4          CityFactory factory = new CityFactory();
5
6          // Will use same city instance
7          new Person("Forster", "Gamlen", "7654 Ruskin Center",
8                  factory.of("United States", "US", "Carson City"));
9          new Person("Nevin", "Roddell", "5 Del Sol Alley",
10                 factory.of("United States", "US", "Carson City"));
11          new Person("Hunter", "Lewins", "33733 Heffernan Circle",
12                 factory.of("United States", "US", "Carson City"));
13
14          new Person("Reube", "Gregoretti", "08 Melrose Street",
15                  factory.of("Japan", "JP", "Mutsu"));
16      }
17  }

```

Advantages:

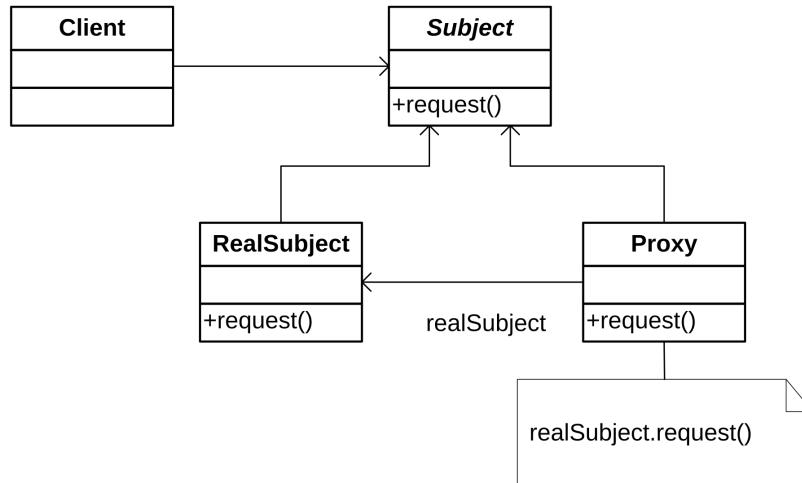
- Improves the performance by **reducing** the number of objects.
- **Reduces the memory footprint** as the common properties are shared between objects using intrinsic attributes.

Disadvantages:

- If there are **no shareable attributes** in an object, the pattern is of no use.
- May introduce **run-time costs** associated with transferring, finding, and/or computing extrinsic state, especially if it was formerly stored as intrinsic state.

6.2.7 Proxy

Provide a surrogate or placeholder for another object to control access to it.



Use cases:

- Proxies are often used in situations where a more complex reference to an object is required instead of a normal pointer. A distinction is usually made between the following types of proxies:
 - **Remote proxy** provide a local representation of another remote object or resource.
 - **Virtual proxy** wrap expensive objects and loads them on-demand.
 - **Protection proxy** provide access control to an original object.
 - **Smart reference** executes additional operations when an object is accessed.

6.2.7.1 Example: Spaceship

Spaceship

```

1 interface Spaceship {
2     void fly();
3 }
  
```

Pilot

```

1 public final class Pilot {
2
3     private final String name;
4
5     public Pilot(String name) {
6         this.name = name;
7     }
8
9     public String name() {
10        return this.name;
11    }
12 }
  
```

MillenniumFalcon

```
1 public final class MillenniumFalcon implements Spaceship {  
2  
3     @Override  
4     public void fly() {  
5         System.out.println("Welcome, Han!");  
6     }  
7 }
```

MillenniumFalconProxy

```
1 public final class MillenniumFalconProxy implements Spaceship {  
2  
3     private final Pilot pilot;  
4     private final Spaceship ship;  
5  
6     public MillenniumFalconProxy(Pilot pilot) {  
7         this.pilot = pilot;  
8         this.ship = new MillenniumFalcon();  
9     }  
10  
11    @Override  
12    public void fly() {  
13        if ("Han Solo".equals(this.pilot.name())) {  
14            this.ship.fly();  
15        } else {  
16            System.out.printf("Sorry %s, only Han Solo can fly the Falcon!\n", this.pilot.name());  
17        }  
18    }  
19 }
```

Client

```
1 public class Client {  
2  
3     public static void main(String[] args) {  
4         Spaceship ship = new MillenniumFalconProxy(new Pilot("Han Solo"));  
5         ship.fly();  
6  
7         ship = new MillenniumFalconProxy(new Pilot("Darth Vader"));  
8         ship.fly();  
9     }  
10 }
```

6.2.7.2 Example: ImageViewer

ImageViewer

```

1 interface ImageViewer {
2     void display();
3 }
```

DefaultImageViewer

```

1 public final class DefaultImageViewer implements ImageViewer {
2
3     private final Image image;
4
5     public DefaultImageViewer(String path) {
6         // Slow operation
7         this.image = Image.load(path);
8     }
9
10    @Override
11    public void display() {
12        // Slow operation
13        this.image.display();
14    }
15 }
```

Image

```

1 public interface Image {
2
3     void display();
4
5     static Image load(String path) {
6         // Not implemented
7         return null;
8     }
9 }
```

ImageViewerProxy

```

1 public final class ImageViewerProxy implements ImageViewer {
2
3     private final String path;
4     private ImageViewer viewer;
5
6     public ImageViewerProxy(String path) {
7         this.path = path;
8     }
9
10    @Override
11    public void display() {
12        if (this.viewer == null) {
13            this.viewer = new DefaultImageViewer(this.path);
14        }
15
16        this.viewer.display();
17    }
18 }
```

Client

```
1 public class Client {
2
3     public static void main(String[] args) {
4         ImageViewer beer = new ImageViewerProxy("./img/beer.png");
5         ImageViewer hammock = new ImageViewerProxy("./img/hammock.png");
6
7         beer.display();
8         beer.display(); // DefaultImageViewer will be call and be executed once
9
10        hammock.display();
11    }
12 }
```

Advantages:

- **Security:** Certain conditions can be checked when accessing the object.
- **Performance:** Objects with expensive operations demanding in terms of memory and execution time, can be wrapped so that they are called only when they are really needed, or unnecessary instantiation can be avoided.

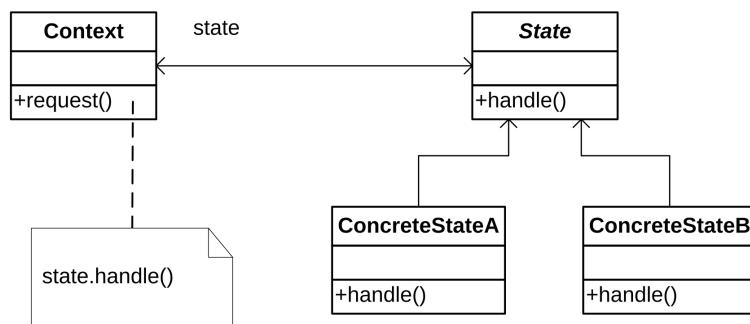
Disadvantages:

- **Performance:** Can also be a disadvantage of the proxy pattern if a proxy object is used to wrap an object that exists somewhere on the network. Since it is a proxy, it can hide from the client the fact that it is a remote communication.
- Add another layer of abstraction, which can lead to detours and increase complexity.

6.3 Behavioural

6.3.1 State

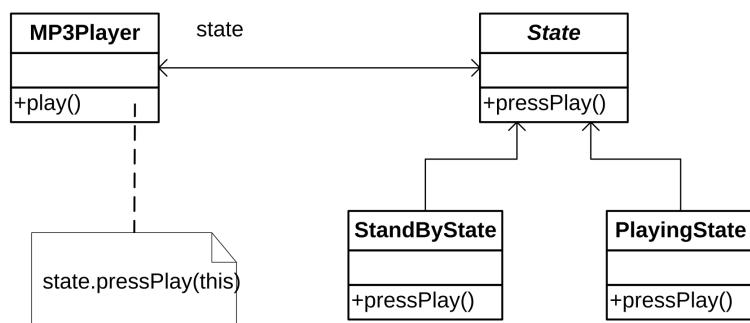
Allow an object to alter its behaviour when its internal state changes. The object will appear to change its class.



Use cases:

- An object should change its external behaviour at runtime depending on its state.
 - **State machines** e.g., parser. The functionality of many text parsers is inevitably state based. So, a compiler parsing source code has to interpret a character depending on the characters read before.
 - Representation of states of network connections

6.3.1.1 Example: MP3Player



MP3Player

