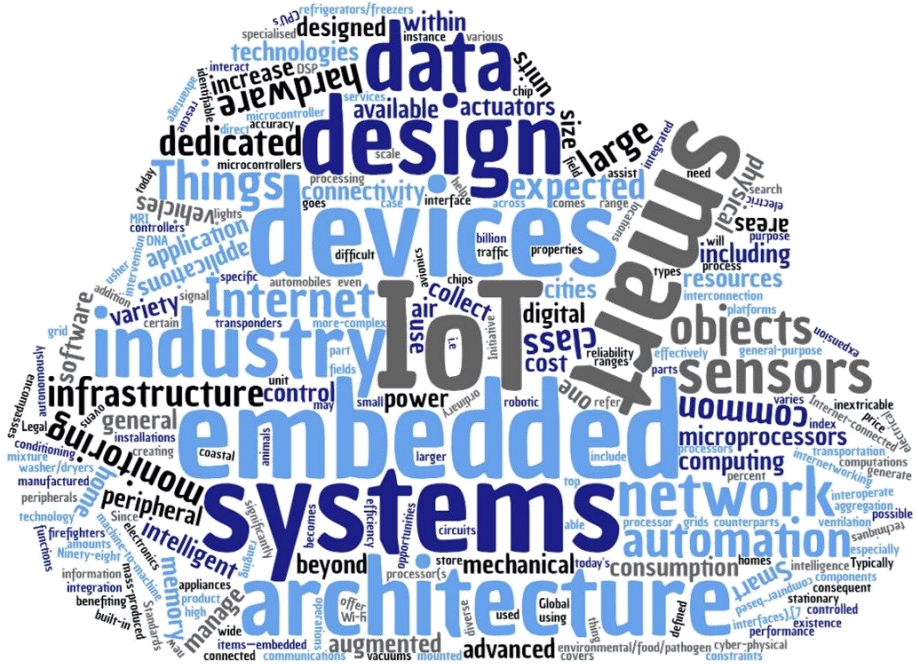
Associate Professorship of Embedded Systems and Internet of Things
Department of Electrical and Computer Engineering
Technical University of Munich

Lecture

Software Architecture for Distributed Embedded Systems

Gang of Four Patterns Intro & Structural Patterns

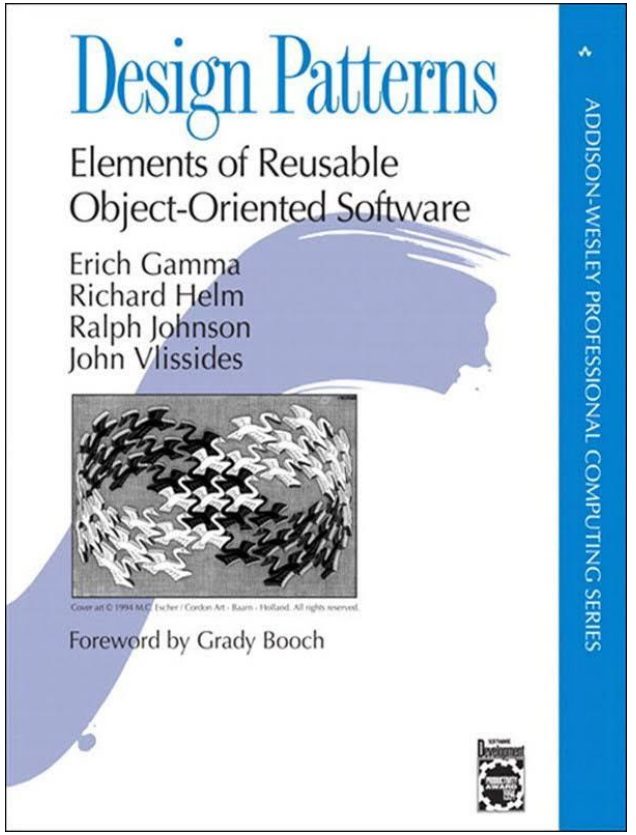
Prof. Dr. Sebastian Steinhorst



Design Patterns

- In software engineering, a **design pattern** is a general repeatable solution to a commonly occurring problem in software design.
- A design pattern isn't a finished design that can be transformed directly into code.
- It is a description or template for how to solve a problem that can be used in many different situations.

The original



Good starting point

Source: https://sourcemaking.com/design_patterns

Uses of Design Patterns

- Design patterns can speed up the development process by providing tested, proven development paradigms.
- Effective software design requires considering issues that may not become visible until later in the implementation.
- Reusing design patterns helps to prevent subtle issues that can cause major problems and improves code readability for coders and architects familiar with the patterns.

- Often, people only understand how to apply certain software design techniques to certain problems. These techniques are difficult to apply to a broader range of problems.
- Design patterns provide general solutions, documented in a format that doesn't require specifics tied to a particular problem.

- In addition, patterns allow developers to communicate using well-known, well understood names for software interactions.
- Common design patterns can be improved over time, making them more robust than ad-hoc designs.

Source: https://sourcemaking.com/design_patterns

Three Types of Patterns

Structural patterns

- Structural Design Patterns are Design Patterns that ease the design by identifying a simple way to realize relationships between entities. **They are mostly concerned with providing interfaces in a suitable form.**

Behavioral patterns

- Behavioral design patterns are design patterns that **identify common communication patterns between objects and realize these patterns**. By doing so, these patterns increase flexibility in carrying out this communication.

Creational patterns

- In software engineering, creational design patterns are design patterns that **deal with object creation mechanisms**, trying to create objects in a manner suitable to the situation. The basic form of object creation could result in design problems or added complexity to the design. Creational design patterns solve this problem by controlling this object creation.

Source: https://sourcemaking.com/design_patterns

Structural patterns

Adapter
Match interfaces of different classes

Flyweight
A fine-grained instance used for efficient sharing

Bridge
Separates an object's interface from its implementation

Proxy
An object representing another object

Composite
A tree structure of simple and composite objects

Decorator
Add responsibilities to objects dynamically

Facade
A single class that represents an entire subsystem

Source: GoF book, https://sourcemaking.com/design_patterns

Behavioral patterns

Chain of responsibility
A way of passing a request between a chain of objects

Command
Encapsulate a command request as an object

Interpreter
A way to include language elements in a program

Iterator
Sequentially access the elements of a collection

Mediator
Defines simplified communication between classes

Memento
Capture and restore an object's internal state

Observer
A way of notifying change to a number of classes

State
Alter an object's behavior when its state changes

Strategy
Encapsulates an algorithm inside a class

Template method
Defer the exact steps of an algorithm to a subclass

Visitor
Defines a new operation to a class without change

Source: GoF book, https://sourcemaking.com/design_patterns

Creational Patterns

Abstract Factory
Creates an instance of several families of classes

Builder
Separates object construction from its representation

Factory Method
Creates an instance of several derived classes

Prototype
A fully initialized instance to be copied or cloned

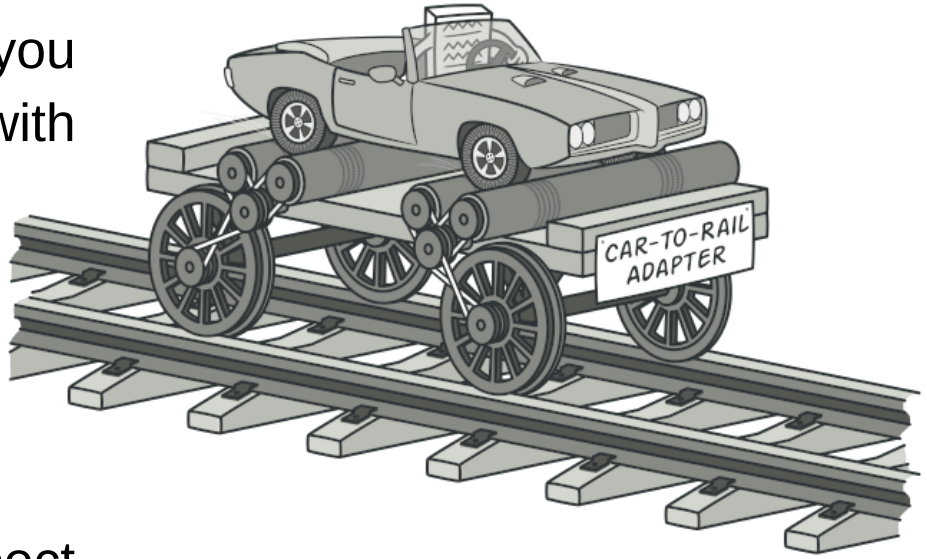
Singleton
A class of which only a single instance can exist

Source: GoF book, https://sourcemaking.com/design_patterns

Structural Pattern: Adapter (a.k.a. Wrapper)

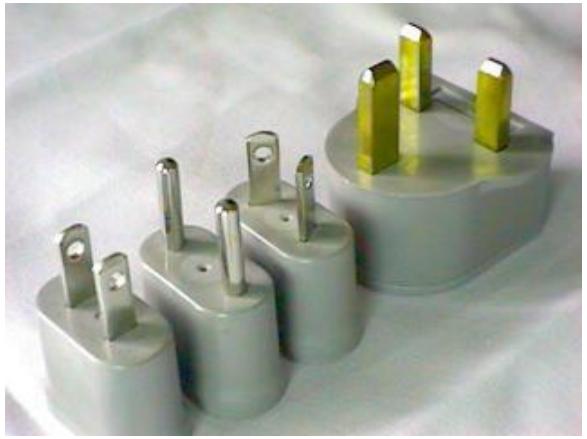
Problem

- An "off the shelf" component offers compelling functionality that you would like to reuse, but its "view of the world" is not compatible with the philosophy and architecture of the system currently being developed.



