DePaul University

Research Project Presentation
Design Patterns:
Adapter

Object-oriented Software Development
SE 350– Spring 2021

Vahid Alizadeh

Week 10.2
June 3, 2021

Announcements

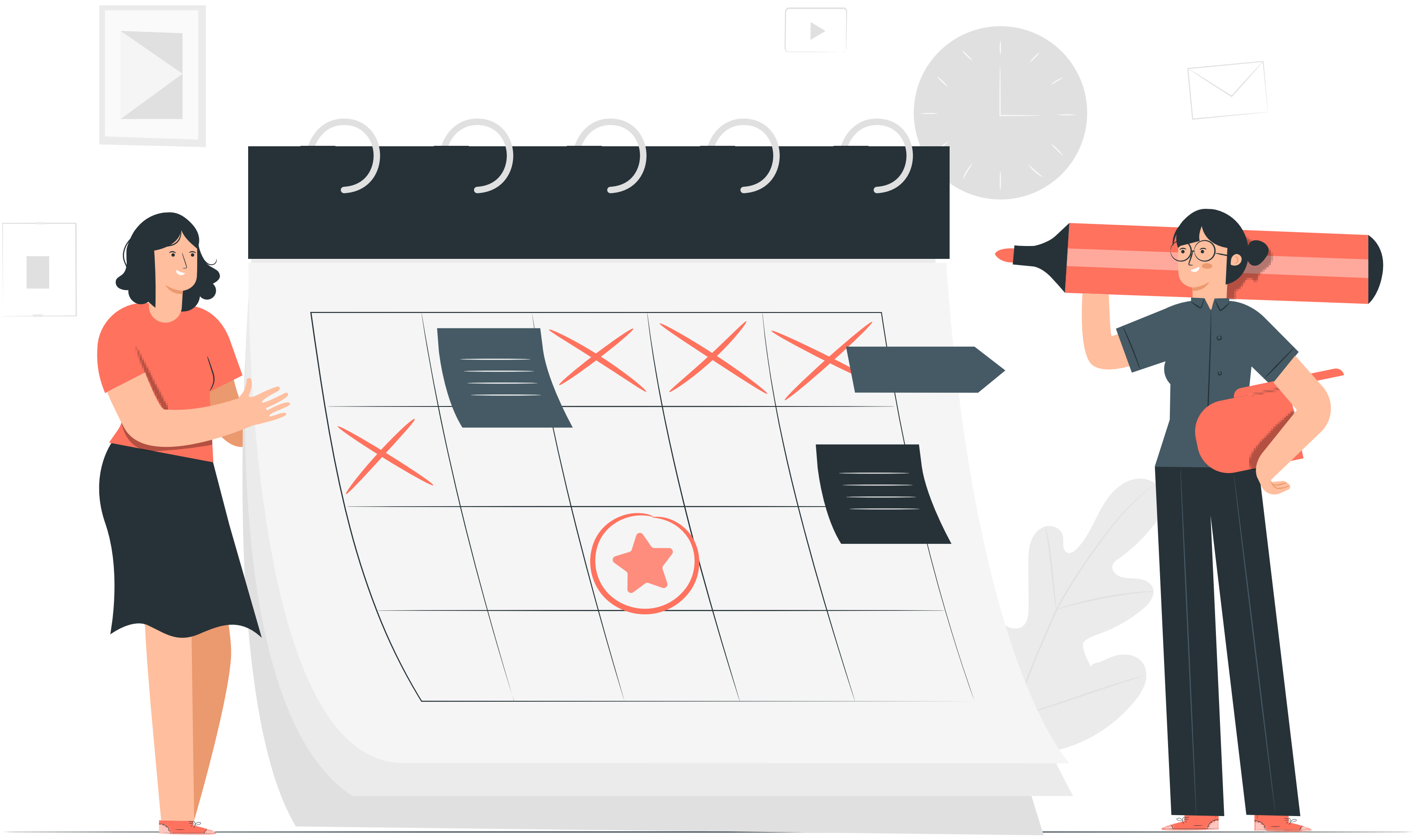
Research Project: Presentation

Future Schedule

Assignment 3 is graded.

Assignment 4 Solutions > GitHub

- ~~Assignment 1~~
- ~~Assignment 2~~
- ~~Mid Term Exam~~
- ~~Assignment 3:~~
  - ~~Release: Week 7~~
  - ~~Due: Week 8~~
- **Assignment 4:**
  - ~~Release: Week 8~~
  - ~~Due: Week 9~~
- **Bonus Research Project:**
  - Presentation Due: Week 10.2 >> **7 Mins**
  - Report Due: Week 11 >> **June 8**
- **Final Exam:**
  - Week 11 - June 9-11 (Wed-Fri)



SE 350: OO Software Development

Final Exam

Instructor: Vahid Alizadeh

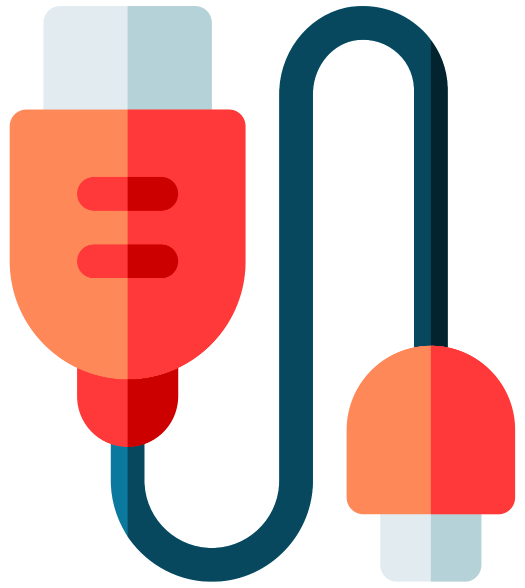
Email: v.alizadeh@depaul.edu

Quarter: Spring 2021

Date: June 9-11, 2021

DePaul University College of Computing and Digital Media

Last update: May 27, 2021

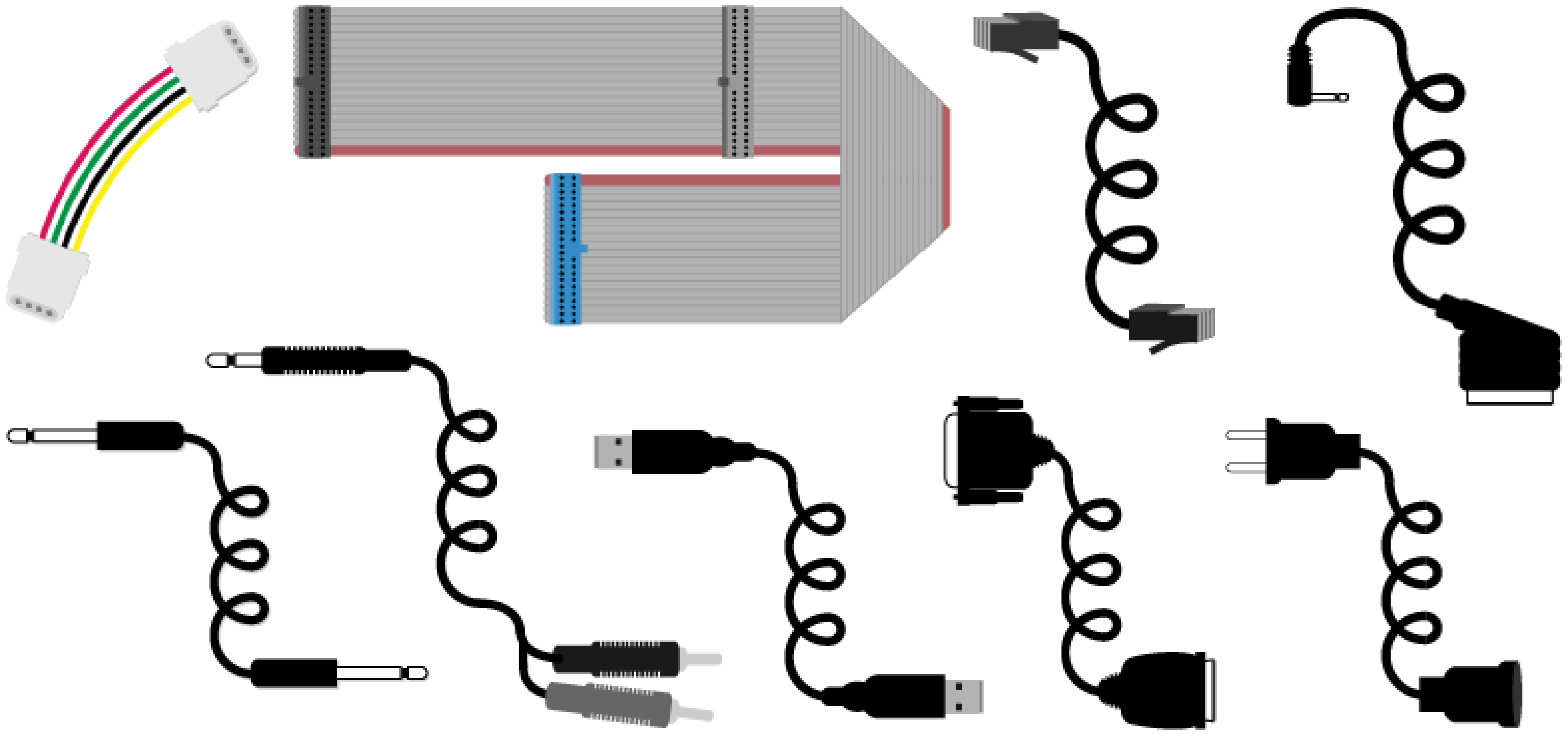


Adapter Pattern

STRUCTURAL

Adapter Pattern Introduction

Adapter is a structural pattern that allows two incompatible interfaces work together. The object that joins these unrelated interface is called an Adapter.



Adapter Design Pattern

INTENT

- Adopt an interface to a new client interface.
- Wrap an old existing class with a new interface.

PROBLEM

- Reusing an already existed component but its representation is not compatible with your architecture.

STRUCTURE

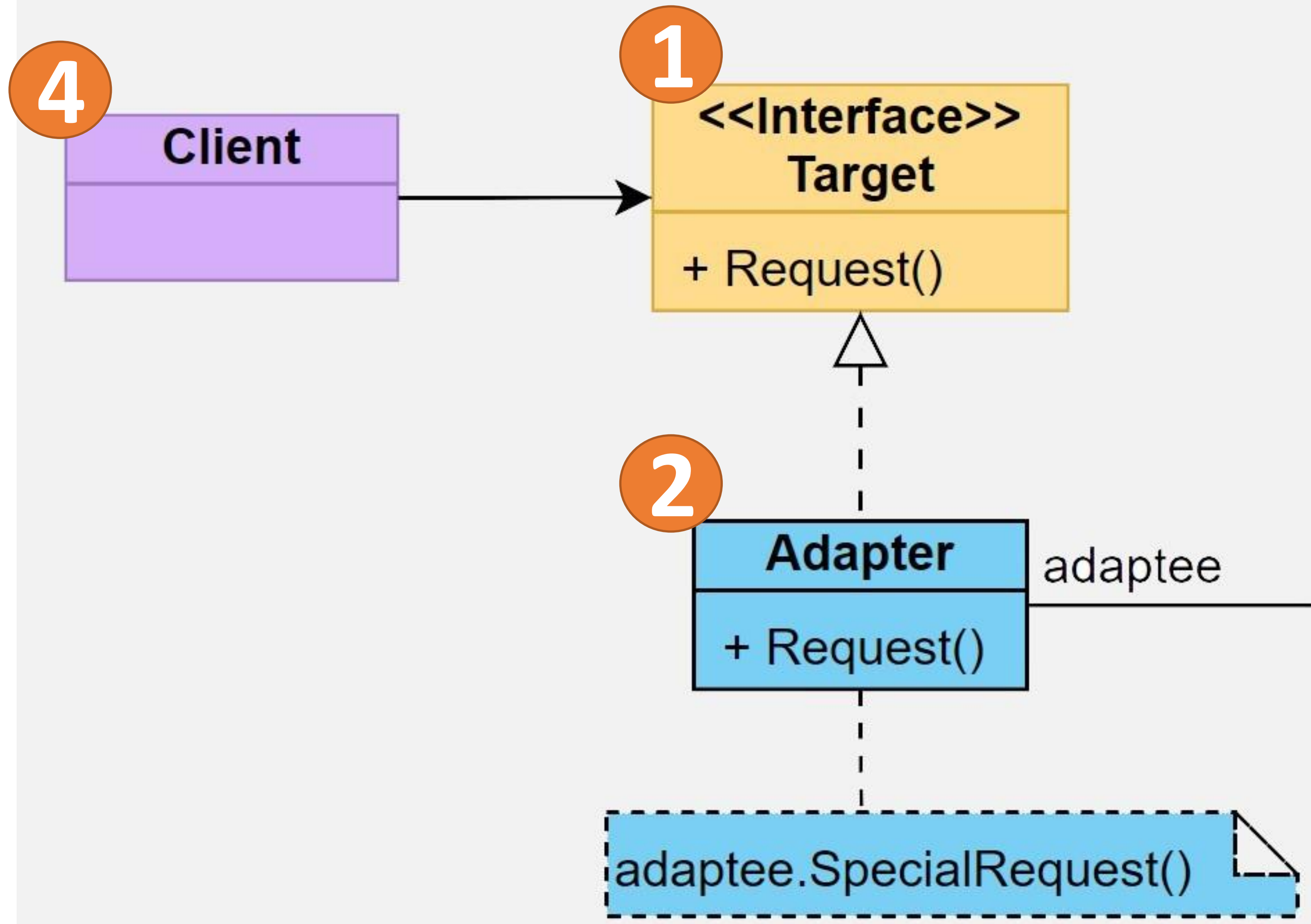
Class Adapter
(via Java Inheritance)