```

1  public class MP3Player {
2      private State state;
3
4      private MP3Player(State state) {
5          this.state = state;
6      }
7      public void play() {
8          state.pressPlay(this);
9      }
10     public void state(State state) {
11         this.state = state;
12     }
13     public State state() {
14         return state;
15     }
16 }
```

State

```

1  public interface State {
2      void pressPlay(MP3Player player);
3 }
```

PlayingState

```

1  public class PlayingState implements State {
2      public void pressPlay(MP3Player player) {
3          player.state(new StandByState());
4      }
5 }
```

StandByState

```

1  public class StandByState implements State {
2      public void pressPlay(MP3Player player) {
3          player.state(new PlayingState());
4      }
5 }
```

Client

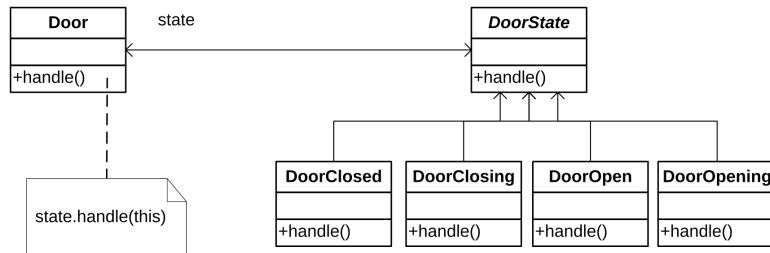
```

1  public class Client {
2      public static void main(String[] args) {
3          MP3Player player = new MP3Player(new PlayingState());
4          player.play();
5          player.play();
6          player.play();
7          player.play();
8          player.play();
9          player.play();
10         player.play();
11     }
12 }
```

Output

```
1 Play state
2 Standby state
3 Play state
4 Standby state
5 Play state
6 Standby state
7 Play state
```

6.3.1.2 Example: Door



Door

```

1  public class Door {
2
3      private DoorState state;
4
5      public Door(DoorState state) {
6          this.state = state;
7      }
8
9      public void handle() {
10         this.state.handle(this);
11     }
12
13     public void state(DoorState state) {
14         this.state = state;
15     }
16
17     public DoorState state() {
18         return this.state;
19     }
20 }
  
```

DoorState

```

1  public interface DoorState {
2      void handle(Door door);
3  }
  
```

DoorClosed

```

1  public class DoorClosed implements DoorState {
2
3      public DoorClosed() {
4          System.out.println("Door closed");
5      }
6
7      @Override
8      public void handle(Door door) {
9          door.state(new DoorOpening());
10     }
11 }
  
```

DoorClosing

```
1 public class DoorClosing implements DoorState {  
2  
3     public DoorClosing() {  
4         System.out.println("Door closing");  
5     }  
6  
7     @Override  
8     public void handle(Door door) {  
9         door.state(new DoorClosed());  
10    }  
11 }
```

DoorOpen

```
1 public class DoorOpen implements DoorState {  
2  
3     public DoorOpen() {  
4         System.out.println("Door open");  
5     }  
6  
7     @Override  
8     public void handle(Door door) {  
9         door.state(new DoorClosing());  
10    }  
11 }
```

DoorOpening

```
1 public class DoorOpening implements DoorState {  
2  
3     public DoorOpening() {  
4         System.out.println("Door opening");  
5     }  
6  
7     @Override  
8     public void handle(Door door) {  
9         door.state(new DoorOpen());  
10    }  
11 }
```

Client

```
1 public class Client {  
2     public static void main(String[] args) {  
3         Door door = new Door(new DoorOpen());  
4         door.handle();  
5         door.handle();  
6         door.handle();  
7         door.handle();  
8         door.handle();  
9         door.handle();  
10    }  
11 }
```

Output

```
1 Door open
2 Door closing
3 Door closed
4 Door opening
5 Door open
6 Door closing
7 Door closed
```

Advantages:

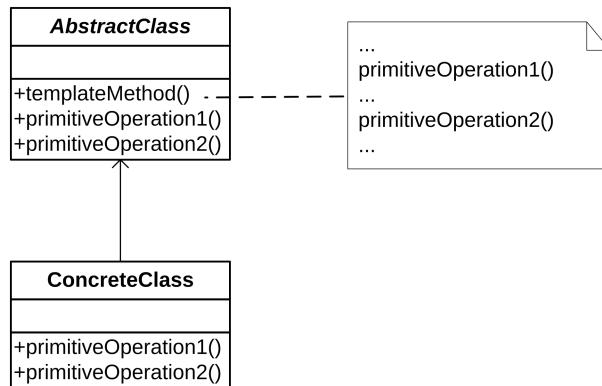
- **Extensibility and change stability.** Easy integration of new states without having to change existing code. Changes to state dependent behaviour only affect a state class and not the context.
- **Comprehensibility.** High cohesion and delegation of responsibilities (the behaviour of a state is encapsulated / localised in the corresponding state object itself) make the code easy to understand.
- **Explicit state transitions.** By introducing objects for each state, the state transition becomes explicit. In addition, inconsistent states are prevented, because a state change is an atomic command (setting a variable).
- The design is **less error-prone** with regard to later changes than broad, repetitive if-else constructs.

Disadvantages:

- **Increased number of classes.** Is less compact than a single class. However, the aim of the design pattern is precisely to distribute the behaviour of a single confusing class to several state objects.

6.3.2 Template Method

Define a skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.



Use cases:

- Define a template that implements similar algorithms and delegates the individual execution steps to subclasses.
- Common behaviour of subclasses should be moved to a superclass to avoid code duplication.
- Modelling of fixed sequences and the specification of variation points.

6.3.2.1 Example: Compiler

CrossCompiler

```

1  public abstract class CrossCompiler {
2      public final void crossCompile() {
3          collectSource();
4          compileToTarget();
5      }
6
7      // Template methods
8      protected abstract void collectSource();
9      protected abstract void compileToTarget();
10 }
  
```

IPhoneCompiler

```

1 public class IPhoneCompiler extends CrossCompiler {
2     protected void collectSource() {
3         // anything specific to this class
4     }
5
6     protected void compileToTarget() {
7         // iphone specific compilation
8     }
9 }
```

AndroidCompiler