Intent

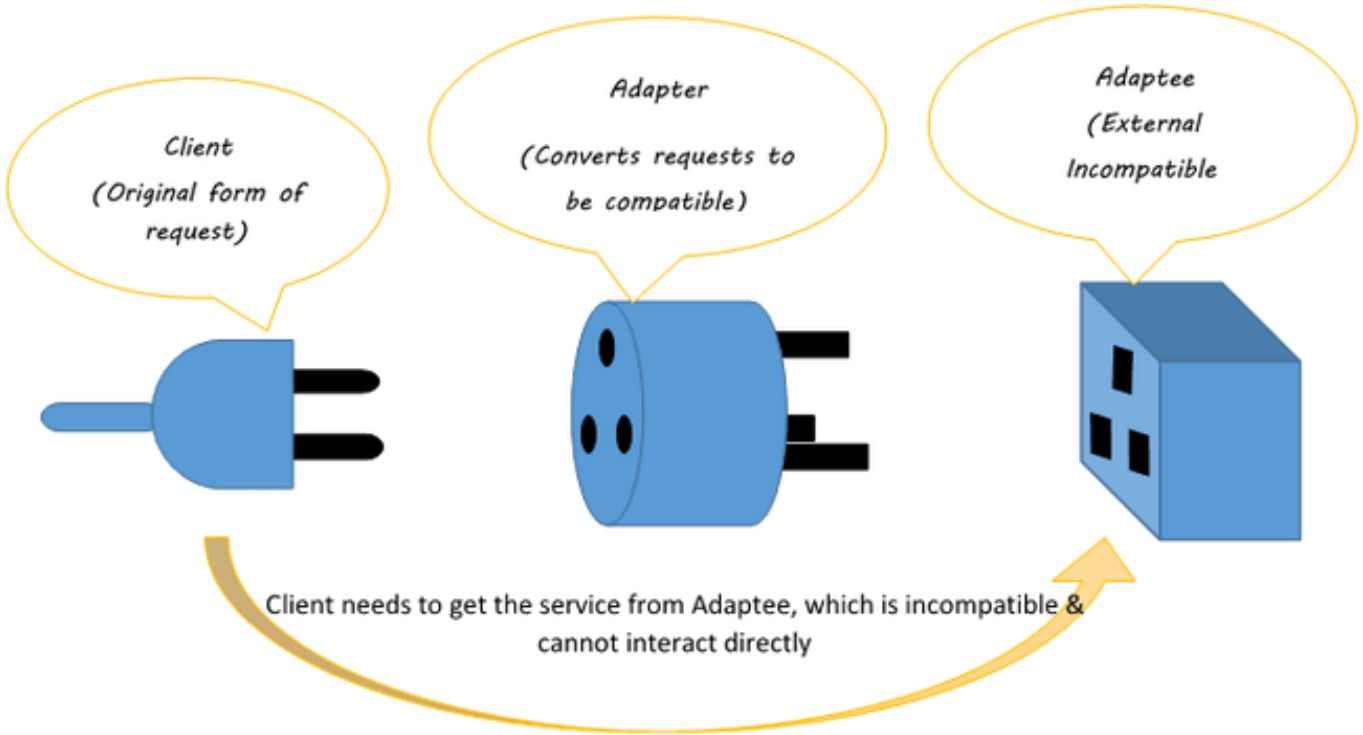
- Convert the interface of a class into another interface clients expect.
- Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- Wrap an existing class with a new interface.
- “Impedance match” an old component to a new system.



Source: https://sourcemaking.com/design_patterns/adapter, https://refactoring.guru/design-patterns/adapter

Adapter Discussion

- Reuse has always been painful and elusive. One reason has been the tribulation of designing something new, while reusing something old. There is always something not quite right between the old and the new. It may be physical dimensions or misalignment. It may be timing or synchronization. It may be unfortunate assumptions or competing standards.
- It is like the problem of inserting a two round prong electrical plug in a three square prong wall outlet – some kind of adapter or intermediary is necessary.



Source: GoF book, https://www.javagists.com/adapter-design-pattern, https://sourcemaking.com/design_patterns/adapter

Participants

Client: Uses the 'Target' interface to communicate with the outer world.

Target: The interface used by the client. This is client domain-specific.

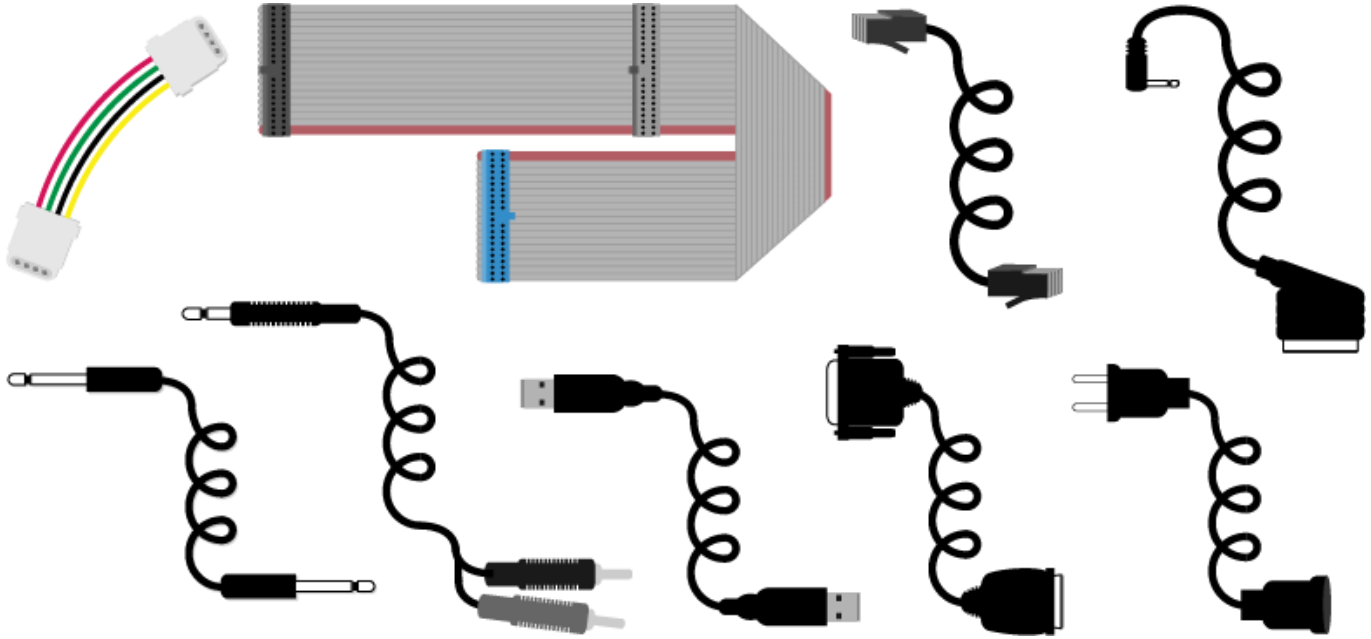
Adapter: This is the component which converts the requests and responses as required by the two incompatible systems.

Adaptee: The interface in the receiving end which associates or inherits from adapter class.

Adapter Discussion

- Adapter is about creating an intermediary abstraction that translates, or maps, the old component to the new system. Clients call methods on the Adapter object which redirects them into calls to the legacy component. This strategy can be implemented either with inheritance or with aggregation.

- Adapter functions as a wrapper or modifier of an existing class. It provides a different or translated view of that class.