Object Adapter
(via Java Composition)

Adapter Design Pattern: Structures

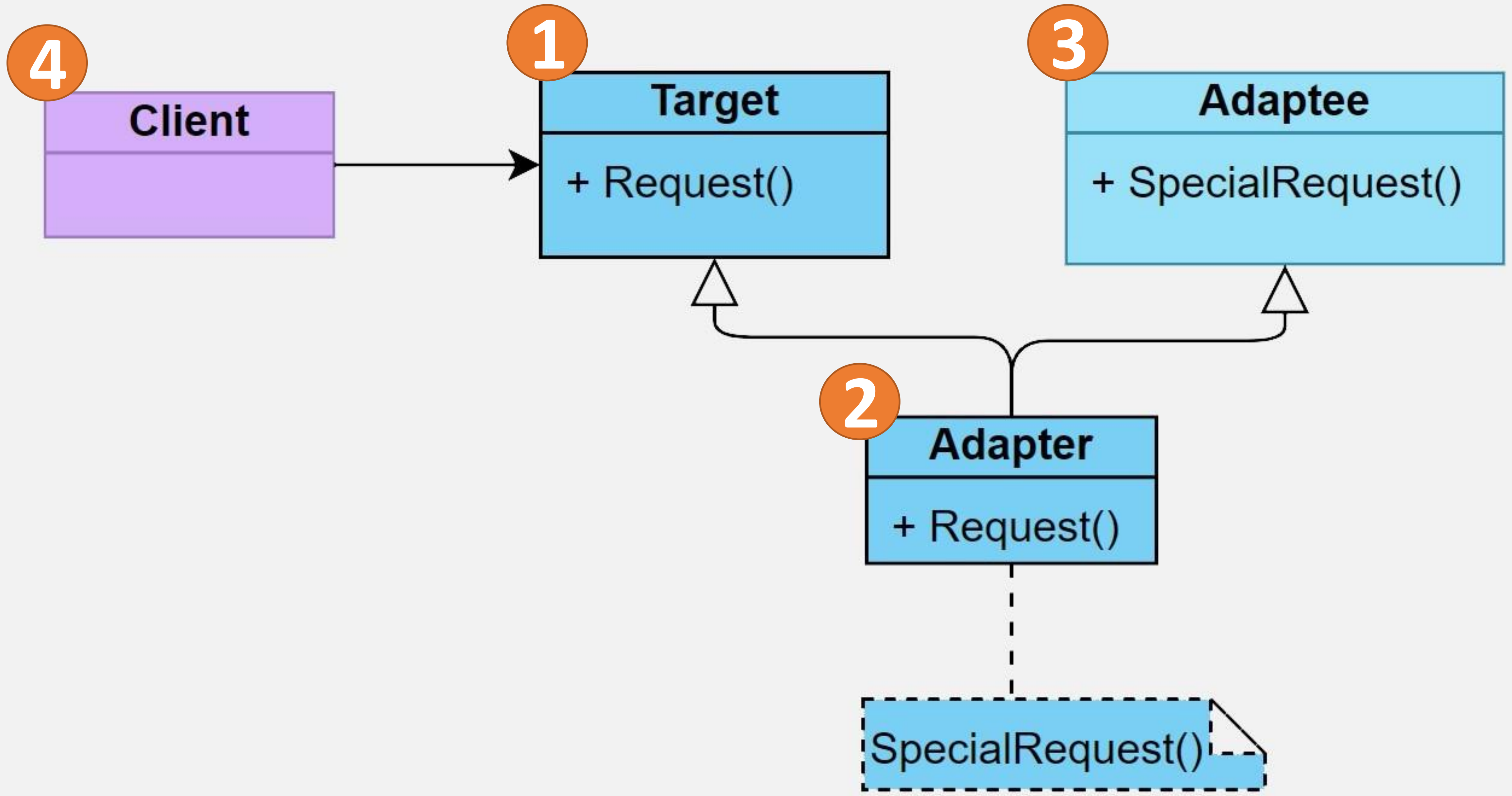
STRUCTURE: Object Adapter

- 1- Target
- 2- Adapter (Uses **composition**)
- 3- Adaptee
- 4- Client



STRUCTURE: Class Adapter

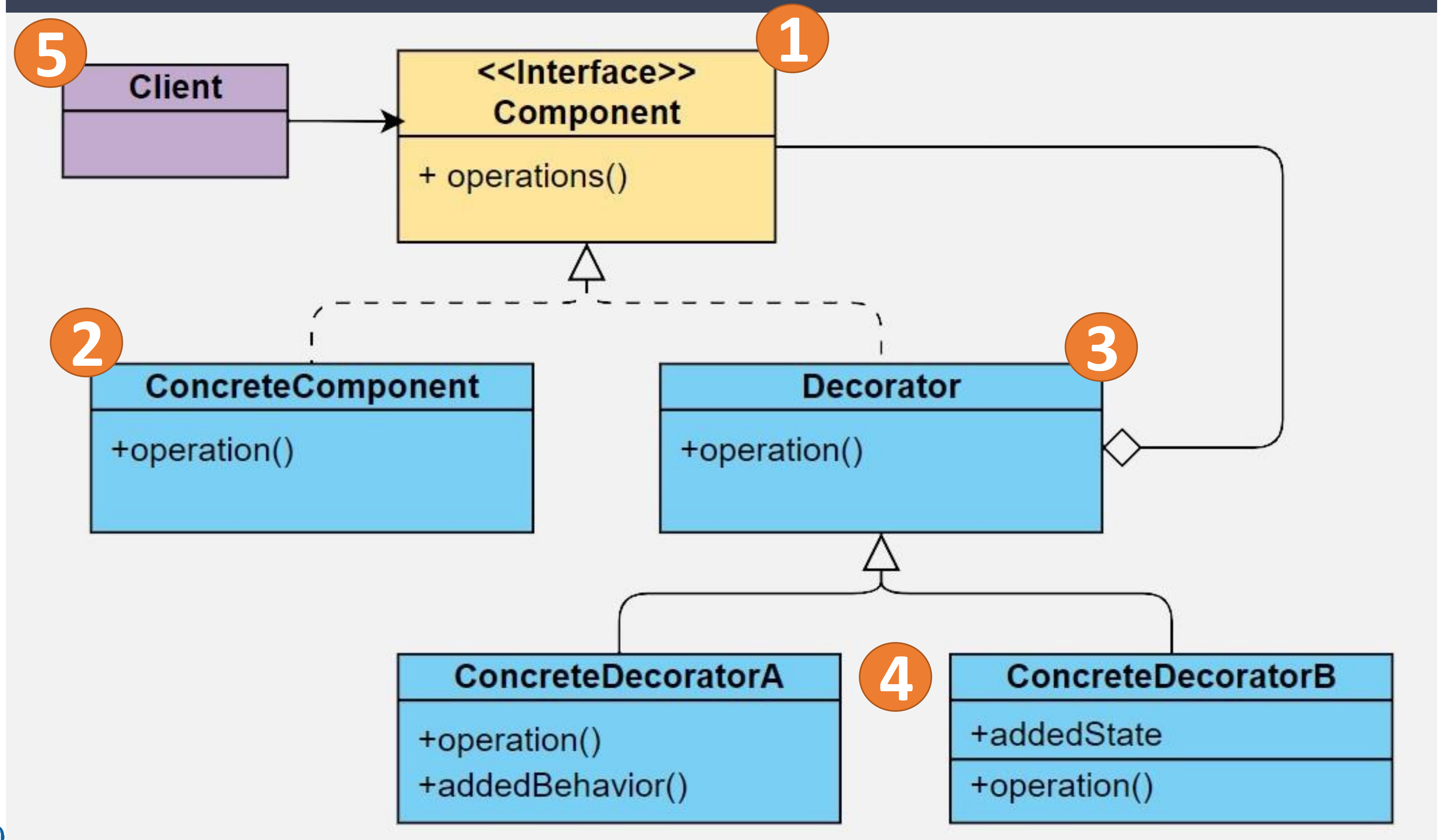
- 1- Target
- 2- Adapter (Uses **inheritance**)
- 3- Adaptee
- 4- Client



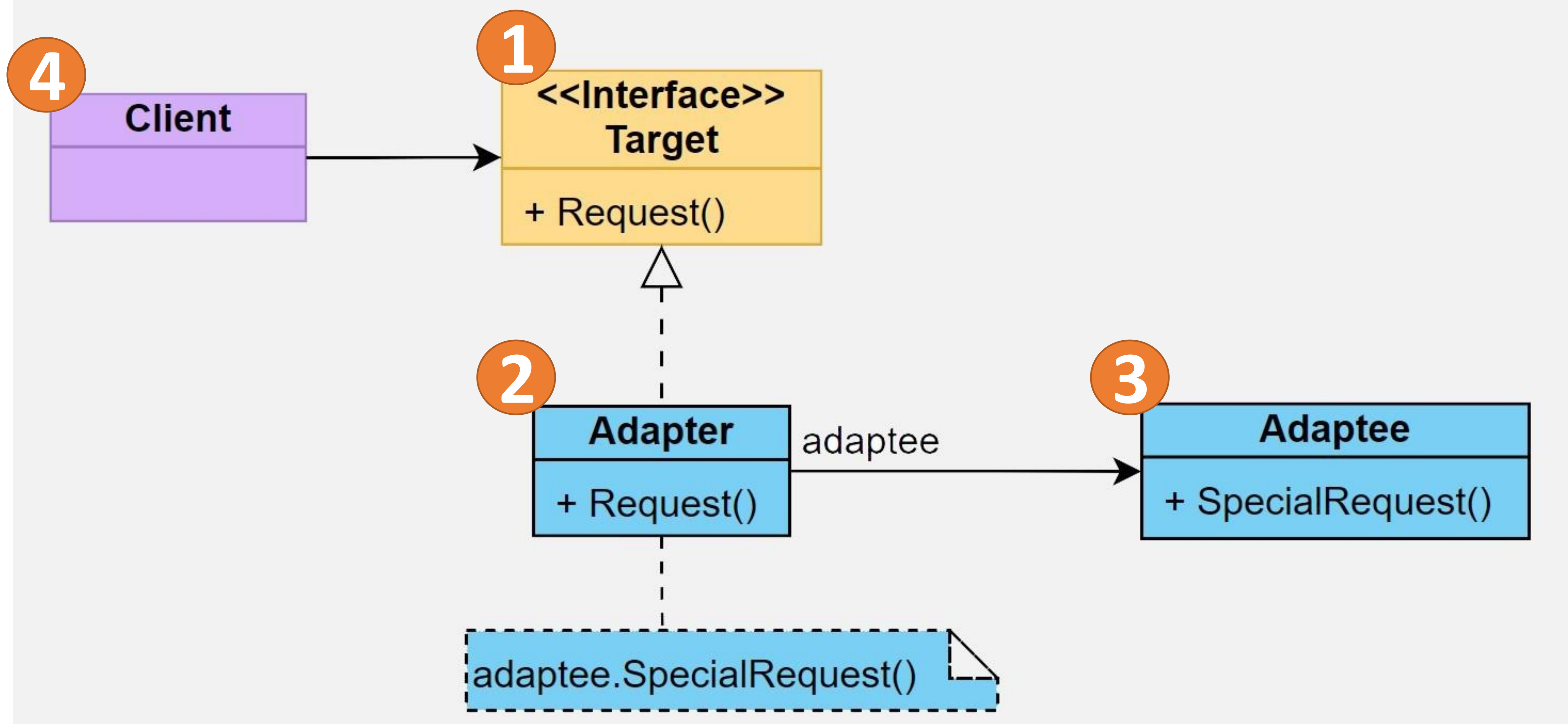
Decorator vs. Adapter

	Adapter	Decorator
Composes "origin" class	True	True
Modifies original interface	True	False
Modifies behavior of interface	False	True
Proxies method calls	True	True