```

1 public class AndroidCompiler extends CrossCompiler {
2     protected void collectSource() {
3         // anything specific to this class
4     }
5
6     protected void compileToTarget() {
7         // android specific compilation
8     }
9 }
```

Client

```

1 public class Client {
2     public static void main(String[] args) {
3         CrossCompiler iphone = new IPhoneCompiler();
4         iphone.crossCompile();
5
6         CrossCompiler android = new AndroidCompiler();
7         android.crossCompile();
8     }
9 }
```

6.3.2.2 Example: Callable**Callable**

```

1 interface Callable extends Runnable {
2     void beforeRun();
3     void afterRun();
4     void onError(Exception e);
5     void onSuccess();
6 }
```

AbstractCallbackRunnable

```
1 abstract class AbstractCallbackRunnable implements Callbackable {
2
3     private final LocalDateTime submittedTime;
4
5     private Status status = Status.UNKNOW;
6     private LocalDateTime startedTime;
7     private LocalDateTime finishedTime;
8
9     AbstractCallbackRunnable() {
10         this.submittedTime = LocalDateTime.now();
11         this.status = Status.WAITING;
12     }
13
14     @Override
15     public void run() {
16         try {
17             beforeRunIntern();
18             startRun();
19             onSuccessIntern();
20         } catch (Exception e) {
21             onErrorIntern(e);
22         } finally {
23             afterRunIntern();
24         }
25     }
26
27     public abstract void startRun() throws Exception;
28
29     private void beforeRunIntern() {
30         this.startedTime = LocalDateTime.now();
31         this.status = Status.RUNNING;
32
33         beforeRun();
34     }
35
36     private void afterRunIntern() {
37         this.finishedTime = LocalDateTime.now();
38         afterRun();
39     }
40
41     private void onErrorIntern(Exception e) {
42         this.status = Status.FAILED;
43         onError(e);
44     }
45
46     private void onSuccessIntern() {
47         this.status = Status.COMPLETED;
48         onSuccess();
49     }
50
51     public Status status() {
52         return this.status;
53     }
54
55     public LocalDateTime submittedTime() {
56         return this.submittedTime;
```

```
57     }
58
59     public LocalDateTime startedTime() {
60         return this.startedTime;
61     }
62
63     public LocalDateTime finishedTime() {
64         return this.finishedTime;
65     }
66 }
```

Status

```
1 public enum Status {
2     UNKNOW, WAITING, RUNNING, COMPLETED, FAILED
3 }
```

DefaultCallbackRunnable

```
1 public class DefaultCallbackRunnable extends AbstractCallbackRunnable {
2
3     @Override
4     public void beforeRun() {
5         System.out.println("beforeRun()");
6     }
7
8     @Override
9     public void afterRun() {
10        System.out.println("afterRun()");
11    }
12
13     @Override
14     public void onError(Exception e) {
15         System.out.println("onError()");
16     }
17
18     @Override
19     public void onSuccess() {
20         System.out.println("onSuccess()");
21     }
22
23     @Override
24     public void startRun() throws Exception {
25         System.out.println("startRun()");
26     }
27 }
```

Client

```
1 public class Client {  
2  
3     public static void main(String[] args) {  
4         ExecutorService executor = Executors.newFixedThreadPool(1);  
5         executor.submit(new DefaultCallbackRunnable());  
6     }  
7 }
```

Advantages:

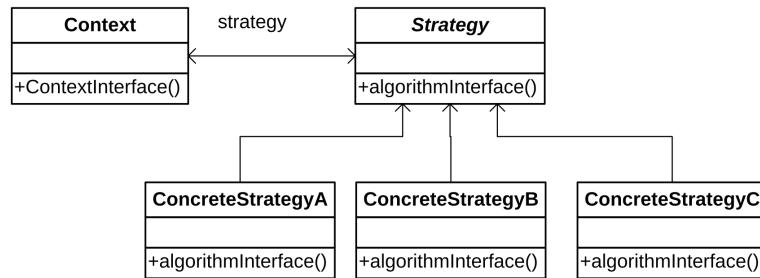
- The use of a template method allows inheriting classes to overwrite certain steps of an algorithm **without changing the structure of the algorithm.**
- Common behaviour is **encapsulated** in one class.
- **All methods** must be overwritten when implementing the subclass.
- Avoiding **duplicate** code.

Disadvantages:

- The design can become **unnecessarily complicated** if subclasses have to implement a large number of methods to make the algorithm more concrete.
- It can become a disadvantage if the algorithm follows a **fixed structure** that cannot be changed.
- It must be **clear** which methods may or must be overwritten.

6.3.3 Strategy

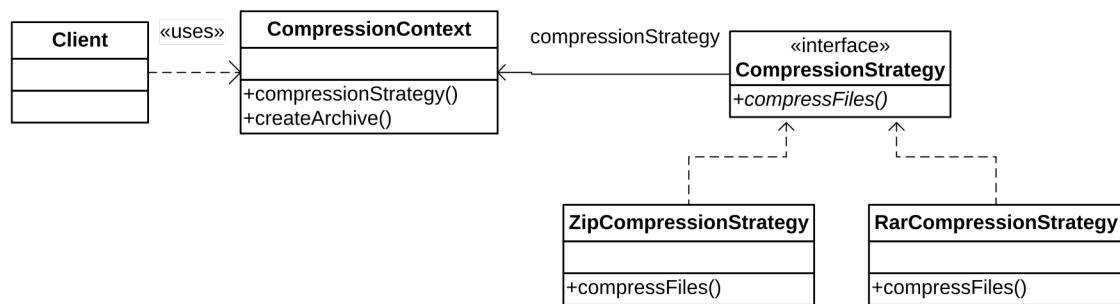
Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.



Use cases:

- For many similar classes that differ only in behaviour.
- For input masks, the logic for validation and plausibility checks of user inputs can be encapsulated.
- Behaviour should also be decoupled from the context if different forms of one and the same function are required
 - Sorting of a collection (`Array`, `List`), whereby the concrete strategies represent different sorting methods.
 - Storage in different file formats.
 - Packer with different compression algorithms.
- Decoupling and hiding complicated algorithm details from the context.
- Multiple branches (`if (...) else if (...) else ...`) can be avoided and this increases the clarity of the code.

6.3.3.1 Example: Compression



CompressionStrategy

```

1 public interface CompressionStrategy {
2     void compressFiles(List<File> files);
3 }
```

RarCompressionStrategy

```

1 public class RarCompressionStrategy implements CompressionStrategy {
2     public void compressFiles(List<File> files) {
3         // using RAR approach
4     }
5 }
```

ZipCompressionStrategy

```

1 public class ZipCompressionStrategy implements CompressionStrategy {
2     public void compressFiles(List<File> files) {
3         // using ZIP approach
4     }
5 }
```

CompressionContext

```

1 public class CompressionContext {
2     private CompressionStrategy strategy;
3
4     // this can be set at runtime by the application preferences
5     public void compressionStrategy(CompressionStrategy strategy) {
6         this.strategy = strategy;
7     }
8
9     // use the strategy
10    public void createArchive(List<File> files) {
11        this.strategy.compressFiles(files);
12    }
13 }
```

Client