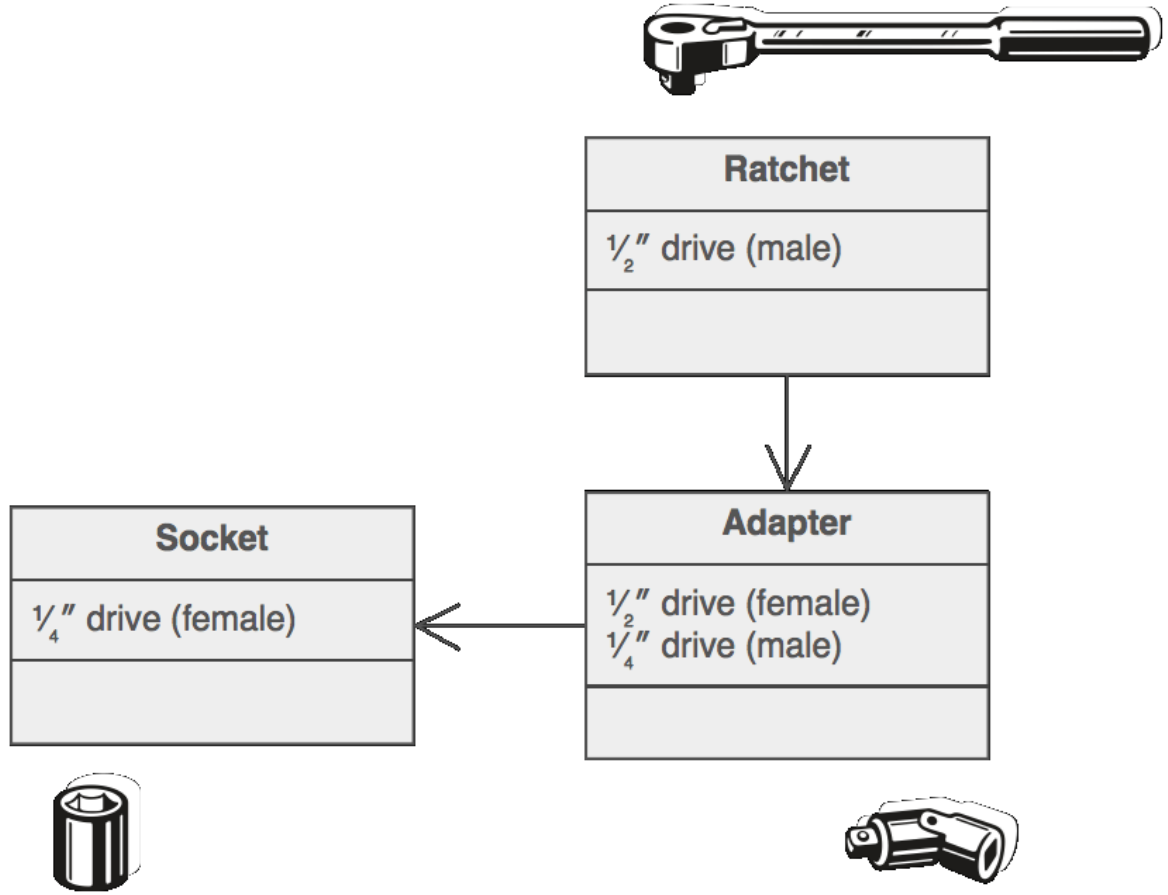


Source: https://sourcemaking.com/design_patterns/adapter

Example

The Adapter pattern allows otherwise incompatible classes to work together by converting the interface of one class into an interface expected by the clients.

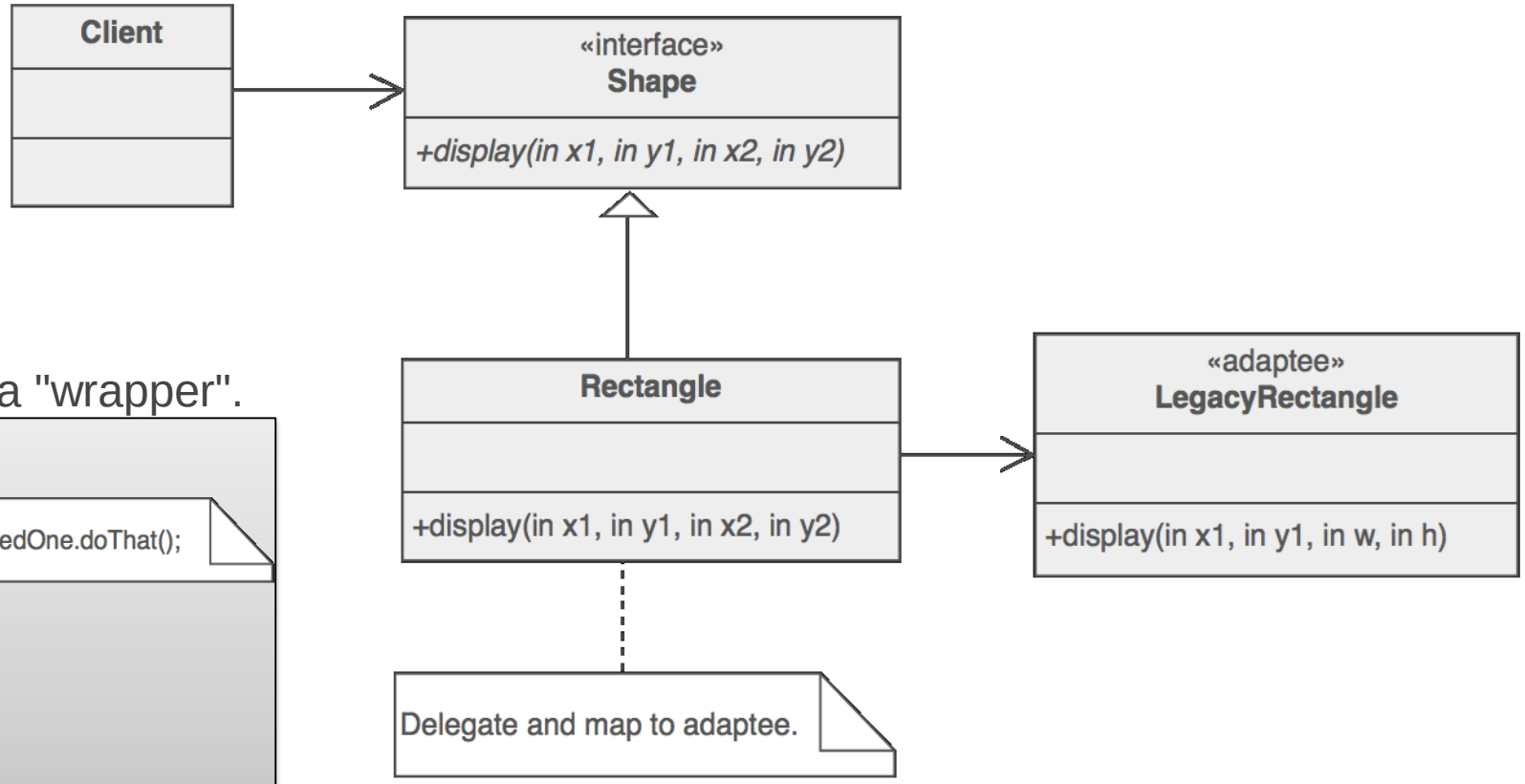
- Socket wrenches provide an example of the Adapter.
- A socket attaches to a ratchet, provided that the size of the drive is the same.
- Typical drive sizes in the United States are 1/2" and 1/4".
- Obviously, a 1/2" drive ratchet will not fit into a 1/4" drive socket unless an adapter is used.
- A 1/2" to 1/4" adapter has a 1/2" female connection to fit on the 1/2" drive ratchet, and a 1/4" male connection to fit in the 1/4" drive socket.