STRUCTURE: Decorator



STRUCTURE: Object Adapter



Adapter Pattern: Real-world Example

1

2

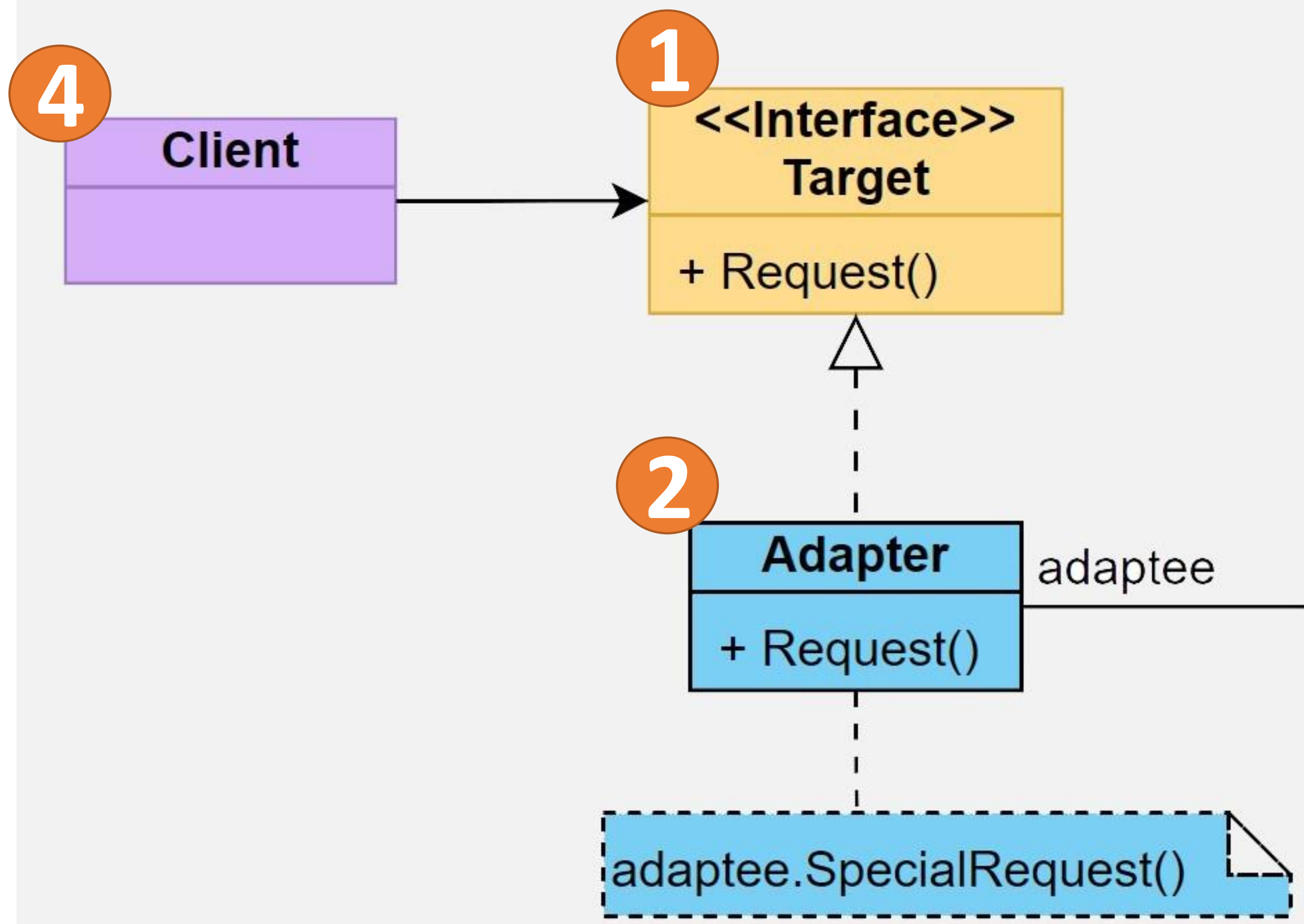
3

Adapter Use Case Example: Voltage Converter App

Object Adapter Implementation

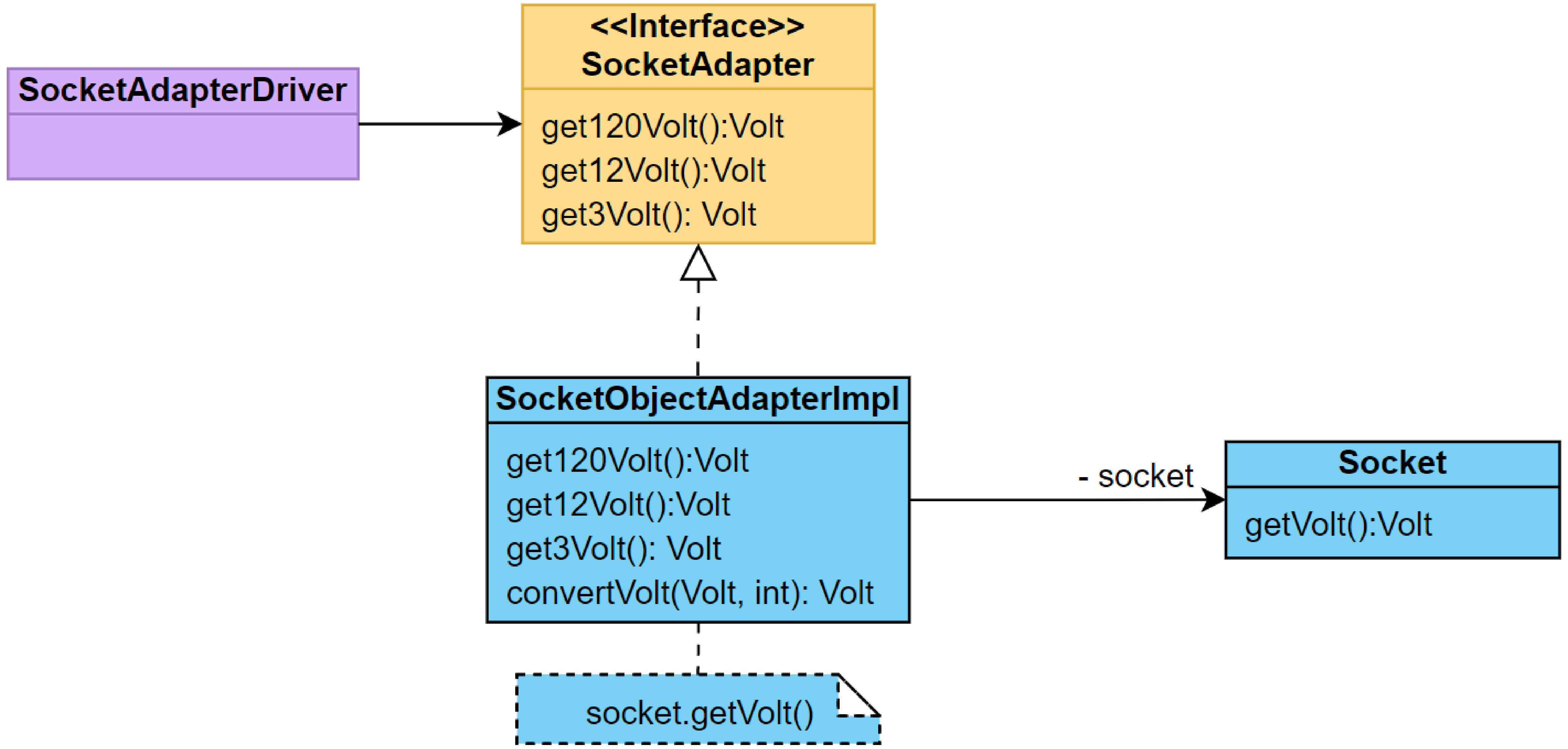
STRUCTURE: Object Adapter

- 1- Target
- 2- Adapter (Uses **composition**)
- 3- Adaptee
- 4- Client



Our Solution

Find the source codes in the course GitHub repository.