```

1 public class Client {
2
3     public static void main(String[] args) {
4         CompressionContext ctx = new CompressionContext();
5
6         // we could assume context is already set by preferences
7         ctx.compressionStrategy(new RarCompressionStrategy());
8
9         // get a list of files
10
11         ctx.createArchive(files);
12     }
13 }
```

6.3.3.2 Example: LogFormatter

LogFormatter

```
1 interface LogFormatter {
2     String format(Log log);
3 }
```

FullLogFormatter

```
1 public class FullLogFormatter implements LogFormatter {
2
3     @Override
4     public String format(Log log) {
5         return String.format("Hash: %s%nAuthor: %s%nMessage: %s%nDate: %s%n",
6             log.hash(),
7             log.author(),
8             log.message(),
9             log.date());
10    }
11 }
```

ShortLogFormatter

```
1 public class ShortLogFormatter implements LogFormatter {
2
3     @Override
4     public String format(Log log) {
5         return String.format("Hash: %s%nAuthor: %s%nMessage: %s%n",
6             log.hash(),
7             log.author(),
8             log.message());
9    }
10 }
```

OnelineLogFormatter

```
1 public class OnelineLogFormatter implements LogFormatter {
2
3     @Override
4     public String format(Log log) {
5         return String.format("%s %s",
6             log.hash(),
7             log.message());
8    }
9 }
```

Log

```
1  public final class Log {  
2  
3      private final String hash;  
4      private final String author;  
5      private final String message;  
6      private final LocalDateTime date;  
7  
8      public Log(String hash, String author, String message, LocalDateTime date) {  
9          this.hash = hash;  
10         this.author = author;  
11         this.message = message;  
12         this.date = date;  
13     }  
14  
15     public String hash() {  
16         return this.hash;  
17     }  
18  
19     public String author() {  
20         return this.author;  
21     }  
22  
23     public String message() {  
24         return this.message;  
25     }  
26  
27     public LocalDateTime date() {  
28         return this.date;  
29     }  
30 }
```

GitConsole

```
1  public class GitConsole {  
2  
3      private final LogFormatter formatter;  
4  
5      public GitConsole(LogFormatter formatter) {  
6          this.formatter = formatter;  
7      }  
8  
9      public void printLog(List<Log> logs) {  
10         logs.forEach(log -> System.out.println(this.formatter.format(log)));  
11     }  
12 }
```

Client

```
1 public class Client {
2
3     public static void main(String[] args) {
4         GitConsole console = new GitConsole(new FullLogFormatter());
5
6         List<Log> logs = List.of(
7             new Log("397c670", "cleancoder", "Clean up", LocalDateTime.now()),
8             new Log("07aa4d5", "cleancoder", "Fix NPE in Log", LocalDateTime.now()));
9
10        console.printLog(logs);
11    }
12 }
```

Advantages:

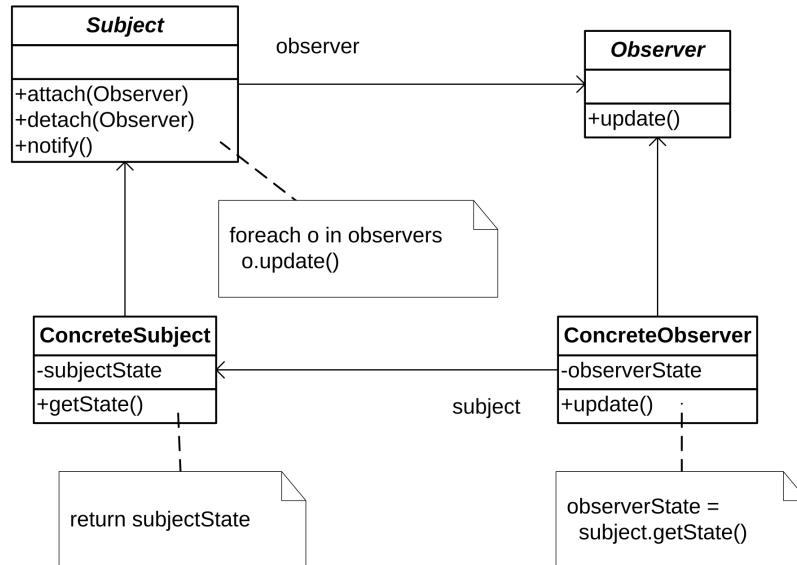
- **Reusability and decoupling of context and behaviour.** A family of algorithms is created, which can be used context-independently. On the one hand, new context objects can use the existing strategies.
- **Dynamic behaviour.** The behaviour of the context can be changed at runtime using appropriate setters.
- It is possible to choose from different implementations, thus increasing flexibility and reusability.
- **Multiple branches** can be avoided and this increases the overview of the code.
- **Alternative implementation.** Also, the same function (e.g., sorting) can be offered by different implementations, which differ in non-functional aspects (performance, memory requirements).

Disadvantages:

- Clients need to know the different strategies in order to choose between them and initialise the context.
 - Problem can be solved with a Factory. This allows the creation logic to be outsourced from the client to a factory.

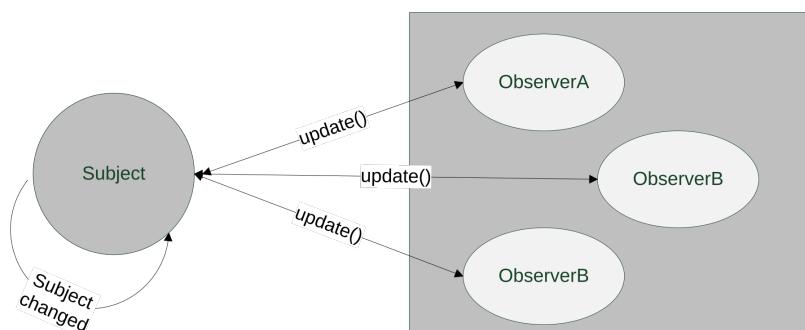
6.3.4 Observer

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

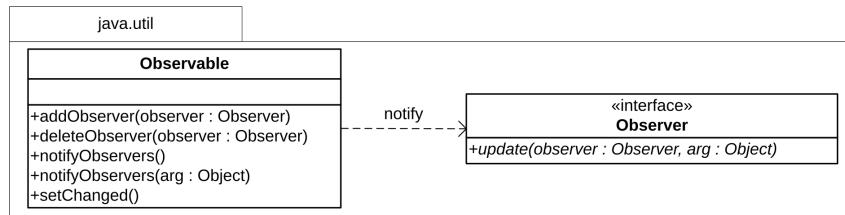


Use cases:

- When changing an object makes it necessary to modify one or more objects.
 - GUIs (user changes data, new data must be updated in all GUI components).
 - MVC patterns (Model-View-Controller) in view-model communication.
 - Every second of a timer pulse, both the digital clock and the analogue clock must be updated.
- Objects should notify other objects without knowing more about the object to be notified.
- Enables loose coupling of objects.



6.3.4.1 Example: DataStore



Screen

```

1 public class Screen implements Observer {
2     public void update(Observable o, Object arg) {
3         // act on the update
4     }
5 }

```

DataStore

```

1 public class DataStore extends Observable {
2     private String data;
3
4     public String data() {
5         return data;
6     }
7
8     public void data(String data) {
9         this.data = data;
10        // mark the observable as changed
11        setChanged();
12    }
13 }

```

Client