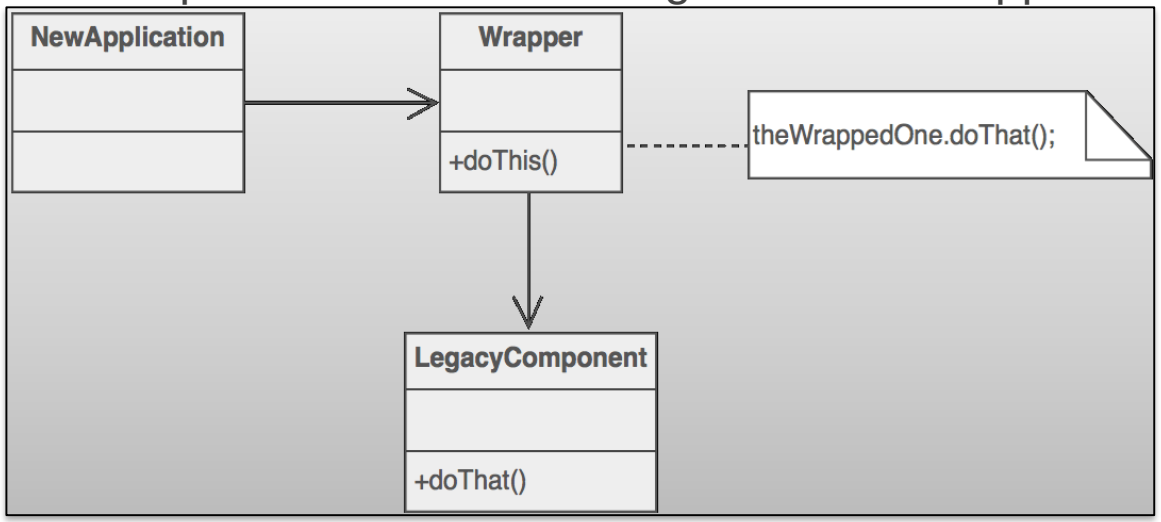
Source: https://sourcemaking.com/design_patterns/adapter

Structure

Below, a legacy Rectangle component's display() method expects to receive "x, y, w, h" parameters. But the client wants to pass "upper left x and y" and "lower right x and y". This incongruity can be reconciled by adding an additional level of indirection – i.e. an Adapter object “Rectangle”.



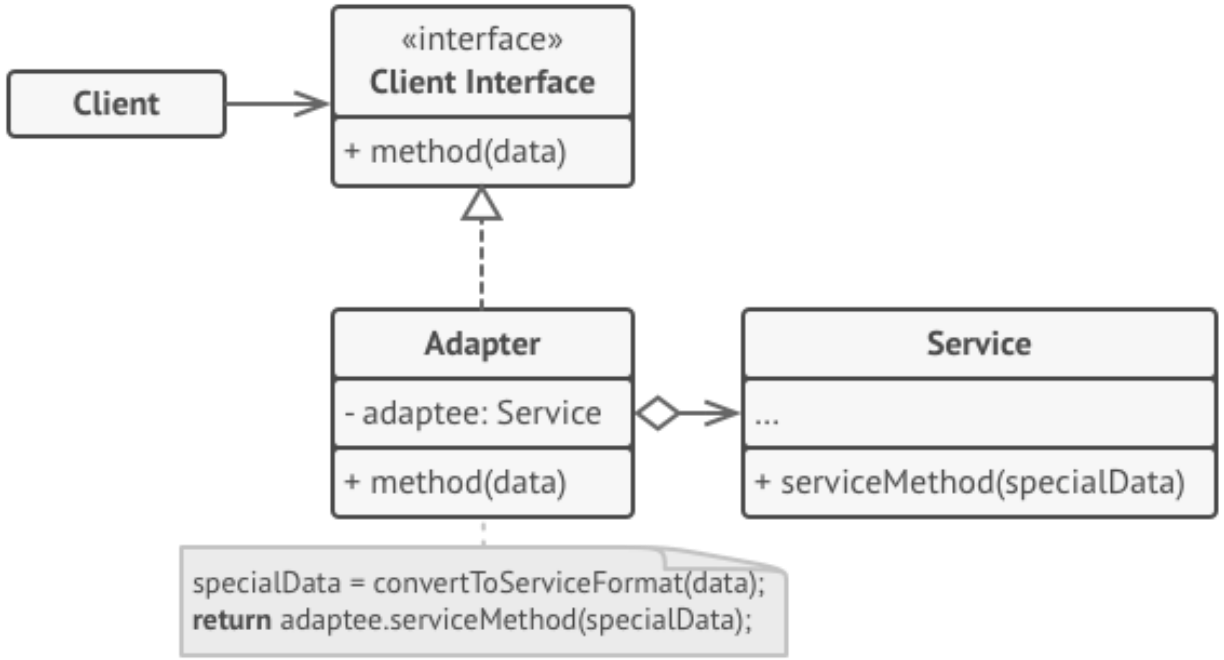
The Adapter could also be thought of as a "wrapper".



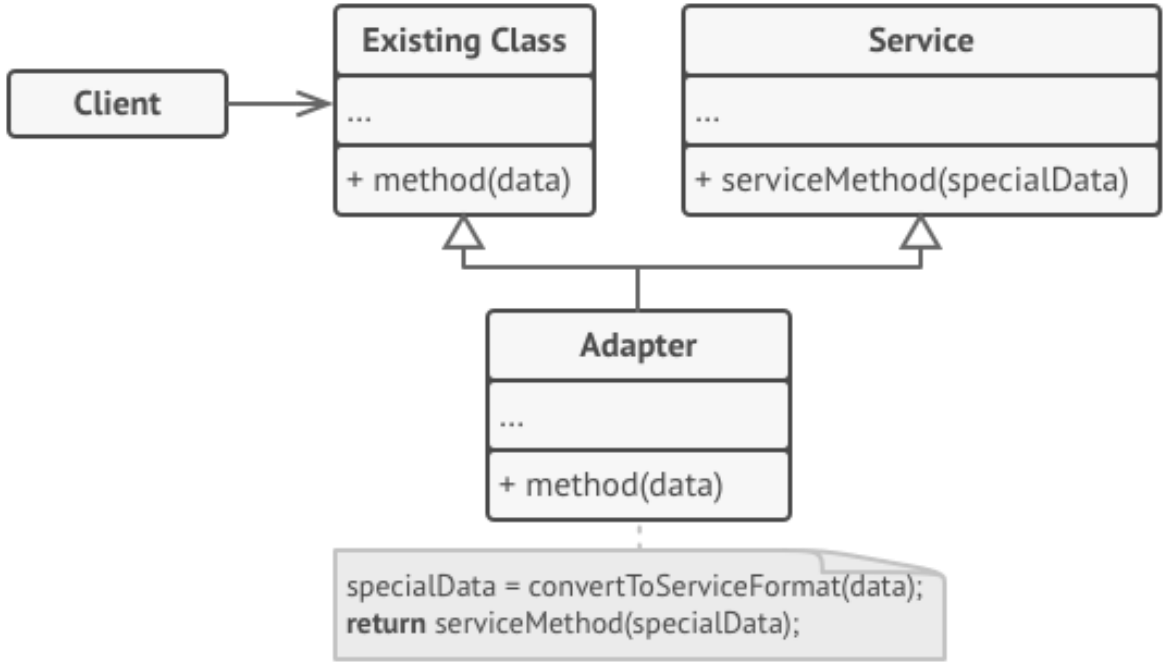
Source: https://sourcemaking.com/design_patterns/adapter

Object Adapter vs. Class Adapter

Object Adapter



Class Adapter



Multiple inheritance in Python:

```
class Existing_Class: pass
class Service: pass
class Adapter(Existing_Class, Service): pass
```

Source: https://refactoring.guru/design-patterns/adapter

Adapter Consequences

Class and object adapters have different trade-offs.

An object adapter

- lets a single Adapter work with many Adaptees—that is, the Adaptee itself and all of its subclasses (if any). The Adapter can also add functionality to all Adaptees at once.
- makes it harder to override Adaptee behavior. It will require subclassing Adaptee and making Adapter refer to the subclass rather than the Adaptee itself.

A class adapter

- adapts Adaptee to Target by committing to a concrete Adaptee class. As a consequence, a class adapter won't work when we want to adapt a class *and* all its subclasses.
- lets Adapter override some of Adaptee's behavior, since Adapter is a subclass of Adaptee.
- introduces only one object, and no additional pointer indirection is needed to get to the adaptee.

Source: GoF book, https://sourcemaking.com/design_patterns/adapter

Related Patterns

- **Bridge** has a structure similar to an object adapter, but Bridge has a different intent: It is meant to separate an interface from its implementation so that they can be varied easily and independently. An adapter is meant to change the interface of an *existing* object.