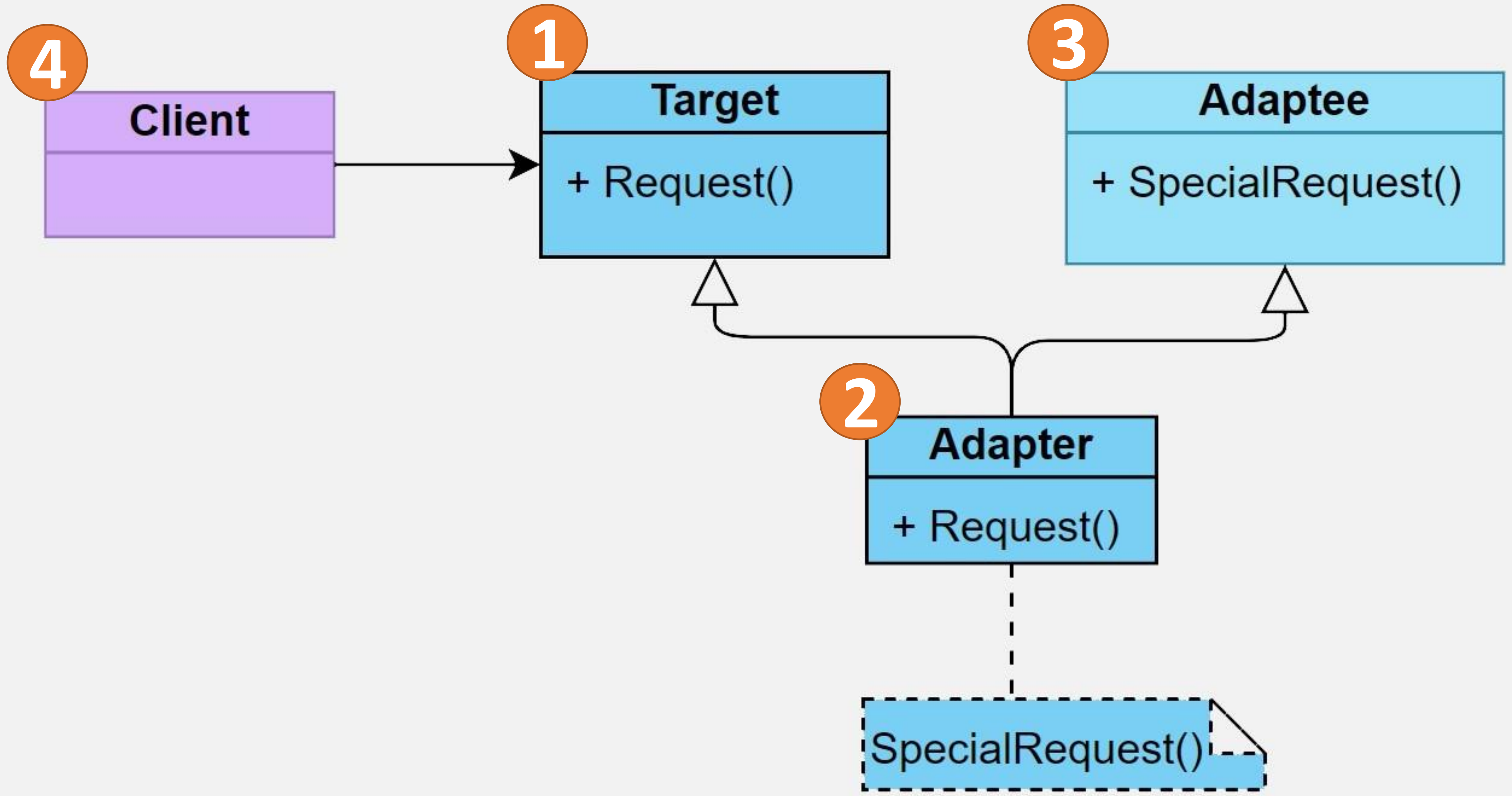


Adapter Use Case Example: Voltage Converter App

Class Adapter Implementation

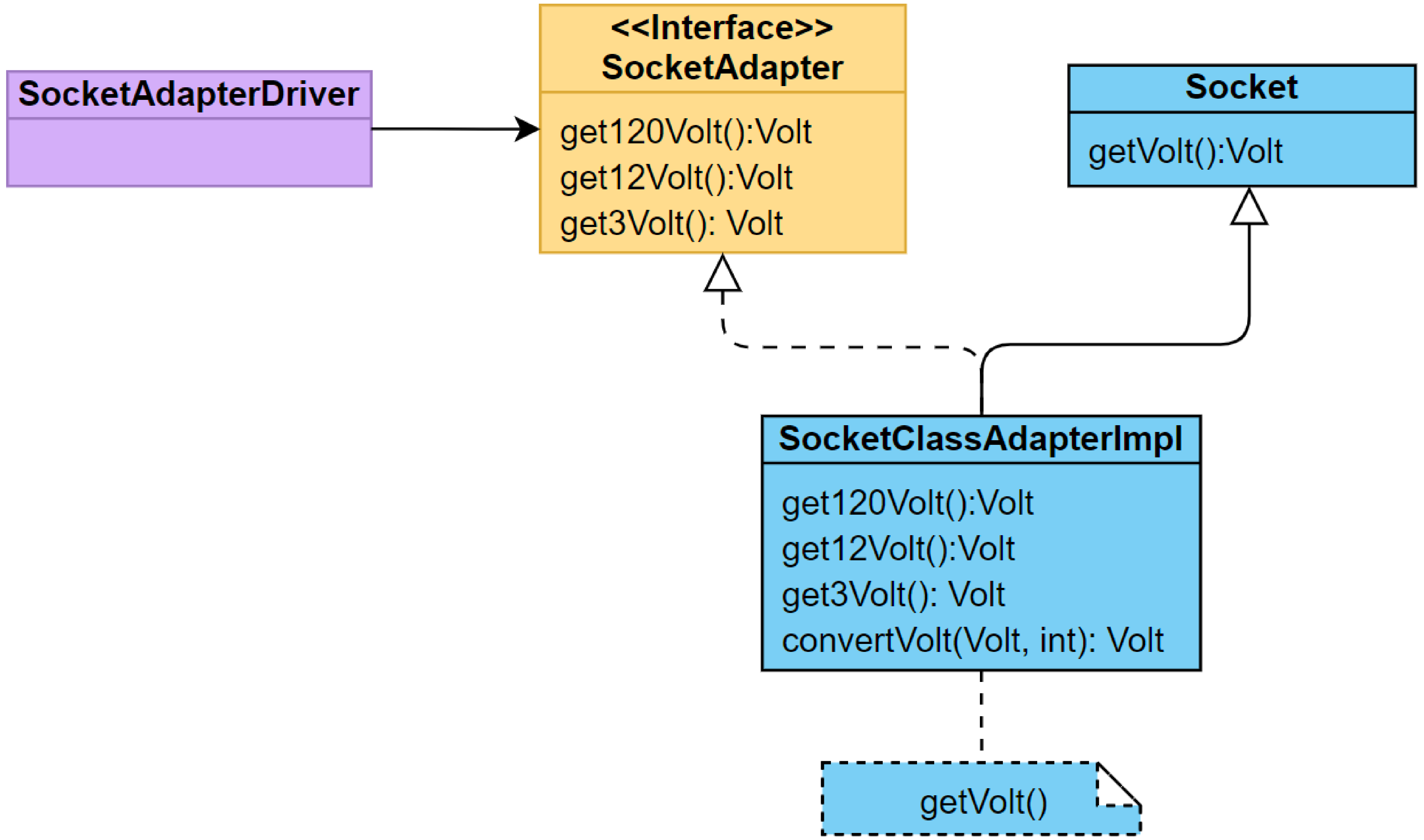
STRUCTURE: Class Adapter

- 1- Target
- 2- Adapter (Uses **inheritance**)
- 3- Adaptee
- 4- Client



Our Solution

Find the source codes in the course GitHub repository.

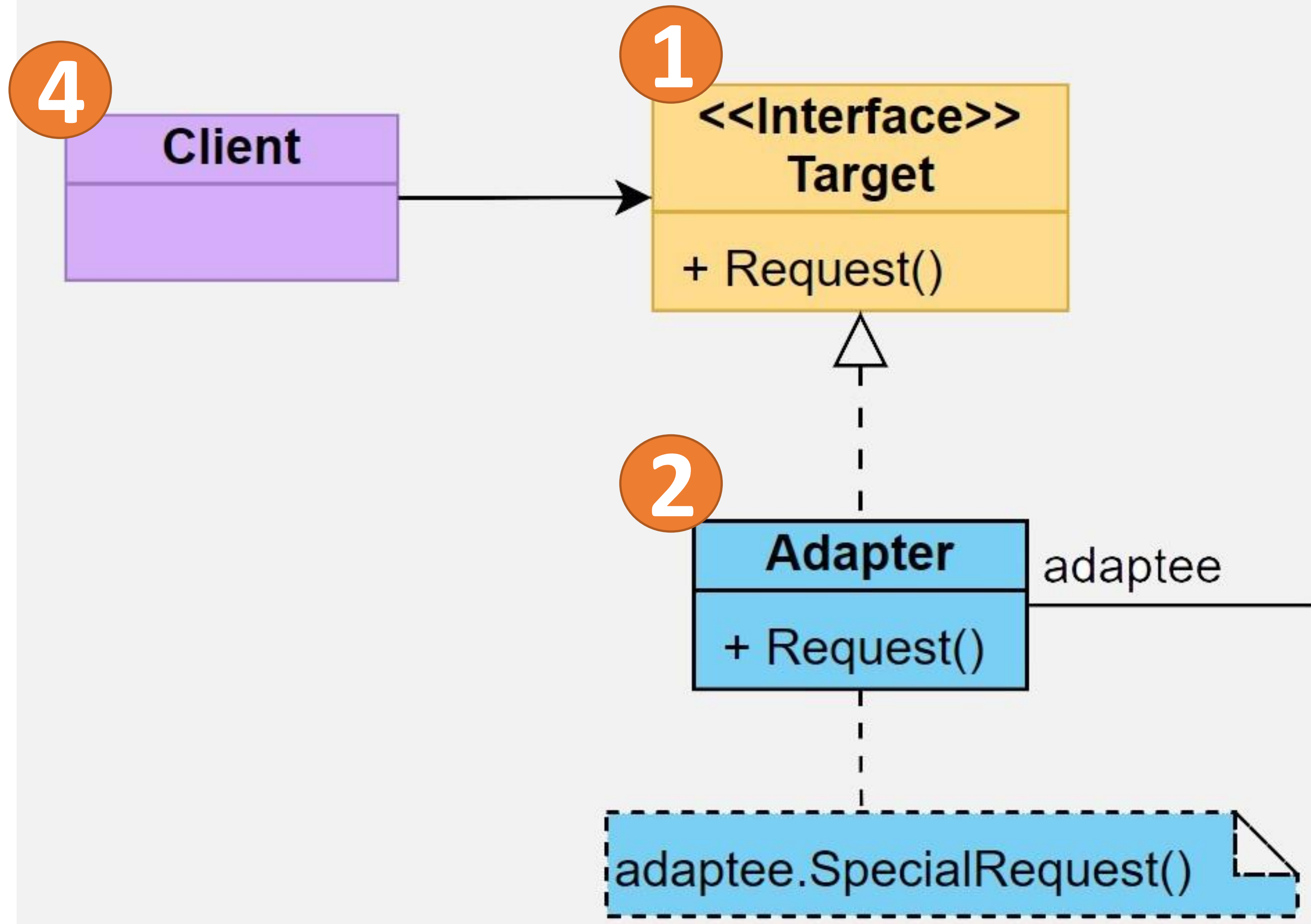


Adapter Use Case Example: Text Formatter

Object Adapter Implementation

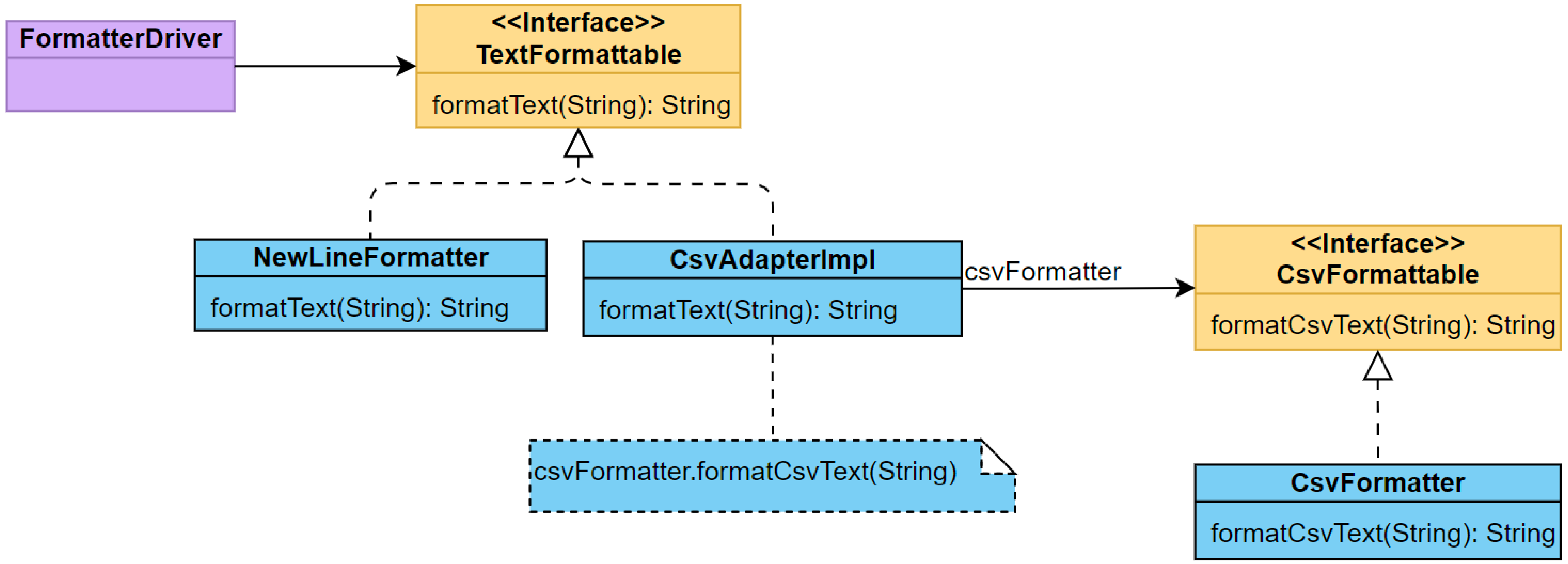
STRUCTURE: Object Adapter

- 1- Target
- 2- Adapter (Uses **composition**)
- 3- Adaptee
- 4- Client



Our Solution

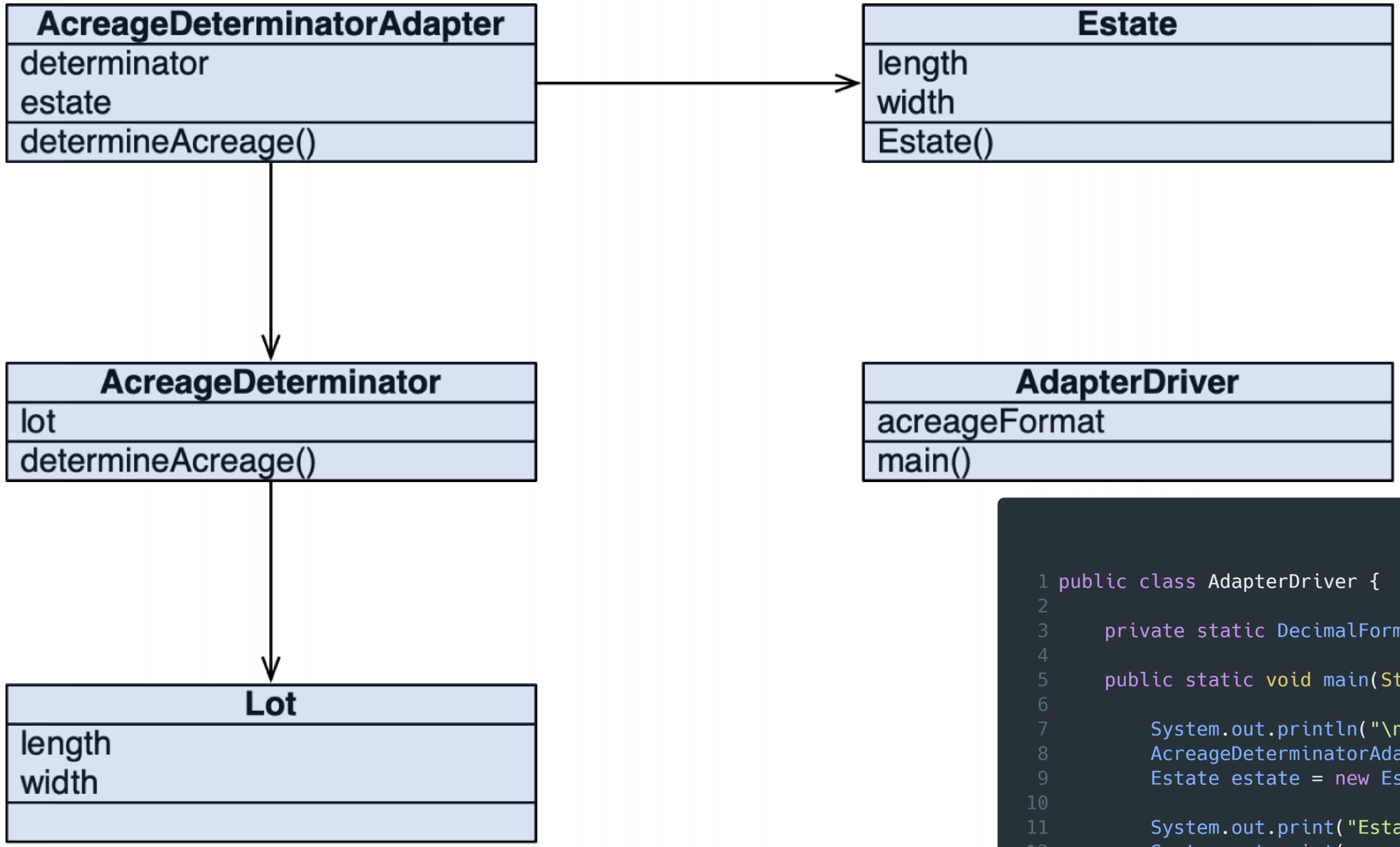
Find the source codes in the course GitHub repository.