```

1 public class Client {
2     public static void main(String[] args) {
3         Screen screen = new Screen();
4         DataStore dataStore = new DataStore();
5
6         // register observer
7         dataStore.addObserver(screen);
8
9         // do something with dataStore
10
11        // send a notification
12        dataStore.notifyObservers();
13    }
14 }

```



The `Observable` class and the `Observer` interface have been deprecated in Java 9. The event model supported by `Observer` and `Observable` is quite limited, the order of notifications delivered by `Observable` is unspecified, and state changes are not in one-for-one correspondence with notifications.

6.3.4.2 Example: Influencer

Subject

```
1 public interface Subject {  
2  
3     void register(Observer observer);  
4  
5     void unregister(Observer observer);  
6  
7     void notifyAllObservers(Object message);  
8  
9 }
```

Observer

```
1 interface Observer {  
2     void update(Object message);  
3 }
```

Influencer

```
1 public class Influencer implements Subject {  
2  
3     private final Queue<Observer> followers = new ConcurrentLinkedQueue<>();  
4  
5     private final String name;  
6  
7     public Influencer(String name) {  
8         this.name = name;  
9     }  
10  
11     @Override  
12     public void register(Observer follower) {  
13         this.followers.add(follower);  
14     }  
15  
16     @Override  
17     public void unregister(Observer follower) {  
18         this.followers.remove(follower);  
19     }  
20  
21     @Override  
22     public void notifyAllObservers(Object message) {  
23         this.followers.forEach(follower -> follower.update(message));  
24     }  
25 }
```

Follower

```

1  public class Follower implements Observer {
2
3      private final String follower;
4
5      public Follower(String follower) {
6          this.follower = follower;
7      }
8
9      @Override
10     public void update(Object message) {
11         System.out.println(String.format("%s received message: %s", this.follower, message));
12     }
13 }
```

Client

```

1  public class Client {
2
3      public static void main(String[] args) {
4          Influencer influencer = new Influencer("Java");
5
6          influencer.register(new Follower("PHP"));
7          influencer.register(new Follower("QBasic"));
8          influencer.register(new Follower("TypeScript"));
9
10         influencer.notifyAllObservers("Java rocks!");
11     }
12 }
```

Advantages:

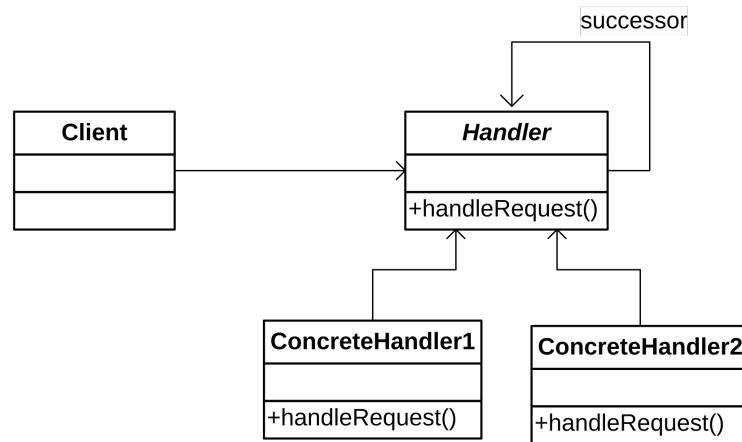
- **Consistency** of condition. Automatic adjustment of the observer state when the subject changes.
- **Reusability**. The Observer Pattern allows subject and observer to be varied independently.
- **Easy extension** of different observers that observe a single subject.

Disadvantages:

- **Update cascades and cycles**. In complex systems with many subjects and observers, update cascades can easily occur because the observers do not know about each other and cannot estimate the consequences of a single modification to a subject.

6.3.5 Chain of Responsibility

Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.



Use cases:

- Providing a low coupling between an object that should be handled by potential handler objects.
- Where a request can be handled by more than one object.
- Processing a request at runtime in a dynamically sequential order.

6.3.5.1 Example: Purchase

Purchase

```

1  public final class Purchase {
2
3      private final double amount;
4
5      public Purchase(double amount) {
6          this.amount = amount;
7      }
8
9      public double amount() {
10         return this.amount;
11     }
12 }
```

PurchaseAuthorizeFlow

```
1 public interface PurchaseAuthorizeFlow {
2     void next(Employee nextEmployee);
3 }
```

Employee

```
1 public interface Employee extends PurchaseAuthorizeFlow {
2     void authorize(Purchase purchase);
3 }
```

AbstractEmployee

```
1 public abstract class AbstractEmployee implements Employee {
2
3     private Employee nextEmployee;
4
5     @Override
6     public void next(Employee nextEmployee) {
7         this.nextEmployee = nextEmployee;
8     }
9
10    public Employee next() {
11        return this.nextEmployee;
12    }
13 }
```

CTO

```
1 public class CTO extends AbstractEmployee {
2
3     @Override
4     public void authorize(Purchase purchase) {
5         if (purchase.amount() > 100_000d) {
6             System.out.println("CTO");
7         } else {
8             next().authorize(purchase);
9         }
10    }
11 }
```

VP

```
1 public class VP extends AbstractEmployee {
2
3     @Override
4     public void authorize(Purchase purchase) {
5         if (purchase.amount() > 10_000d && purchase.amount() < 100_000d) {
6             System.out.println("VP");
7         } else {
8             next().authorize(purchase);
9         }
10    }
11 }
```

TeamLead

```
1 public class TeamLead extends AbstractEmployee {
2
3     @Override
4     public void authorize(Purchase purchase) {
5         if (purchase.amount() <= 10_000d) {
6             System.out.println("TeamLead");
7         } else {
8             next().authorize(purchase);
9         }
10    }
11 }
```

Client

```
1 public class Client {
2
3     public static void main(String[] args) {
4         Employee cto = new CTO();
5         Employee vp = new VP();
6         Employee teamLead = new TeamLead();
7
8         teamLead.next(vp);
9         vp.next(cto);
10
11         teamLead.authorize(new Purchase(100));      // TeamLead
12         teamLead.authorize(new Purchase(99_999)); // VP
13         teamLead.authorize(new Purchase(500_000)); // CTO
14     }
15 }
```

6.3.5.2 Example: Authentication

Authentication

```
1 public interface Authentication {  
2 }
```

AuthenticationFilter

```
1 public interface AuthenticationFilter {  
2     boolean isAuthenticated(Authentication authentication);  
3     void setNextFilter(AuthenticationFilter nextFilter);  
4 }
```

AbstractAuthenticationFilter

```
1 public abstract class AbstractAuthenticationFilter implements AuthenticationFilter {  
2  
3     private AuthenticationFilter nextFilter;  
4  
5     @Override  
6     public void setNextFilter(AuthenticationFilter nextFilter) {  
7         this.nextFilter = nextFilter;  
8     }  
9  
10    public boolean nextFilter(Authentication authentication) {  
11        if (this.nextFilter != null) {  
12            return this.nextFilter.isAuthenticated(authentication);  
13        }  
14  
15        return false;  
16    }  
17 }
```

BasicAuthenticationFilter