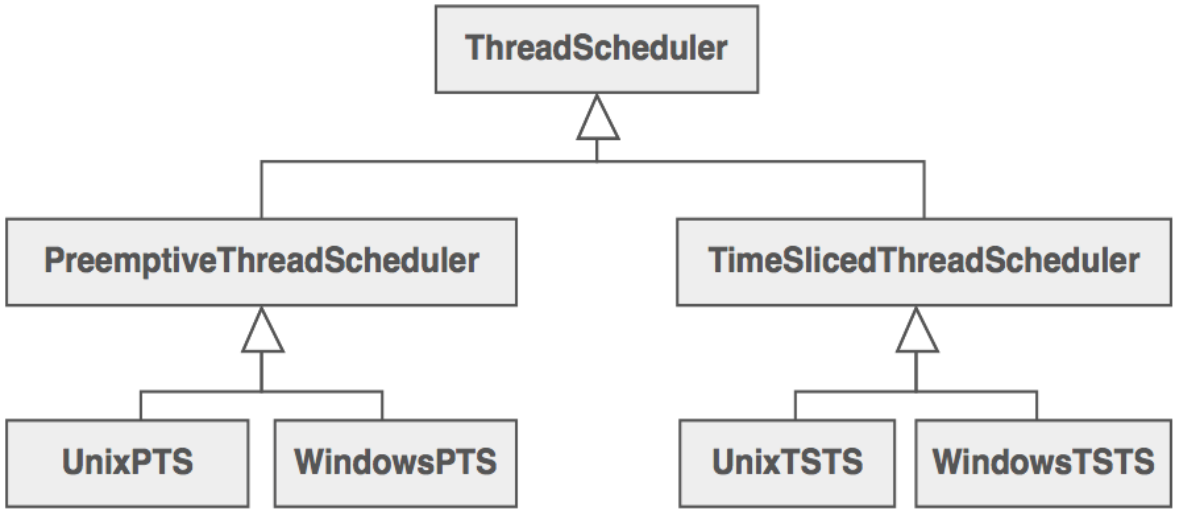
- **Decorator** enhances another object without changing its interface. A decorator is thus more transparent to the application than an adapter is. As a consequence, Decorator supports recursive composition, which isn't possible with pure adapters.

- **Proxy** defines a representative or surrogate for another object and does not change its interface.

Source: GoF book, https://sourcemaking.com/design_patterns/adapter

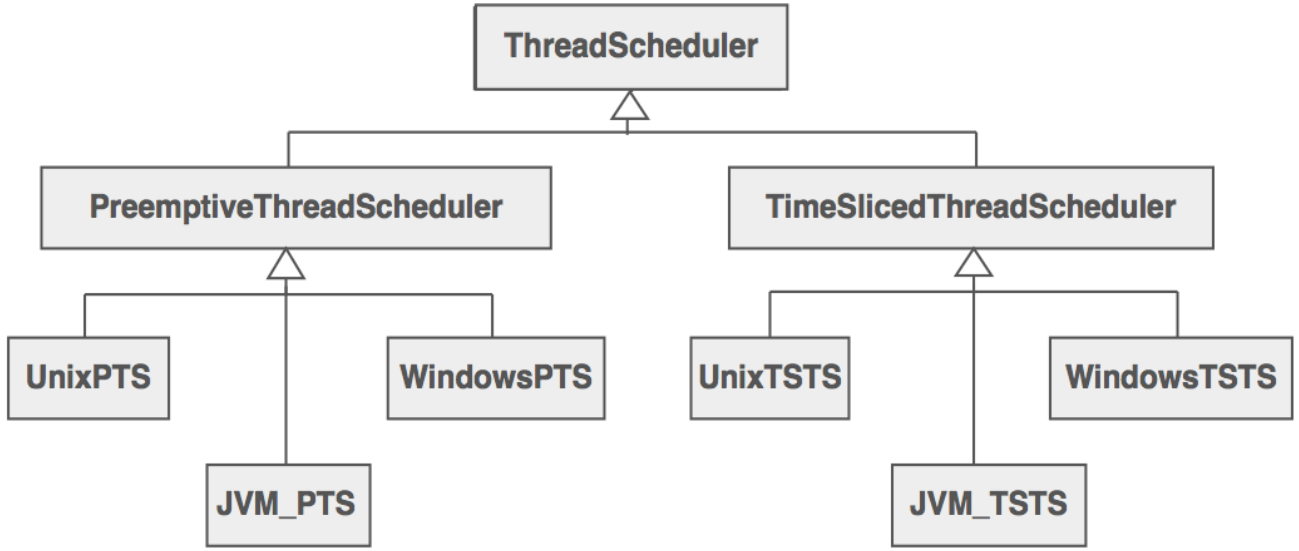
Motivation of Structural Pattern: Bridge

Consider the domain of "thread scheduling".



- There are two types of thread schedulers, and two types of operating systems or "platforms".
- Given this approach to specialization, we have to define a class for each permutation of these two dimensions.

- If we add a new platform (say ... Java's Virtual Machine), what would our hierarchy look like?
- The number of classes we would have to define is the product of the number of scheduling schemes and the number of platforms.



Source: GoF book, https://sourcemaking.com/design_patterns/bridge

Structural Pattern: Bridge

Problem

- When an abstraction can have one of several possible implementations, the usual way to accommodate them is to use inheritance.
- An abstract class defines the interface to the abstraction, and concrete subclasses implement it in different ways.
- But this approach isn't always flexible enough: Inheritance binds an implementation to the abstraction permanently, which makes it difficult to modify, extend, and reuse abstractions and implementations independently.
- Locks in compile-time binding between interface and implementation.
- The abstraction and implementation cannot be independently extended or composed.

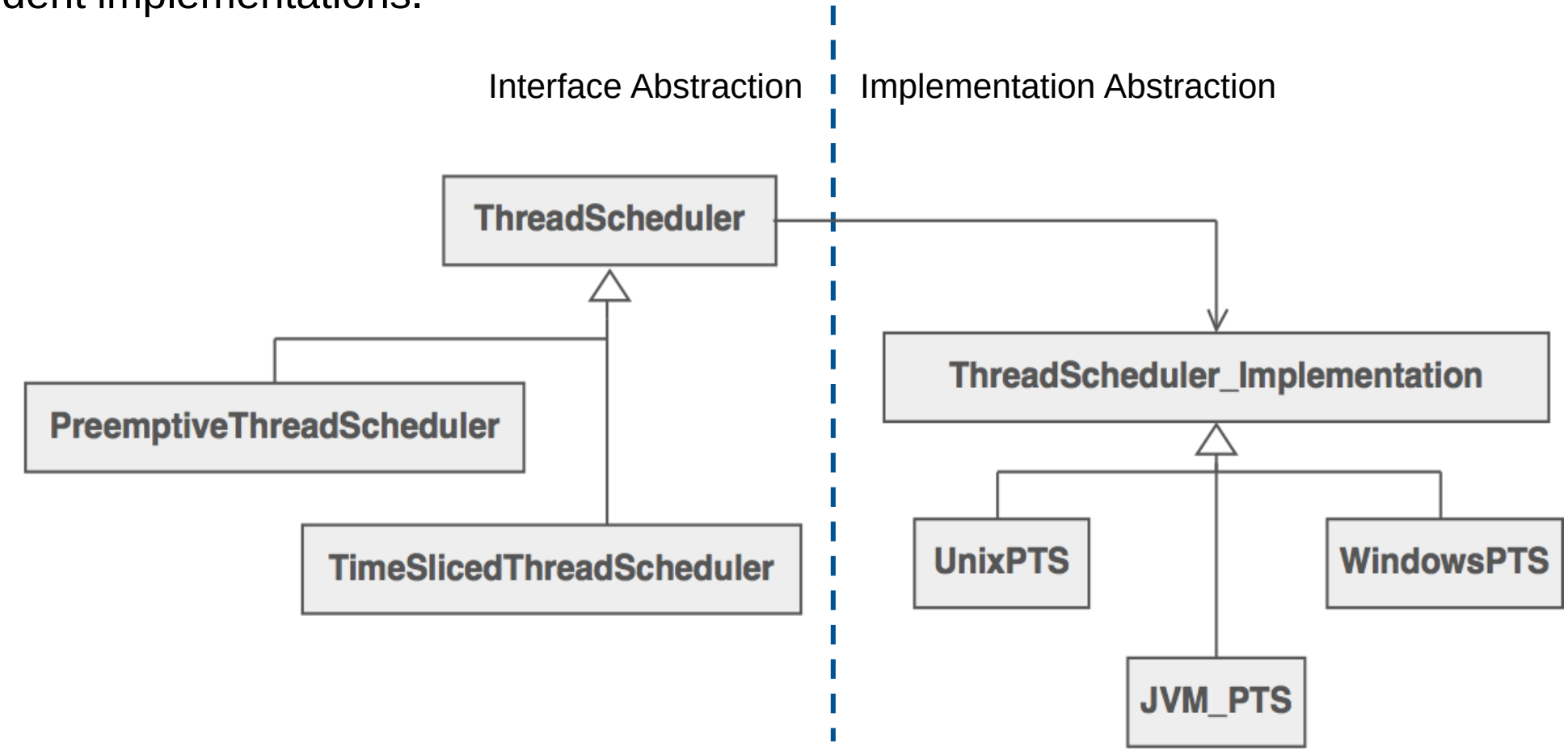
Intent

- Decouple an abstraction from its implementation so that the two can vary independently.
- Publish interface in an inheritance hierarchy, and bury implementation in its own inheritance hierarchy.
- Beyond encapsulation, to insulation.