Adapter Use Case Example: Real Estate App

```java
public class AcreageDeterminatorAdapter {

    AcreageDeterminator determinator;
    Estate estate;

    public double determineAcreage(Estate estate) {
        determinator = new AcreageDeterminator();
        this.estate = estate;
        Lot lot = new Lot();

        lot.length = this.estate.length;
        lot.width = this.estate.width;

        return (determinator.determineAcreage(lot) / 43560);
    }
}
```

```java
public class Estate {

    public double length;
    public double width;

    public Estate(int length, int width)
    {
        this.length = length;
        this.width = width;
    }
}
```




```java
public class AcreageDeterminator {

    Lot lot;

    public double determineAcreage(Lot lot) {
        this.lot = lot;
        return this.lot.length * this.lot.width;
    }
}
```

```java
public class Lot {

    public double length;
    public double width;
}
```

```java
public class AdapterDriver {

    private static DecimalFormat acreageFormat = new DecimalFormat(".##");

    public static void main(String[] args) {

        System.out.println("\n\nReal Estate Land Area Calculation");
        AcreageDeterminatorAdapter adAdapter = new AcreageDeterminatorAdapter();
        Estate estate = new Estate(2300, 6325);

        System.out.print("Estate Acreage: ");
        System.out.print(acreageFormat.format(adAdapter.determineAcreage(estate)));
    }
}
```

Find the source codes in the course GitHub repository.

Adapter Pattern Pros & Cons

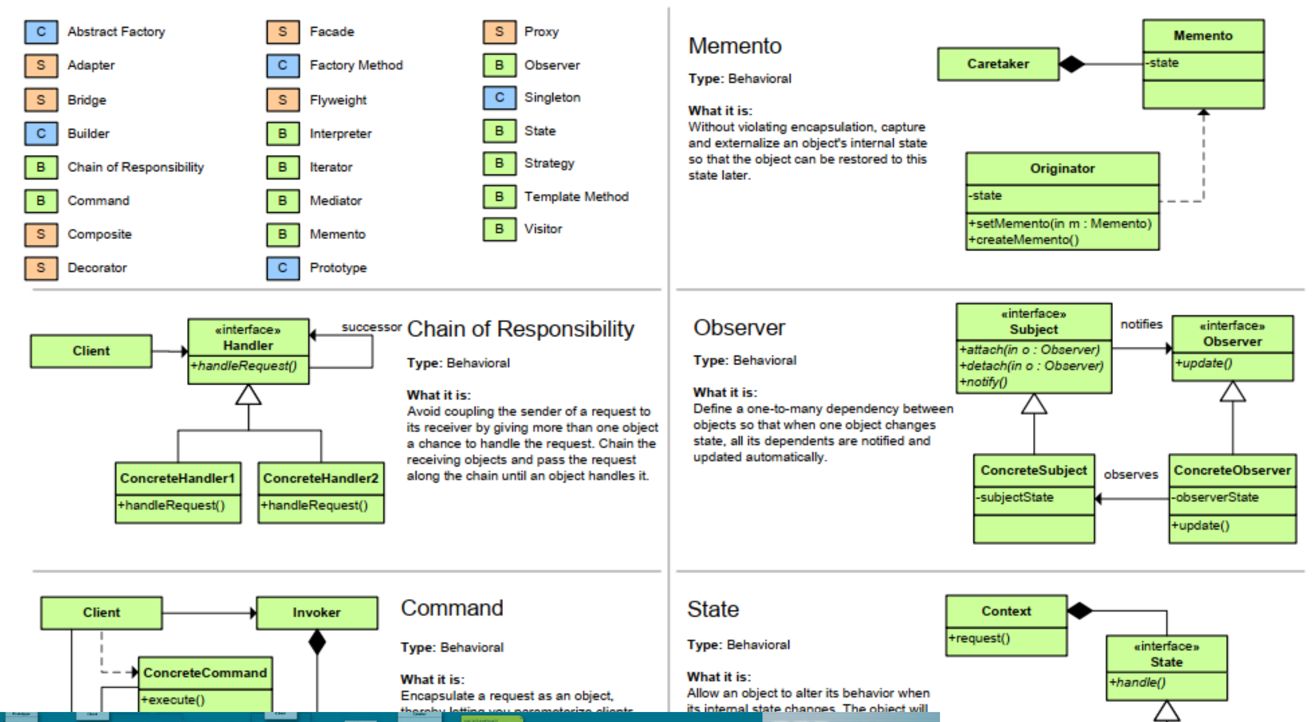
Pros	Cons
Reusability and Flexibility	We must forward all requests which increases the overhead.
Simpler client code. Swap between adapters via polymorphism.	Some architectures require using many adaptations in an adapter.
Single Responsibility Principle	Increases the overall complexity due to the introduction of new interfaces and classes.
Open/Closed Principle	