```
1 public class BasicAuthenticationFilter extends AbstractAuthenticationFilter {  
2  
3     @Override  
4     public boolean isAuthenticated(Authentication authentication) {  
5         if (authentication instanceof BasicAuthentication) {  
6             // logic  
7             return true;  
8         }  
9  
10        return nextFilter(authentication);  
11    }  
12 }
```

BasicAuthentication

```
1 public class BasicAuthentication implements Authentication {  
2 }
```

BearerAuthenticationFilter

```
1 public class BearerAuthenticationFilter extends AbstractAuthenticationFilter {  
2  
3     @Override  
4     public boolean isAuthenticated(Authentication authentication) {  
5         if (authentication instanceof BearerAuthentication) {  
6             // logic  
7             return true;  
8         }  
9  
10        return nextFilter(authentication);  
11    }  
12 }
```

BearerAuthentication

```
1 public class BearerAuthentication implements Authentication {  
2 }
```

DigestAuthenticationFilter

```
1 public class DigestAuthenticationFilter extends AbstractAuthenticationFilter {  
2  
3     @Override  
4     public boolean isAuthenticated(Authentication authentication) {  
5         if (authentication instanceof DigestAuthentication) {  
6             // logic  
7             return true;  
8         }  
9  
10        return nextFilter(authentication);  
11    }  
12 }
```

DigestAuthentication

```
1 public class DigestAuthentication implements Authentication {  
2 }
```

Client

```
1 public class Client {  
2  
3     public static void main(String[] args) {  
4         AuthenticationFilter digestFilter = new DigestAuthenticationFilter();  
5         AuthenticationFilter bearerFilter = new BearerAuthenticationFilter();  
6         AuthenticationFilter basicFilter = new BasicAuthenticationFilter();  
7  
8         digestFilter.setNextFilter(bearerFilter);  
9         bearerFilter.setNextFilter(basicFilter);  
10  
11         System.out.println(digestFilter.isAuthenticated(new BearerAuthentication())); // true  
12     }  
13 }
```

Advantages:

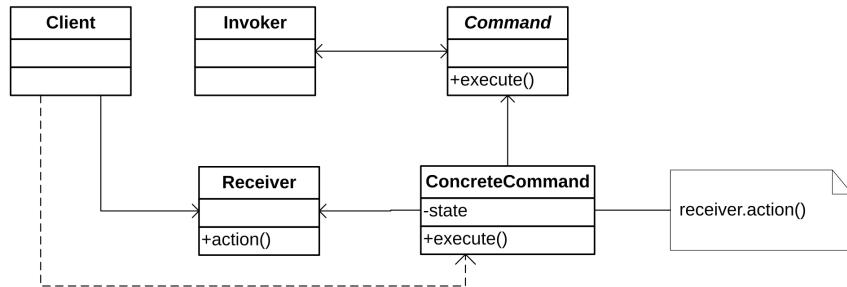
- The object does not need to know the structure of the chain, which decouples the sender of the request and its recipients.
- Simplifies your object as it does not need to know the structure of the chain and does not need to hold a direct reference to its members.
- Allows responsibilities to be added and removed dynamically by changing the members or the order of the chain.

Disadvantages:

- It is not guaranteed that objects are handled in the chain's flow, but this can also be seen as an advantage.
- It can be difficult to observe and debug the runtime properties.

6.3.6 Command

Encapsulates a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.



Use cases:

- When you want to **queue** operations or **schedule** their execution.
- Implementing **undo** or **redo** operations.

6.3.6.1 Example: FileSystem

Command

```

1 public interface Command {
2     void execute(FileSystem fs);
3 }
```

CreateCommand

```

1 public class CreateCommand implements Command {
2
3     private final File file;
4
5     public CreateCommand(File file) {
6         this.file = file;
7     }
8
9     @Override
10    public void execute(FileSystem fs) {
11        fs.create(this.file);
12    }
13 }
```

DeleteCommand

```
1 public class DeleteCommand implements Command {  
2  
3     private final File file;  
4  
5     public DeleteCommand(File file) {  
6         this.file = file;  
7     }  
8  
9     @Override  
10    public void execute(FileSystem fs) {  
11        fs.delete(this.file);  
12    }  
13}
```

MoveCommand

```
1 public class MoveCommand implements Command {  
2  
3     private final File source;  
4     private final File target;  
5  
6     public MoveCommand(File source, File target) {  
7         this.source = source;  
8         this.target = target;  
9     }  
10  
11    @Override  
12    public void execute(FileSystem fs) {  
13        fs.move(this.source, this.target);  
14    }  
15}
```

FileSystem

```
1 public class FileSystem {  
2  
3     public void create(File file) {  
4         System.out.println(String.format("Create %s",  
5             file.getName()));  
6     }  
7  
8     public void delete(File file) {  
9         System.out.println(String.format("Delete %s",  
10            file.getName()));  
11    }  
12  
13    public void move(File source, File target) {  
14        System.out.println(String.format("Move %s to %s",  
15            source.getName(),  
16            target.getName()));  
17    }  
18}
```

BatchFileSystemExecuter

```
1 public class BatchFileSystemExecuter {
2
3     private final FileSystem fs;
4
5     public BatchFileSystemExecuter(FileSystem fs) {
6         this.fs = fs;
7     }
8
9     public void execute(List<Command> commands) {
10        commands.forEach(c -> c.execute(this.fs));
11    }
12 }
```

Client

```
1 public class Client {
2
3     public static void main(String[] args) {
4         FileSystem fs = new FileSystem();
5         BatchFileSystemExecuter executer = new BatchFileSystemExecuter(fs);
6
7         List<Command> commands = List.of(
8             new CreateCommand(new File("file.tmp")),
9             new DeleteCommand(new File("file.tmp")),
10            new CreateCommand(new File("secret.txt")),
11            new MoveCommand(new File("secret.txt"), new File("topsecret.txt")));
12
13         executer.execute(commands);
14     }
15 }
```

Output

```
1 Create file.tmp
2 Delete file.tmp
3 Create secret.txt
4 Move secret.txt to topsecret.txt
```

6.3.6.2 Example: Television

Command

```
1 public interface Command {  
2     void execute();  
3 }
```

TurnOn

```
1 public class TurnOn implements Command {  
2  
3     private final Television device;  
4  
5     public TurnOn(Television device) {  
6         this.device = device;  
7     }  
8  
9     @Override  
10    public void execute() {  
11        this.device.on();  
12    }  
13 }
```

TurnOff

```
1 public class TurnOff implements Command {  
2  
3     private final Television device;  
4  
5     public TurnOff(Television device) {  
6         this.device = device;  
7     }  
8  
9     @Override  
10    public void execute() {  
11        this.device.off();  
12    }  
13 }
```

VolumeUp

```
1 public class VolumeUp implements Command {  
2  
3     private final Television device;  
4  
5     public VolumeUp(Television device) {  
6         this.device = device;  
7     }  
8  
9     @Override  
10    public void execute() {  
11        this.device.volumeUp();  
12    }  
13 }
```

VolumeDown

```
1 public class VolumeDown implements Command {  
2  
3     private final Television device;  
4  
5     public VolumeDown(Television device) {  
6         this.device = device;  
7     }  
8  
9     @Override  
10    public void execute() {  
11        this.device.volumenDown();  
12    }  
13}
```

Television