Source: GoF book, https://sourcemaking.com/design_patterns/bridge

Bridge Pattern separates abstraction of interface and implementation

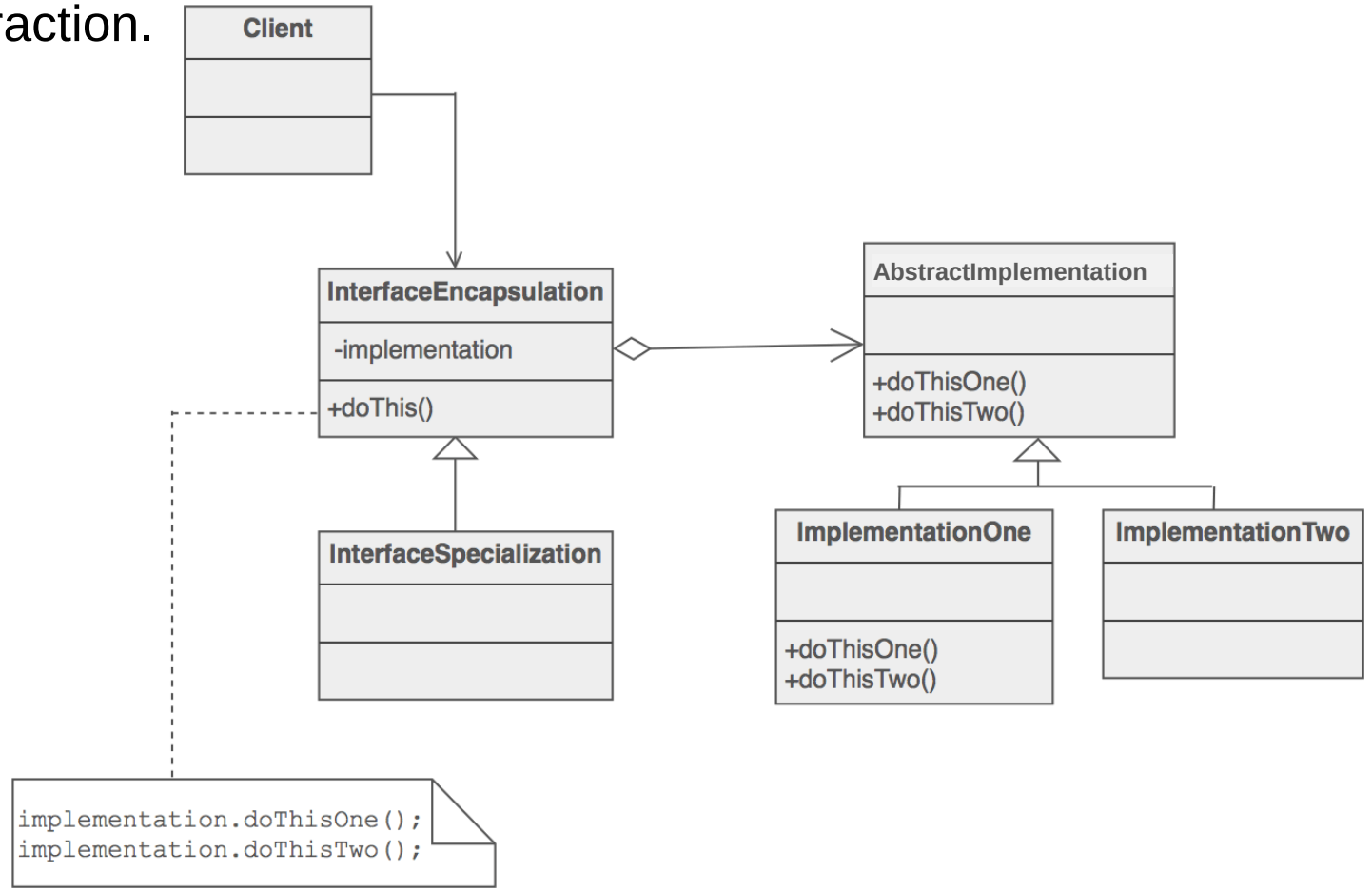
The Bridge design pattern proposes refactoring this exponentially explosive inheritance hierarchy into two orthogonal hierarchies – one for platform-independent abstractions, and the other for platform-dependent implementations.



Source: GoF book, https://sourcemaking.com/design_patterns/bridge

Structure

- The Client doesn't want to deal with platform-dependent details. The Bridge pattern encapsulates this complexity behind an abstraction "wrapper".
- Bridge emphasizes identifying and decoupling "interface" abstraction from "implementation" abstraction.



Participants

InterfaceEncapsulation (Abstraction)
- defines the abstraction's interface
- maintains a reference to an object of type Implementor

InterfaceSpecialization (RefinedAbstraction)
- Extends the interface defined by Abstraction

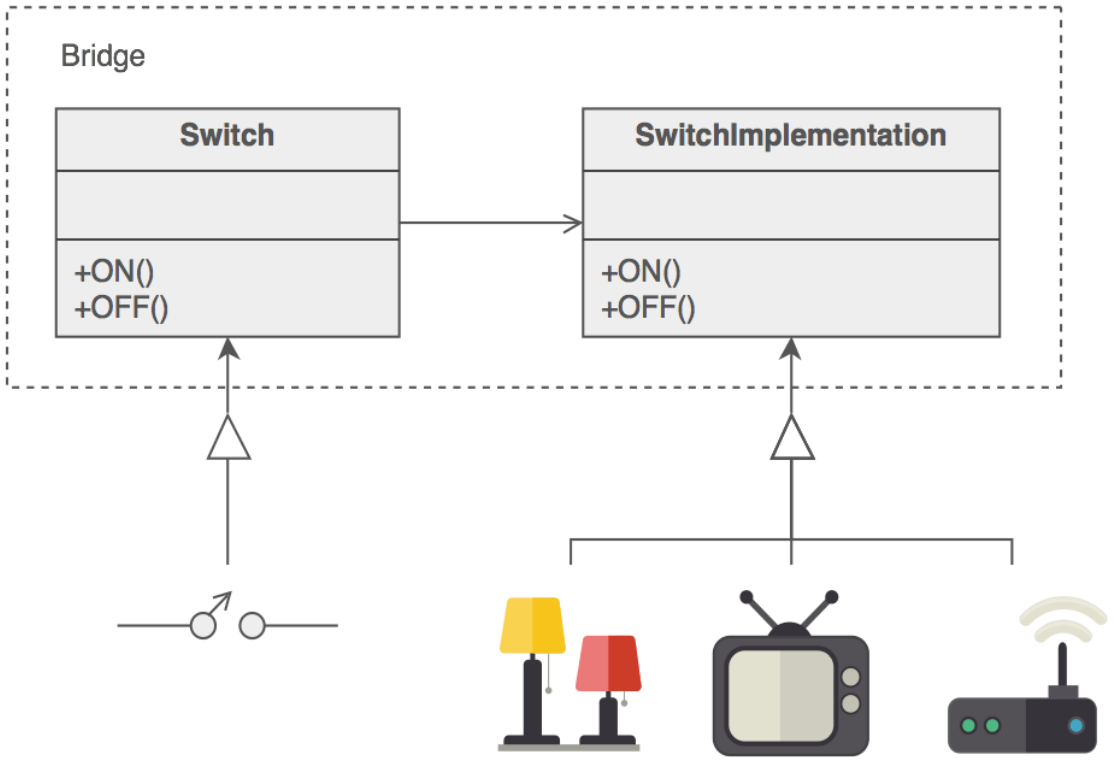
AbstractImplementation (Implementor)
- defines the interface for implementation classes. This interface doesn't have to correspond exactly to Abstraction's interface

Implementation (ConcreteImplementor)
- implements the Implementor interface and defines its concrete implementation