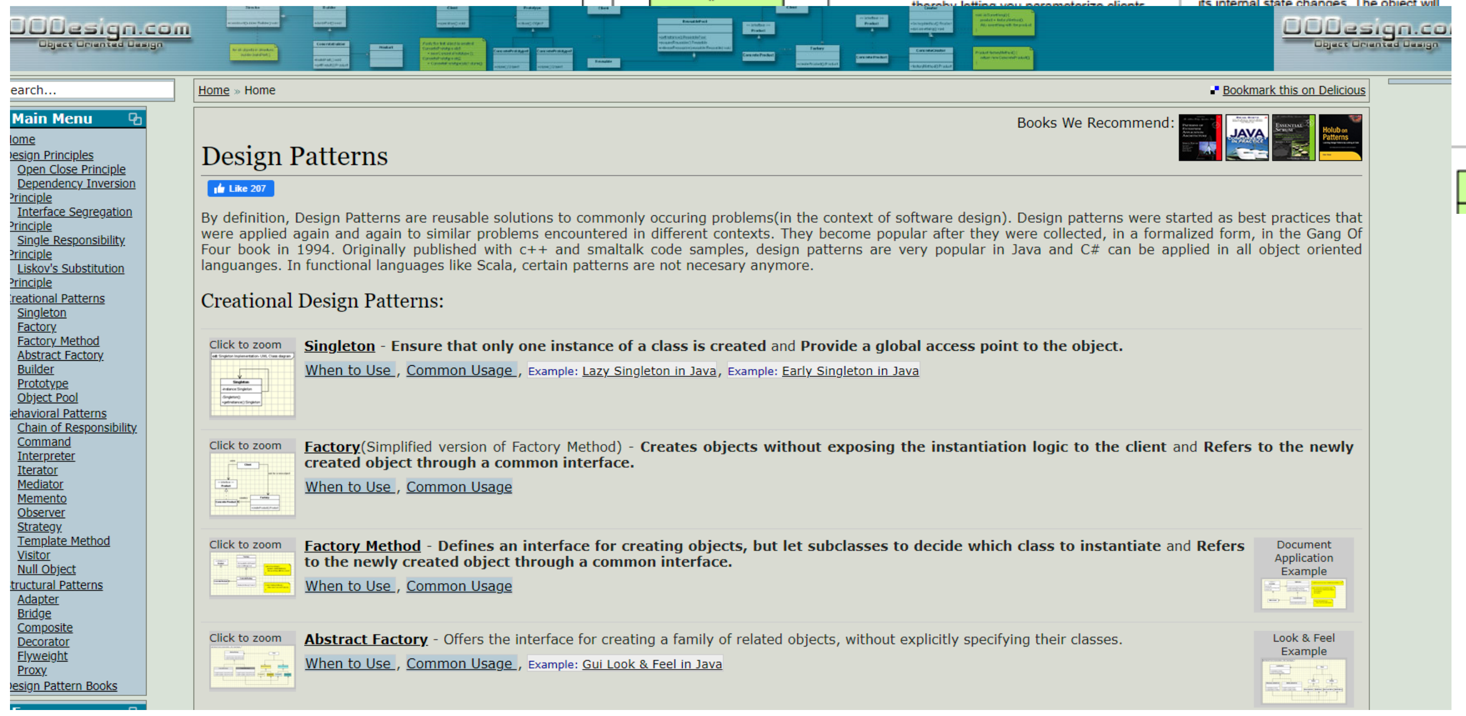
Resources

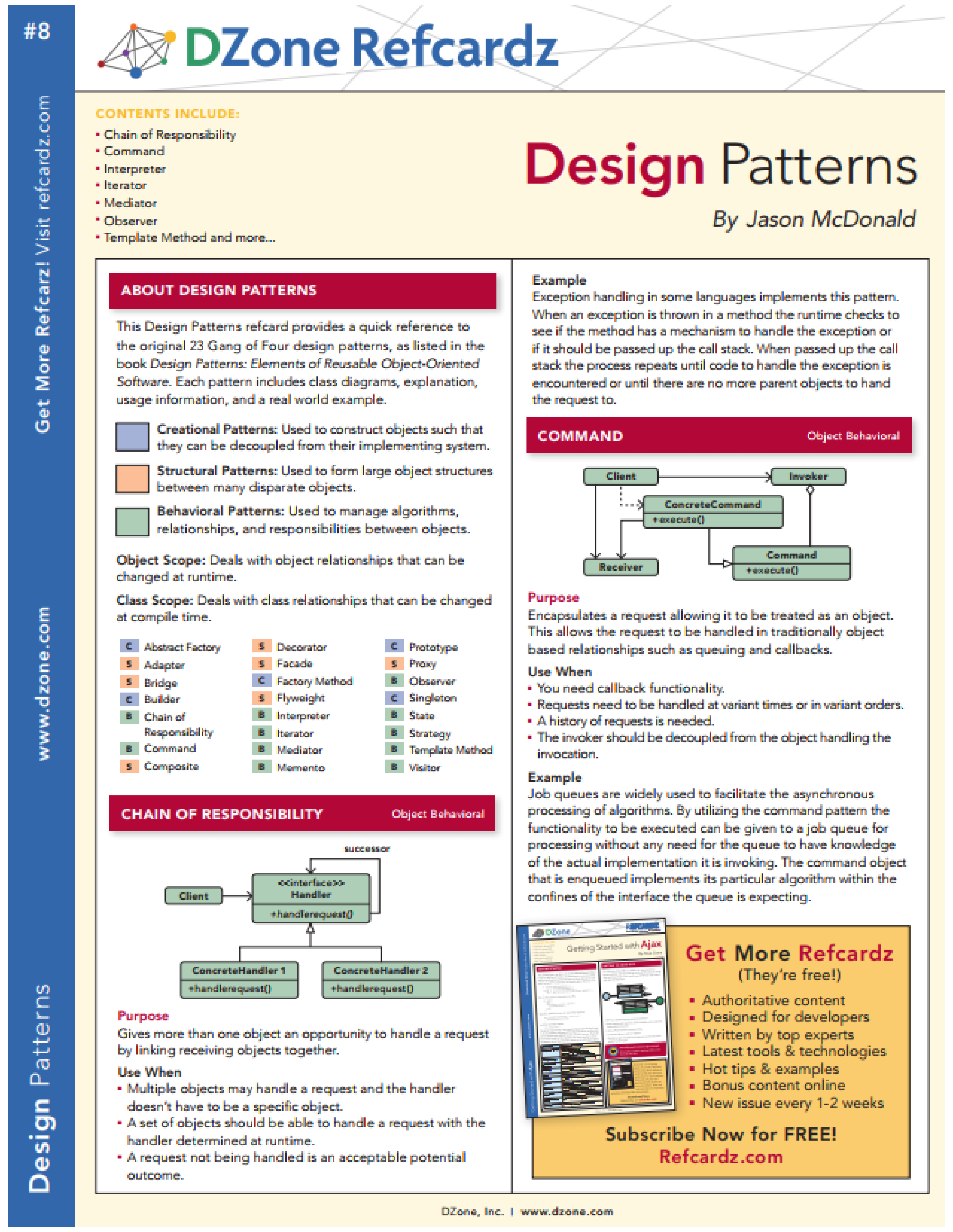
Design Patterns

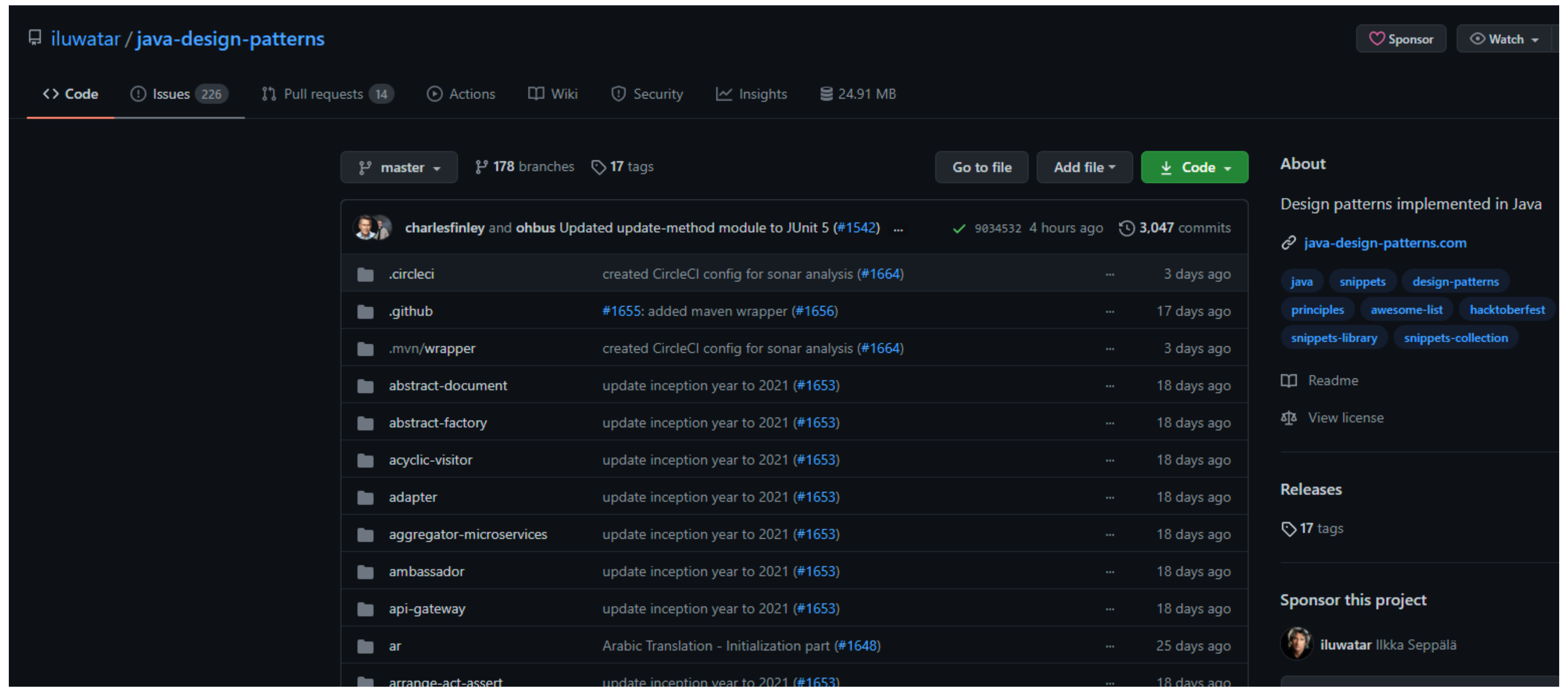
Design Patterns Resources

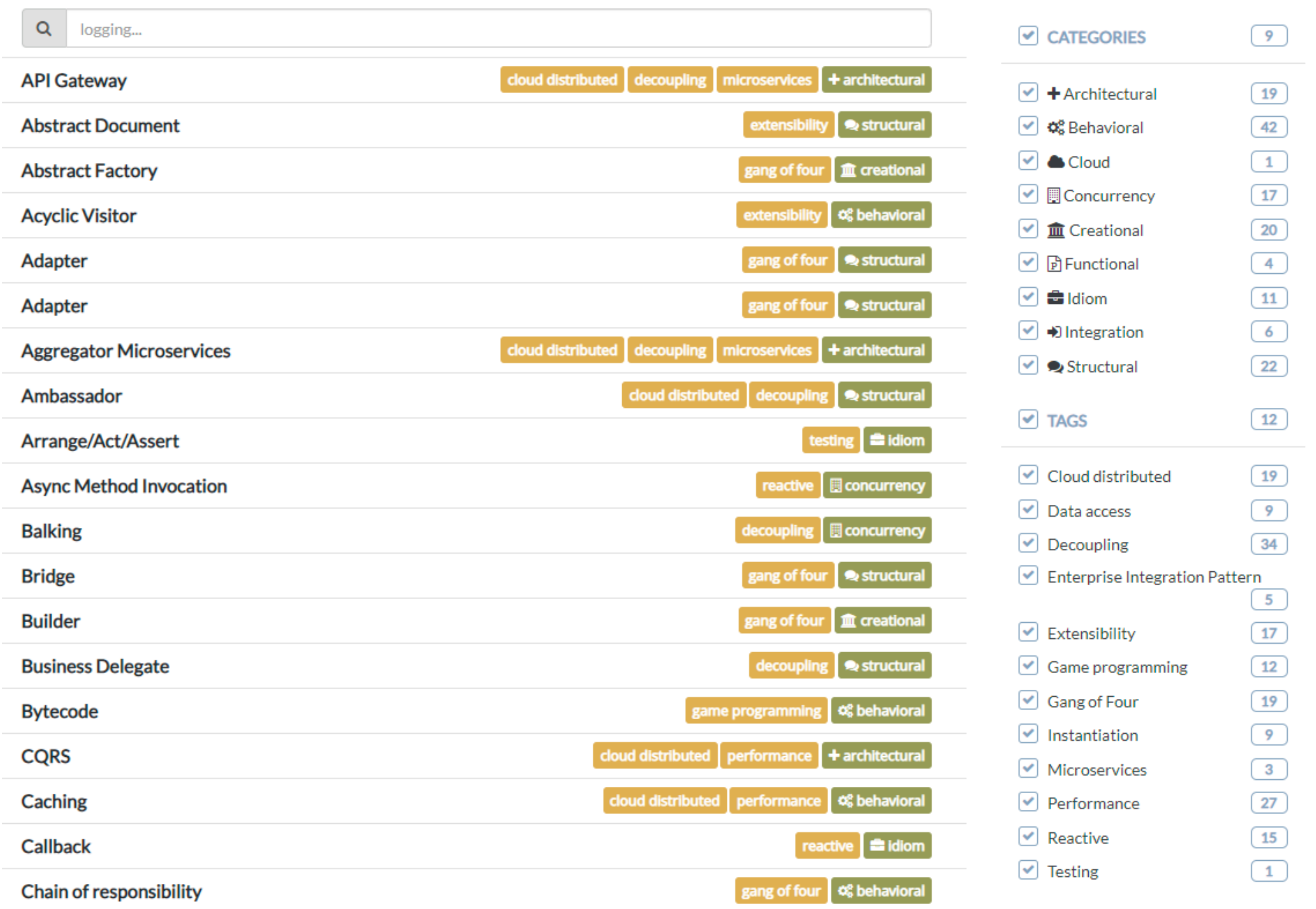
- **Cheat sheets**
  - Two cheat sheets are uploaded to D2L.
- **Code Repositories**
  - Repo 1 (Very complete)
  - Repo 2 (Only GoF)
  - Repo 3 (Only GoF Structures)
- **Online Catalogs**
  - https://www.oodesign.com/
  - https://java-design-patterns.com/
  - http://www.design-patterns-stories.com/