```
1 public class Television {  
2  
3     private int volume = 0;  
4  
5     public void on() {  
6         System.out.println("TV is on");  
7     }  
8  
9     public void off() {  
10        System.out.println("TV is off");  
11    }  
12  
13    public void volumeUp() {  
14        this.volume++;  
15  
16        System.out.println("Volume: " + this.volume);  
17    }  
18  
19    public void volumenDown() {  
20        this.volume--;  
21  
22        System.out.println("Volume: " + this.volume);  
23    }  
24}
```

DeviceButton

```
1 public class DeviceButton {  
2  
3     private Command command;  
4  
5     public void command(Command command) {  
6         this.command = command;  
7     }  
8  
9     public void press() {  
10        this.command.execute();  
11    }  
12}
```

Client

```
1 public class Client {  
2  
3     public static void main(String[] args) {  
4         Television television = new Television();  
5  
6         DeviceButton power = new DeviceButton();  
7         DeviceButton volume = new DeviceButton();  
8  
9         power.command(new TurnOn(television));  
10        power.press();  
11  
12        volume.command(new VolumeUp(television));  
13        volume.press();  
14        volume.press();  
15        volume.press();  
16  
17        volume.command(new VolumeDown(television));  
18        volume.press();  
19  
20        power.command(new TurnOff(television));  
21        power.press();  
22    }  
23 }
```

Output

```
1 TV is on  
2 Volume: 1  
3 Volume: 2  
4 Volume: 3  
5 Volume: 2  
6 TV is off
```

Advantages:

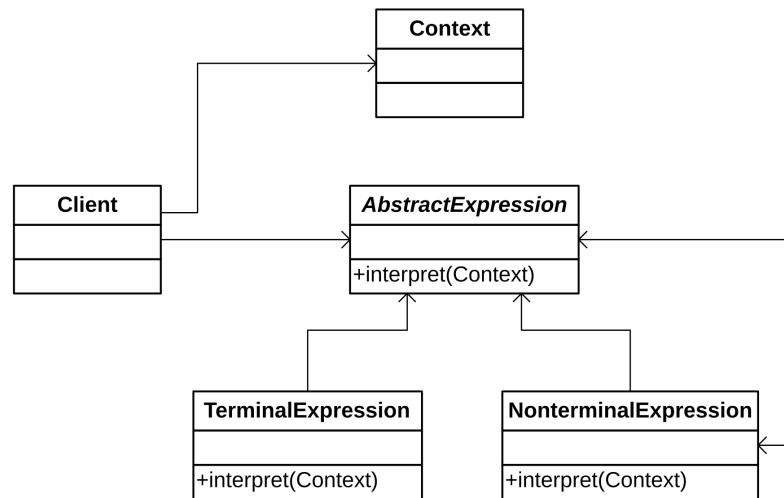
- New commands can be added easily without changing existing code.
- Possibility of implementing **undo** and **redo** operation.
- You can implement a **sequence** of operations.

Disadvantages:

- Large number of classes and objects working together to achieve a desired goal.

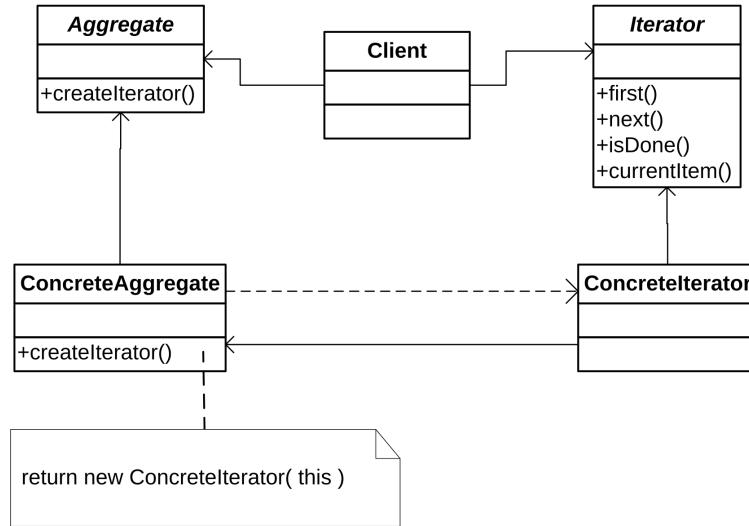
6.3.7 Interpreter

Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.



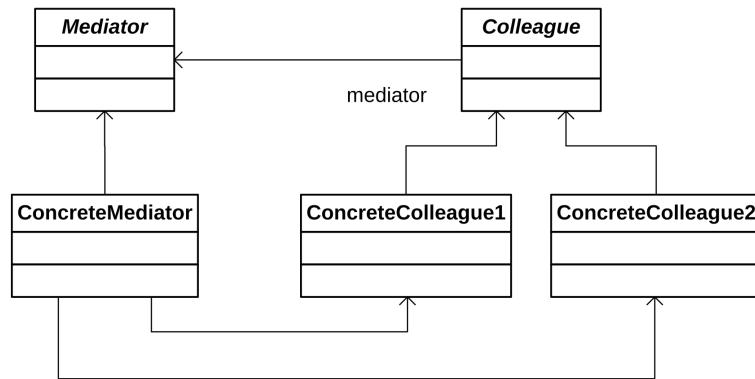
6.3.8 Iterator

Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.



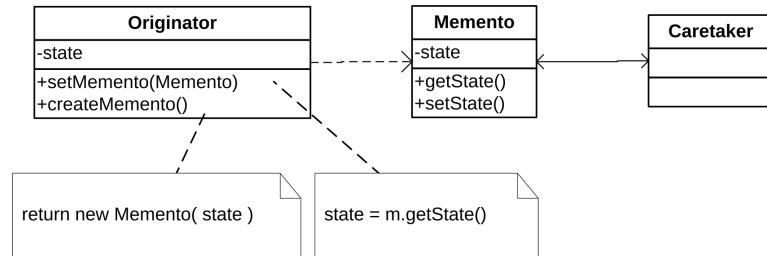
6.3.9 Mediator

Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.



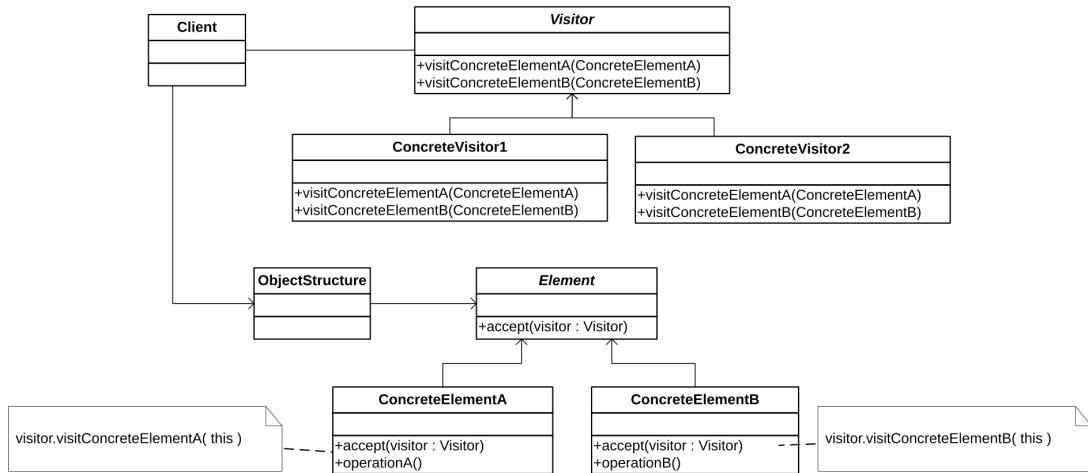
6.3.10 Memento

Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.



6.3.11 Visitor

Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.



Use cases:

- Lets you define a new operation without changing the classes of the elements on which it operates.
- Similar operations must be performed on objects of different types across a heterogeneous class hierarchy.

6.3.11.1 Example: Fridge

FridgeElement