Source: GoF book, https://sourcemaking.com/design_patterns/bridge

Example

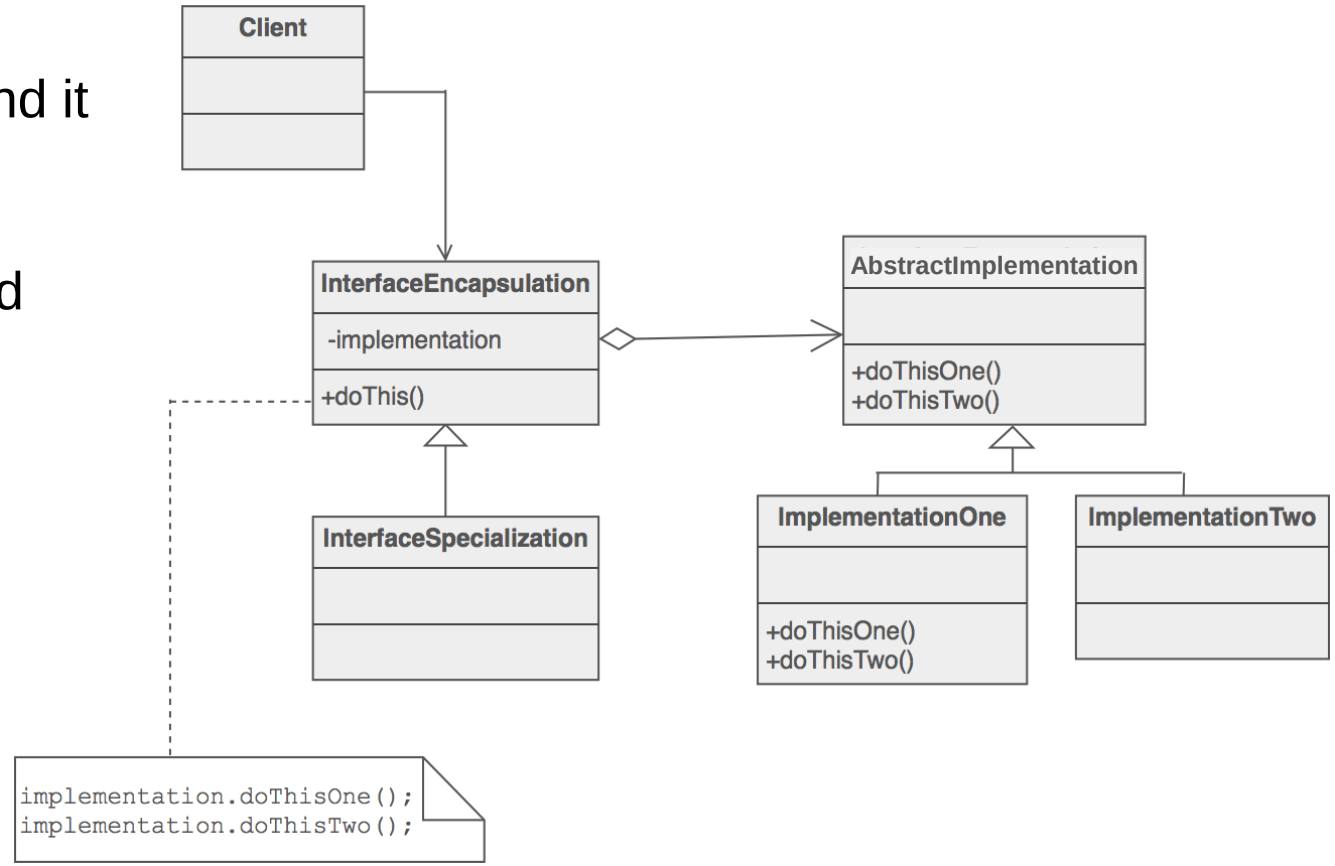
- The Bridge pattern decouples an abstraction from its implementation, so that the two can vary independently.
- A household switch controlling lights, ceiling fans, etc. is an example of the Bridge.
- The purpose of the switch is to turn a device on or off.
- The actual switch can be implemented as a pull chain, simple two position switch, or a variety of dimmer switches.



Source: GoF book, https://sourcemaking.com/design_patterns/bridge

Discussion

- Decompose the component's interface and implementation into orthogonal class hierarchies.
- The interface class contains a pointer to the abstract implementation class.
- This pointer is initialized with an instance of a concrete implementation class, but all subsequent interaction from the interface class to the implementation class is limited to the abstraction maintained in the implementation base class.
- The client interacts with the interface class, and it in turn "delegates" all requests to the implementation class.
- The interface object is the "handle" known and used by the client; while the implementation object, or "body", is safely encapsulated to ensure that it may continue to evolve, or be entirely replaced (or shared) at run-time.



Source: GoF book, https://sourcemaking.com/design_patterns/bridge

Bridge Pattern: Check list

1. Decide if two orthogonal dimensions exist in the domain. These independent concepts could be: abstraction/platform, or domain/infrastructure, or front-end/back-end, or interface/implementation.

2. Design the separation of concerns: what does the client want, and what do the platforms provide.

3. Design a platform-oriented interface that is minimal, necessary, and sufficient. Its goal is to decouple the abstraction from the platform.

4. Define a derived class of that interface for each platform.

5. Create the abstraction base class that "has a" platform object and delegates the platform-oriented functionality to it.

6. Define specializations of the abstraction class if desired.

Source: GoF book, https://sourcemaking.com/design_patterns/bridge

Patterns related to Bridge

- Adapter makes things work after they're designed; Bridge makes them work before they are.
- Bridge is designed up-front to let the abstraction and the implementation vary independently while Adapter is retrofitted to make unrelated classes work together.
- State, Strategy, Bridge (and to some degree Adapter) have similar solution structures. They all share elements of the "handle/body" idiom. They differ in intent - that is, they solve different problems.
- The structure of State and Bridge are identical (except that Bridge admits hierarchies of envelope classes, whereas State allows only one). The two patterns use the same structure to solve different problems: State allows an object's behavior to change along with its state, while Bridge's intent is to decouple an abstraction from its implementation so that the two can vary independently.
- If interface classes delegate the creation of their implementation classes (instead of creating/coupling themselves directly), then the design usually uses the Abstract Factory pattern to create the implementation objects.

Source: GoF book, https://sourcemaking.com/design_patterns/bridge

Structural Pattern: Proxy (a.k.a Surrogate)

Problem

- You need to support resource-hungry objects, and you do not want to instantiate such objects unless and until they are actually requested by the client.

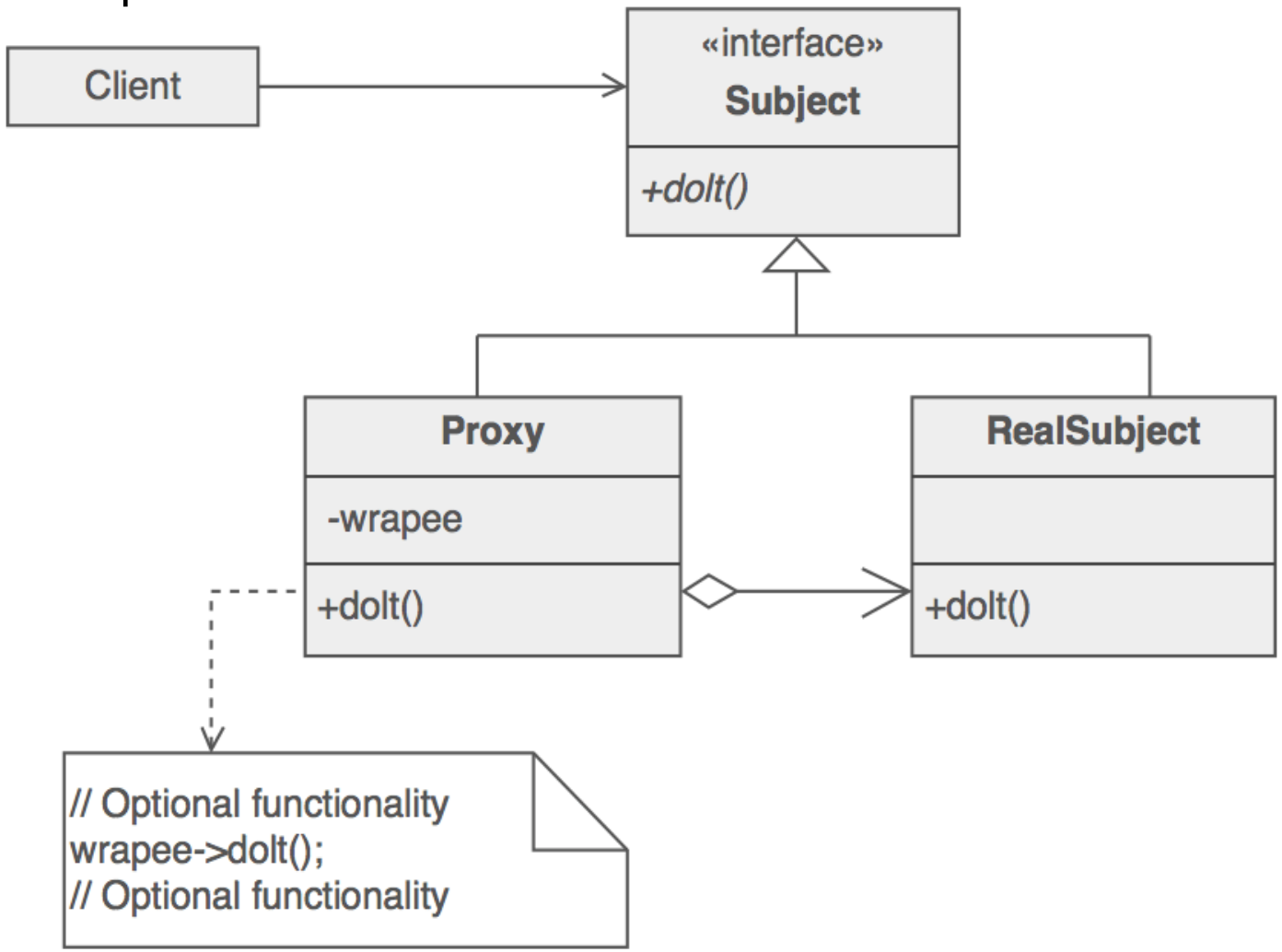
Intent

- Provide a surrogate or placeholder for another object to control access to it.
- Use an extra level of indirection to support distributed, controlled, or intelligent access.
- Add a wrapper and delegation to protect the real component from undue complexity.