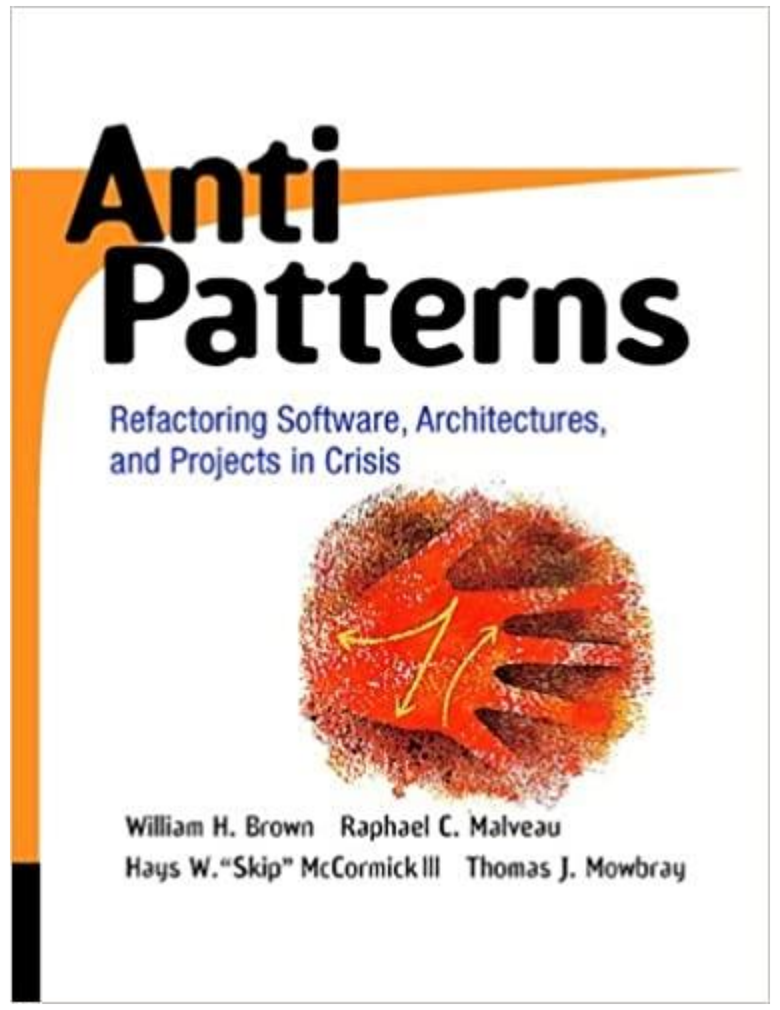
DePaul University

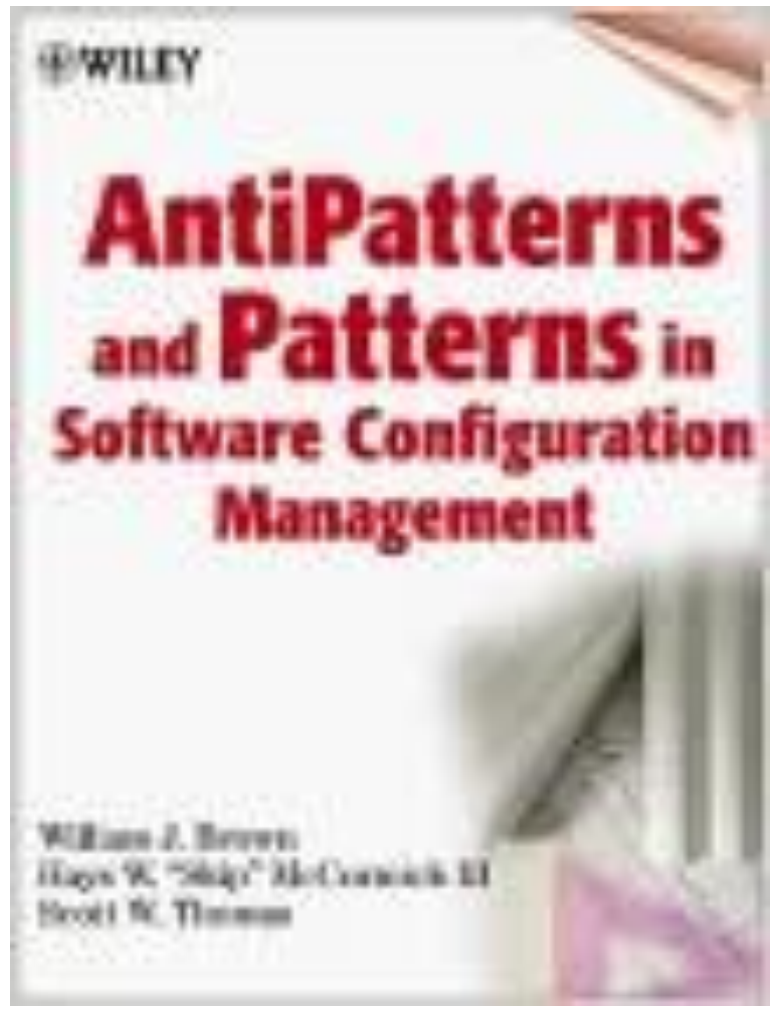
Resources

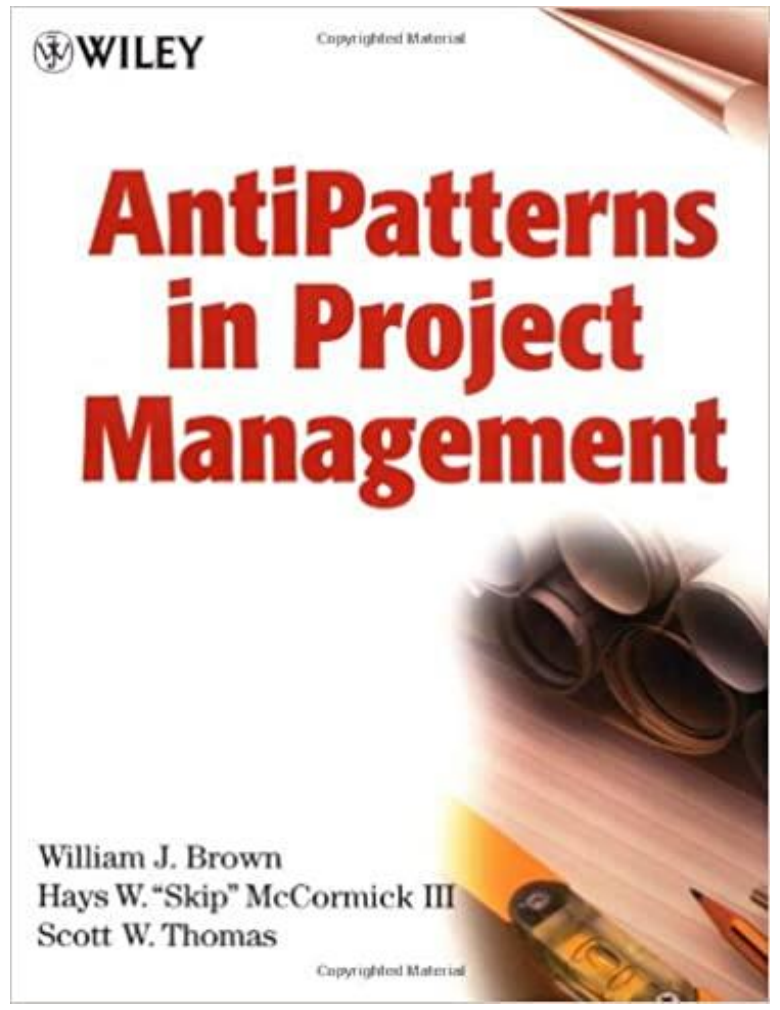
Code Quality

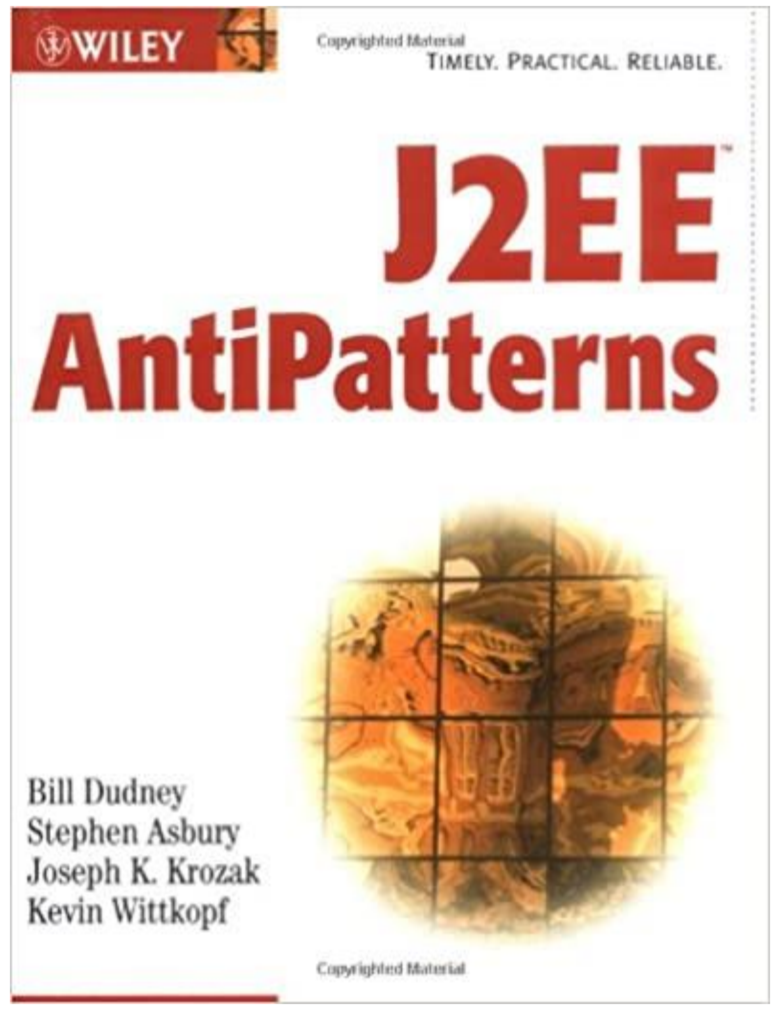
Books

- **Books:**
  - AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis
  - Anti-Patterns and Patterns in Software Configuration Management
  - AntiPatterns in Project Management
  - J2EE AntiPatterns





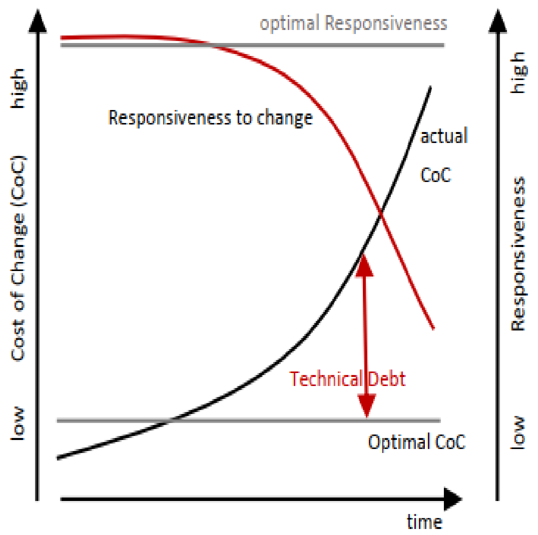




Clean Cheat Sheets

Clean Code

Why Clean Code
Code is clean if it can be understood easily – by everyone on the team. With understandability comes readability, changeability, extensibility and maintainability. All the things needed to keep a project going over a long time without accumulating up a large amount of technical debt.



Writing clean code from the start in a project is an investment in keeping the cost of change as constant as possible throughout the lifecycle of a software product. Therefore, the initial cost of change is a bit higher when writing clean code (grey line) than quick and dirty programming (black line), but is paid back quite soon. Especially if you keep in mind that most of the cost has to be paid during maintenance of the software. Unclean code results in technical debt that increases over time if not refactored into clean code. There are other reasons leading to Technical Debt such as bad processes and lack of documentation, but unclean code is a major driver. As a result, your ability to respond to changes is reduced (red line).

In Clean Code, Bugs Cannot Hide
Most software defects are introduced when changing existing code. The reason behind this is that the developer changing the code cannot fully grasp the effects of the changes made. Clean code minimises the risk of introducing defects by making the code as easy to understand as possible.

Principles
Loose Coupling +
Two classes, components or modules are coupled when at least one of them uses the other. The less these items know about each other, the looser they are coupled.
A component that is only loosely coupled to its environment can be more

Smells
Rigidity –
The software is difficult to change. A small change causes a cascade of subsequent changes.

Fragility –
The software breaks in many places due to a single change.

Immobility –
You cannot reuse parts of the code in other projects because of involved risks and high effort.

Viscosity of Design –
Taking a shortcut and introducing technical debt requires less effort than doing it right.

Viscosity of Environment –
Building, testing and other tasks take a long time. Therefore, these activities are not executed properly by everyone and technical debt is introduced.

Needless Complexity –
The design contains elements that are currently not useful. The added complexity makes the code harder to comprehend. Therefore, extending and changing the code results in higher effort than necessary.

Needless Repetition –
Code contains exact code duplications or design duplicates (doing the same thing in a different way). Making a change to a duplicated piece of code is more expensive and more error-prone because the change has to be made in several places with the risk that one place is not changed accordingly.

Opacity –
The code is hard to understand. Therefore, any change takes additional time to first reengineer the code and is more likely to result in defects due to not understanding the side effects.

Class Design
Single Responsibility Principle (SRP) +
A class should have one, and only one, reason to change.

Open Closed Principle (OCP) +
You should be able to extend a classes behaviour without modifying it.

Liskov Substitution Principle (LSP) +
Derived classes must be substitutable for their base classes.

Dependency Inversion Principle (DIP) +
Depend on abstractions, not on concretions.

Interface Segregation Principle (ISP) +
Make fine grained interfaces that are client-specific.

Stable Abstractions Principle (SAP) +
Abstractness increases with stability

General
Follow Standard Conventions +
Coding-, architecture-, design guidelines (check them with tools)

Keep it Simple, Stupid (KISS) +
Simpler is always better. Reduce complexity as much as possible.

Boy Scout Rule +
Leave the campground cleaner than you found it.

Root Cause Analysis +
Always look for the root cause of a problem. Otherwise, it will get you again.

Multiple Languages in One Source File –
C#, Java, JavaScript, XML, HTML, XAML, English, German …

Environment
Project Build Requires Only One Step +
Check out and then build with a single command.

Executing Tests Requires Only One Step +
Run all unit tests with a single command.

Source Control System +
Always use a source control system.

Continuous Integration +
Assure integrity with Continuous Integration

Overridden Safeties –
Do not override warnings, errors, exception handling – they will catch you.

Dependency Injection
Decouple Construction from Runtime +
Decoupling the construction phase completely from the runtime helps to simplify the runtime behaviour.

Design
Keep Configurable Data at High Levels +
If you have a constant such as default or configuration value that is known and expected at a high level of abstraction, do not bury it in a low-level function. Expose it as an argument to the low-level function called from the high-level function.

Don't Be Arbitrary +
Have a reason for the way you structure your code, and make sure that reason is communicated by the structure of the code. If a structure appears

Code at Wrong Level of Abstraction –
Functionality is at wrong level of abstraction, e.g. a PercentageFull property on a generic IStack<T>.

Fields Not Defining State –
Fields holding data that does not belong to the state of the instance but are used to hold temporary data. Use local variables or extract to a class abstracting the performed action.

Over Configurability –
Prevent configuration just for the sake of it – or because nobody can decide how it should be. Otherwise, this will result in overly complex, unstable systems.

Micro Layers –
Do not add functionality on top, but simplify overall.

Dependencies
Make Logical Dependencies Physical +
If one module depends upon another, that dependency should be physical, not just logical. Don't make assumptions.

Singletons / Service Locator –
Use dependency injection. Singletons hide dependencies.

Base Classes Depending On Their Derivatives –
Base classes should work with any derived class.

Too Much Information –
Minimise interface to minimise coupling

Feature Envy –
The methods of a class should be interested in the variables and functions of the class they belong to, and not the variables and functions of other classes. Using accessors and mutators of some other object to manipulate its data, is envying the scope of the other object.

Artificial Coupling –
Things that don't depend upon each other should not be artificially coupled.

Hidden Temporal Coupling –
If, for example, the order of some method calls is important, then make sure that they cannot be called in the wrong order.

Transitive Navigation –
Aka Law of Demeter, writing shy code.
A module should know only its direct dependencies.

Naming
Choose Descriptive / Unambiguous Names +
Names have to reflect what a variable, field, property stands for. Names

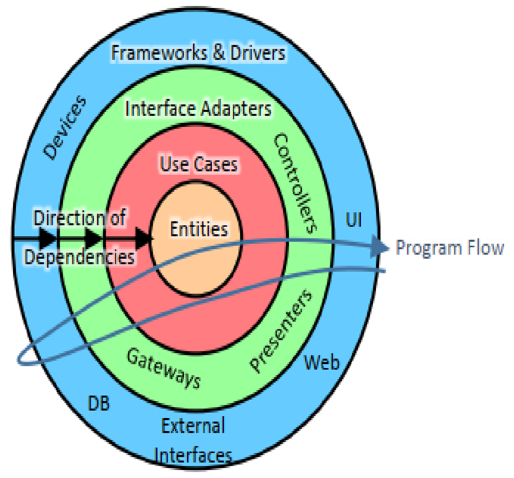
Clean Architecture

Why is a clean, simple, flexible, evolvable, and agile architecture important?
Software architecture is the high level structure of a software system, the discipline of creating such structures, and the documentation of these structures. [1]
It is the set of structures needed to reason about the software system, and comprises the software elements, the relations between them, and the properties of both elements and relations. [2]

In today's software development world, requirements change, environments change, team members change, technologies change, and so should the architecture of our systems.