```

1 public interface FridgeElement {
2     void accept(Visitor visitor);
3 }
  
```

Visitor

```

1 public interface Visitor {
2
3     void visit(Beer beer);
4
5     void visit(Milk milk);
6
7     void visit(Butter butter);
8
9 }
  
```

Beer

```
1 public class Beer implements FridgeElement {  
2  
3     @Override  
4     public void accept(Visitor visitor) {  
5         visitor.visit(this);  
6     }  
7  
8     LocalDate bestBeforeDate() {  
9         return LocalDate.of(2022, 1, 28);  
10    }  
11 }
```

Butter

```
1 public class Butter implements FridgeElement {  
2  
3     @Override  
4     public void accept(Visitor visitor) {  
5         visitor.visit(this);  
6     }  
7  
8     LocalDate expiryDate() {  
9         return LocalDate.of(2021, 10, 2);  
10    }  
11 }
```

Milk

```
1 public class Milk implements FridgeElement {  
2  
3     @Override  
4     public void accept(Visitor visitor) {  
5         visitor.visit(this);  
6     }  
7  
8     Date bestBeforeDate() {  
9         return new Date(2021 - 1900, 0, 1);  
10    }  
11 }
```

BestBeforeDateVisitor

```
1 public class BestBeforeDateVisitor implements Visitor {  
2  
3     private final LocalDate compareDate;  
4  
5     public BestBeforeDateVisitor(LocalDate compareDate) {  
6         this.compareDate = compareDate;  
7     }  
8  
9     @Override  
10    public void visit(Beer beer) {  
11        checkBestBeforeDate(beer, beer.bestBeforeDate()); // method differ
```

```

12     }
13
14     @Override
15     public void visit(Milk milk) {
16         checkBestBeforeDate(milk, milk.bestBeforeDate() // method differ
17             .toInstant()
18             .atZone(ZoneId.systemDefault())
19             .toLocalDate());
20     }
21
22     @Override
23     public void visit(Butter butter) {
24         checkBestBeforeDate(butter, butter.expiryDate()); // method differ
25     }
26
27     private void checkBestBeforeDate(FridgeElement element, LocalDate bestBeforeDate) {
28         if (bestBeforeDate.isAfter(this.compareDate)) {
29             System.out.println(element.getClass().getSimpleName());
30         }
31     }
32 }
```

Client

```

1  public class Client {
2
3      public static void main(String[] args) {
4          List<FridgeElement> fridge = List.of(
5              new Beer(),
6              new Beer(),
7              new Milk(),
8              new Butter());
9
10         Visitor visitor = new BestBeforeDateVisitor(LocalDate.of(2021, 1, 1));
11
12         System.out.println("Over the best before date:");
13
14         for (FridgeElement element : fridge) {
15             element.accept(visitor);
16         }
17     }
18 }
```

Client output

```

1 Over the best before date:
2 Beer
3 Beer
4 Butter
```

6.3.11.2 Example: [Figures](#)

Figure

```
1 interface Figure {
2     <T> T accept(Visitor<T> visitor);
3 }
```

Visitor

```
1 interface Visitor<T> {
2
3     T visit(Square square);
4
5     T visit(Circle circle);
6
7     T visit(Rectangle rectangle);
8 }
```

Circle

```
1 public class Circle implements Figure {
2
3     private final double radius;
4
5     public Circle(double radius) {
6         this.radius = radius;
7     }
8
9     public double radius() {
10        return this.radius;
11    }
12
13    @Override
14    public <T> T accept(Visitor<T> visitor) {
15        return visitor.visit(this);
16    }
17 }
```

Rectangle

```
1 public class Rectangle implements Figure {
2
3     private final double weight;
4     private final double height;
5
6     public Rectangle(double weight, double height) {
7         this.weight = weight;
8         this.height = height;
9     }
10
11    public double weight() {
12        return this.weight;
13    }
14 }
```

```
15     public double height() {
16         return this.height;
17     }
18
19     @Override
20     public <T> T accept(Visitor<T> visitor) {
21         return visitor.visit(this);
22     }
23 }
```

Square

```
1  public class Square implements Figure {
2
3     private final double side;
4
5     public Square(double side) {
6         this.side = side;
7     }
8
9     public double side() {
10        return this.side;
11    }
12
13    @Override
14    public <T> T accept(Visitor<T> visitor) {
15        return visitor.visit(this);
16    }
17 }
```

AreaVisitor

```
1  public class AreaVisitor implements Visitor<Double> {
2
3     @Override
4     public Double visit(Square square) {
5         return square.side() * square.side();
6     }
7
8     @Override
9     public Double visit(Circle circle) {
10        return Math.PI * circle.radius() * circle.radius();
11    }
12
13     @Override
14     public Double visit(Rectangle rectangle) {
15         return rectangle.height() * rectangle.weight();
16     }
17 }
```

PerimeterVisitor

```
1 public class PerimeterVisitor implements Visitor<Double> {
2
3     @Override
4     public Double visit(Square element) {
5         return 4 * element.side();
6     }
7
8     @Override
9     public Double visit(Circle element) {
10        return 2 * Math.PI * element.radius();
11    }
12
13     @Override
14     public Double visit(Rectangle element) {
15         return 2 * element.height() + 2 * element.height();
16     }
17 }
```

Client

```
1 public class Client {
2
3     public static void main(String[] args) {
4         List<Figure> figures = List.of(
5             new Circle(3),
6             new Square(4),
7             new Rectangle(2, 8));
8
9         calculateArea(figures);
10        calculatePerimeter(figures);
11    }
12
13     private static void calculateArea(List<Figure> figures) {
14         Visitor<Double> visitor = new AreaVisitor();
15
16         double totalArea = figures.stream()
17             .mapToDouble(f -> f.accept(visitor))
18             .sum();
19
20         System.out.println(String.format("Total area: %f ", totalArea));
21     }
22
23     private static void calculatePerimeter(List<Figure> figures) {
24         Visitor<Double> visitor = new PerimeterVisitor();
25
26         double totalPerimeter = figures.stream()
27             .mapToDouble(f -> f.accept(visitor))
28             .sum();
29
30         System.out.println(String.format("Total perimeter: %f", totalPerimeter));
31     }
32 }
```

Client output

```
1 Total area: 60,274334
2 Total perimeter: 66,849556
```

Advantages:

- New operations can be **added easily**.
- The logic of the operations is **centralised** in the visitor and not dispersed.
- The visitor pattern allows the operation to be defined **without changing the class** of any of the objects in the collection.

Disadvantages:

- The implementation of the visitor concrete classes could **break the encapsulation** of the visited objects. Due to the fact, that a new visitor class needs access to the public members of these objects.