Source: GoF book, https://sourcemaking.com/design_patterns/proxy

Structure

By defining a Subject interface, the presence of the Proxy object standing in place of the RealSubject is transparent to the client.



Source: GoF book, https://sourcemaking.com/design_patterns/proxy

Participants

Proxy

- maintains a reference that lets the proxy access the real subject.
- provides an interface identical to Subject's so that a proxy can be substituted for the real subject.
- controls access to the real subject and may be responsible for creating and deleting it.

Subject

- defines the common interface for RealSubject and Proxy so that a Proxy can be used anywhere a RealSubject is expected.

RealSubject

- defines the real object that the proxy represents

Applicability

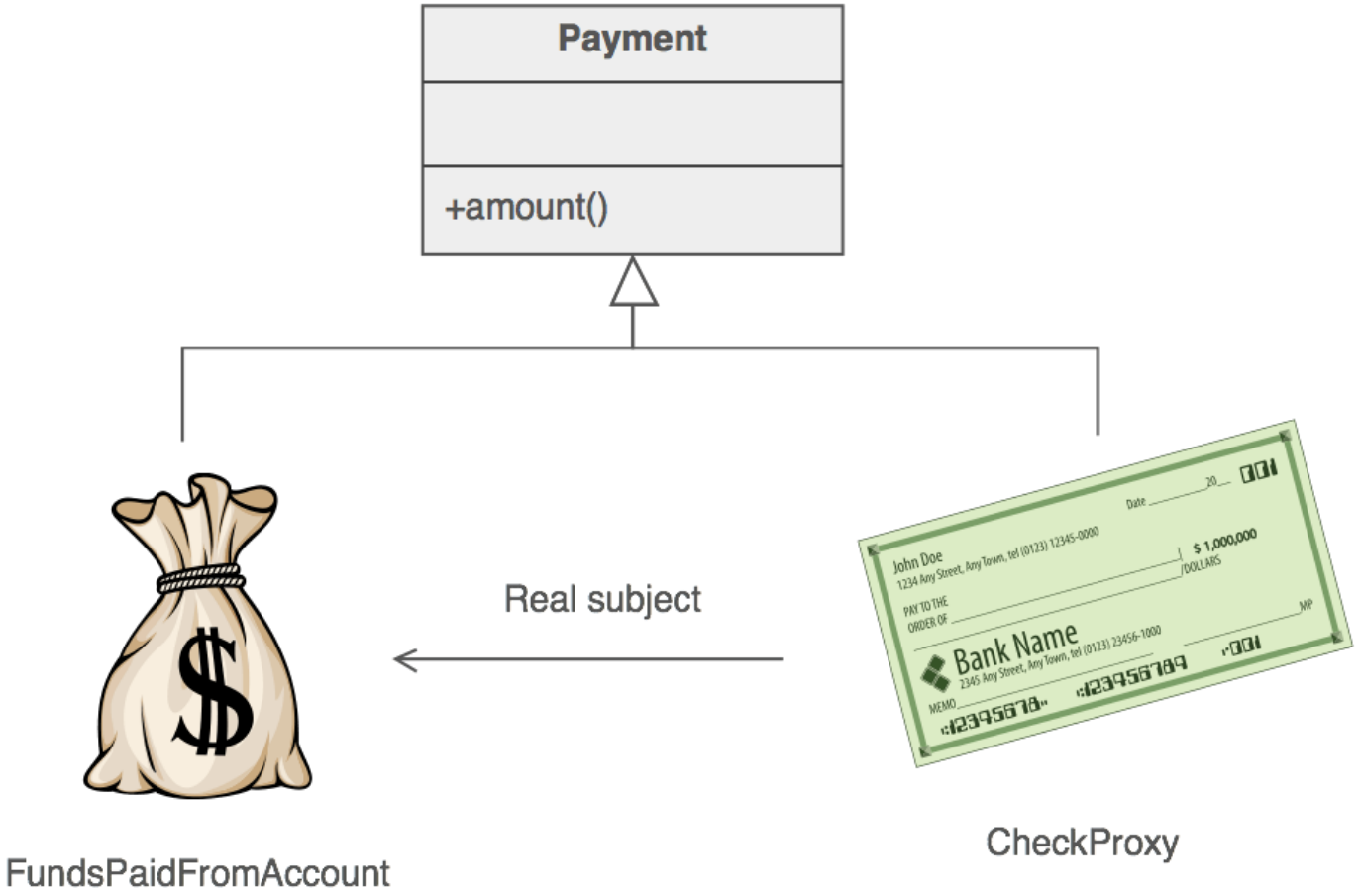
There are four common situations in which the Proxy pattern is applicable.

1. A **virtual proxy** is a placeholder for "expensive to create" objects. The real object is only created when a client first requests/accesses the object.
2. A **remote proxy** provides a local representative for an object that resides in a different address space. This is what the "stub" code in RPC and CORBA provides.
3. A **protective proxy** controls access to a sensitive master object. The "surrogate" object checks that the caller has the access permissions required prior to forwarding the request.
4. A **smart proxy** interposes additional actions when an object is accessed. Typical uses include:
   i. Counting the number of references to the real object so that it can be freed automatically when there are no more references (aka smart pointer),
   ii. Loading a persistent object into memory when it's first referenced,
   iii. Checking that the real object is locked before it is accessed to ensure that no other object can change it.

Source: GoF book, https://sourcemaking.com/design_patterns/proxy

Example

- The Proxy provides a surrogate or place holder to provide access to an object.
- A check or bank draft is a proxy for funds in an account.
- A check can be used in place of cash for making purchases and ultimately controls access to cash in the issuer's account.



Source: GoF book, https://sourcemaking.com/design_patterns/proxy

Check List

1. Identify the leverage or "aspect" that is best implemented as a wrapper or surrogate.
2. Define an interface that will make the proxy and the original component interchangeable.
3. Consider defining a Factory that can encapsulate the decision of whether a proxy or original object is desirable.
4. The wrapper class holds a pointer to the real class and implements the interface.
5. The pointer may be initialized at construction, or on first use.
6. Each wrapper method contributes its leverage, and delegates to the wrappee object.

Source: GoF book, https://sourcemaking.com/design_patterns/proxy

Relation to other Patterns

- Adapter:
  - An adapter provides a different interface to the object it adapts.
  - By contrast, a proxy provides the same interface as its subject.
  - However, a proxy used for access protection might refuse to perform an operation that the subject will perform, so its interface may be effectively a subset of the subject's.

- Decorator:
  - Although decorators can have similar implementations as proxies, decorators have a different purpose.
  - A decorator adds one or more responsibilities to an object, whereas a proxy controls access to an object.
  - Proxies vary in the degree to which they are implemented like a decorator.
  - A protection proxy might be implemented exactly like a decorator.
  - On the other hand, a remote proxy will not contain a direct reference to its real subject but only an indirect reference, such as "host ID and local address on host."
  - A virtual proxy will start off with an indirect reference such as a file name but will eventually obtain and use a direct reference.

Source: GoF book, https://sourcemaking.com/design_patterns/proxy

Wrap-up Structural Patterns

- Structural design patterns all deal with enhancing the use of **interfaces** to achieve certain improvements in software architecture.
- We looked at the Adapter, Bridge and Proxy patterns as examples for structural patterns.
- Adapter makes otherwise incompatible interfaces of classes compatible such that they can interoperate.
- Bridge separates the abstraction from the implementation, such that the implementation can develop independently from the organization of the interface.
- Proxy makes objects only available at first use or filters other properties. This functionality is hidden to the external interface.