The architecture defines the parts of a system that are hard and costly to change. Therefore we are in need of a clean, simple, flexible, evolvable, and agile architecture to be able to keep up with all the changes surrounding us.

Clean architecture [3]
An architecture that allows to replace details and is easy to verify.



Entities: Entities encapsulate enterprise-wide business rules. An entity can be an object with methods, or it can be a set of data structures and functions.

Use cases: Use cases orchestrate the flow of data to and from the entities, and direct those entities to use their enterprise-wide business rules to achieve the goals of the use cases.

Interface adapters: Adapters that convert data from the format most convenient for the use cases and entities, to the format most convenient for some external agency such as a database or the Web.

Frameworks and drivers: Glue code to connect UI, databases, devices etc. to the inner circles.

Program Flow: Starts on the outside and ends on the outside, but can go

Simple architecture
An architecture that is easy to understand. Simplicity is, however, subjective.

Consistent design decisions
One problem has one solution. Similar problems are solved similarly.

Number of concepts/technologies
Simple solutions make use of only a few different concepts and technologies.

Number of interactions
The less interactions the simpler the design.

A reasonable amount of components with only efferent coupling and most of the others with preferably only afferent coupling.

Size
Small systems/components are easier to grasp than big ones. Build large systems out of small parts.

Modularity
Build your system by connecting independent modules with a clearly defined interface (e.g. with adapters).

Flexible architecture
An architecture that supports change.

Separation of concerns
Divide your system into distinct features with as little overlap in functionality as possible so that they can be combined freely.

Software reflects user's mental model
When the structure and interactions inside the software match the user's mental model, changes in the real world can more easily be applied in software.

Abstraction
Separating ideas from specific implementations provides the flexibility to change the implementation. But beware of `over abstraction`.

Interface slimness
Fat interfaces between components lead to strong coupling. Design the interfaces to be as slim as possible. But beware of `ambiguous interfaces`.

Prefer composition over inheritance
Inheritance increases coupling between parent and child, thereby limiting reuse.

Tangle-/cycle-free dependencies
The dependency graph of the elements of the architecture has no cycles, thus allowing locally bounded changes.

Evolvable architecture

Agile architecture
An architecture that supports agile software development by enabling the principles of the Agile Manifesto [6].

Allow change quickly
The architecture allows quick changes through flexibility and evolvability.

Verifiable at any time
The architecture can be verified (fulfils all quality aspects) at any time (e.g. every Sprint).

Rapid deployment
The architecture supports continuous and rapid deployment so that stakeholders can give feedback continuously.

Always working
The system is always working (probably with limited functionality) so that it is potentially shippable any time/at end of Sprint. Use assumptions, simplifications, simulators, shortcuts, hard-coding to build a walking skeleton.

Workflow
Use a top-down approach to find the architecture.

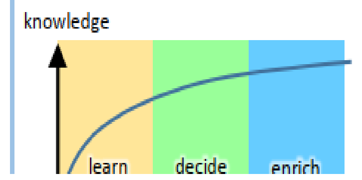
1. Context
What belongs to your system and what does not? Which external services will you use?

2. Break down into parts
Split the whole into parts by applying separation of concerns and the single-responsibility principle.

3. Communication
Which data flows through which call, message or event from one part to another? What are the properties of the channels (sync/async, reliability, …)

4. Repeat for each part
Repeat the above-mentioned three steps for each part as if it were your system.
A part is a bounded context, subsystem or component.

Defer decisions
Decide only things you have enough knowledge about. Otherwise find a way to defer the decision and build up more knowledge. A good architecture allows you to defer most decisions.



Architecture influencing forces
Quality attributes
The needed quality attributes (functionality, reliability, usability, efficiency, maintainability, portability, …) are the primary drivers for architectural decisions.

Team know-how and skills
The whole team understands and supports architecture and can make design decisions according to the architecture.

Easiness of implementation
How easy an envisioned architecture can be implemented is a quality attribute.

Cost of operations
Most costs of a software system accrue during operations, not implementation.

Risks
Every technology, library, and design decision has its risks.

Inherent opportunities
Things the architecture would allow us to do (but without investing any additional effort because we may never need it).

Technology churn
Availability of new (better) technologies, resulting in a need for architecture change.

Trade-offs
Designing an architecture comprises making trade-offs between conflicting goals. Trade-offs must reflect the priorities of quality attributes set by the stakeholders. Trade-offs should be documented and communicated to all stakeholders.

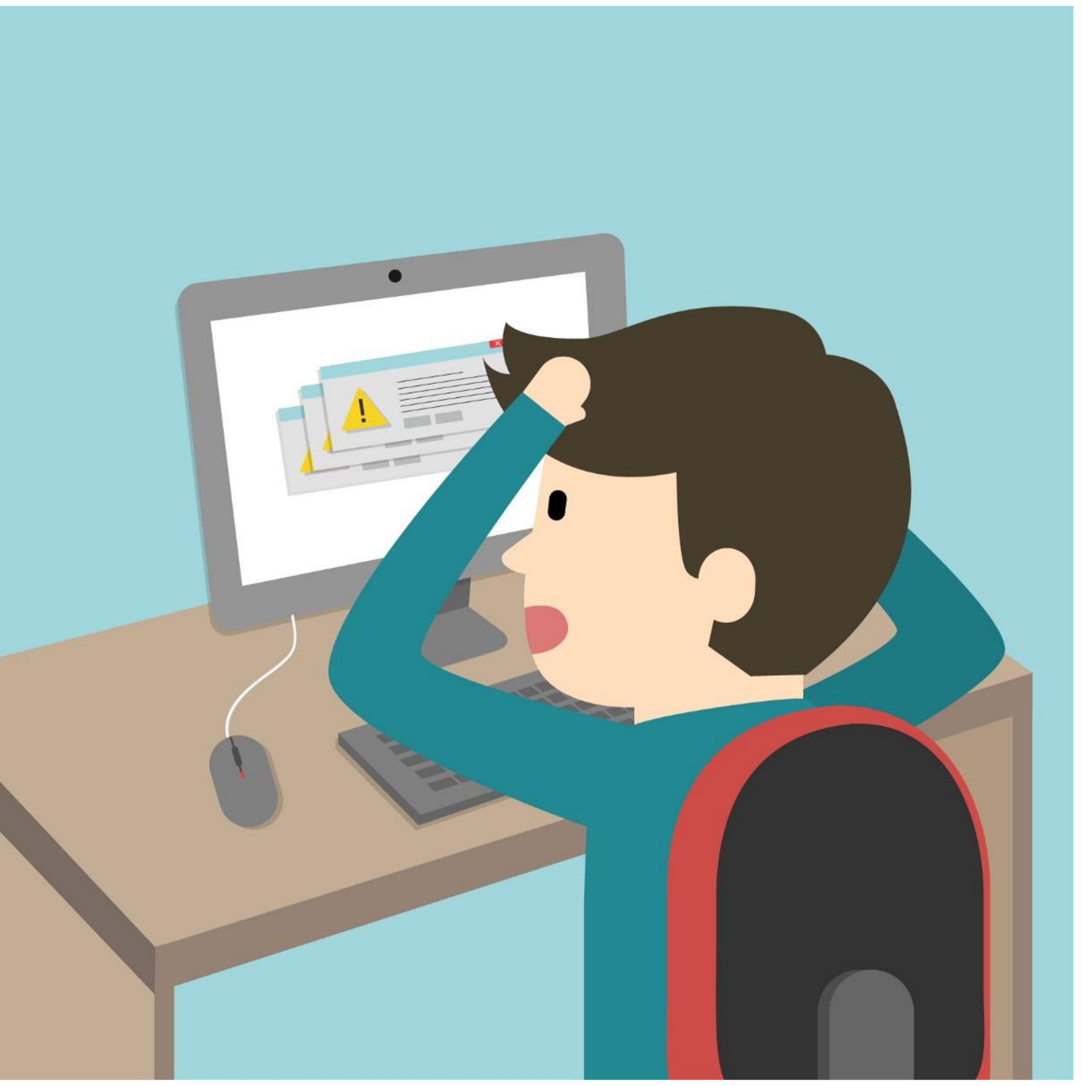
Architecture degrading forces
Architectural drift
Introduction of design decisions into a system's actual architecture that are not included in, encompassed by, or implied by the planned architecture.

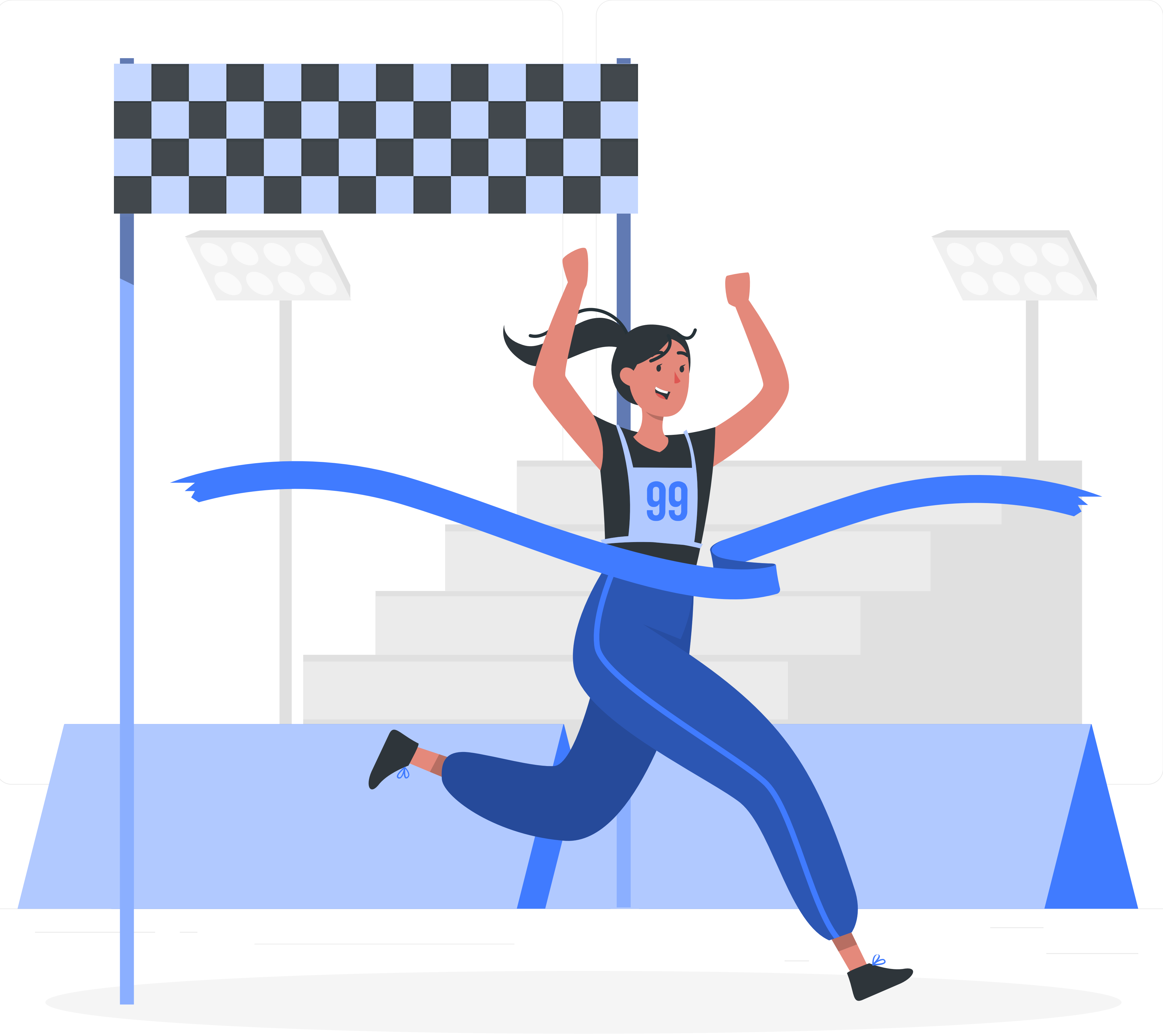
Architectural erosion
Introduction of design decisions into a system's actual architecture that violate its planned architecture.

Architecture killers
Split brain
Different parts of the system claim ownership of the same data or their interpretation resulting in inconsistencies and difficult synchronisation.

Coupling in space and time
E.g. shared code to remove duplication hinders independent advancements,

SE 350 Finish Line





Any Question

????????????????

How do you feel about the course?

Powered by Poll Everywhere

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

Please Send Your Question or Feedback…

Top

New

Powered by Poll Everywhere

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**