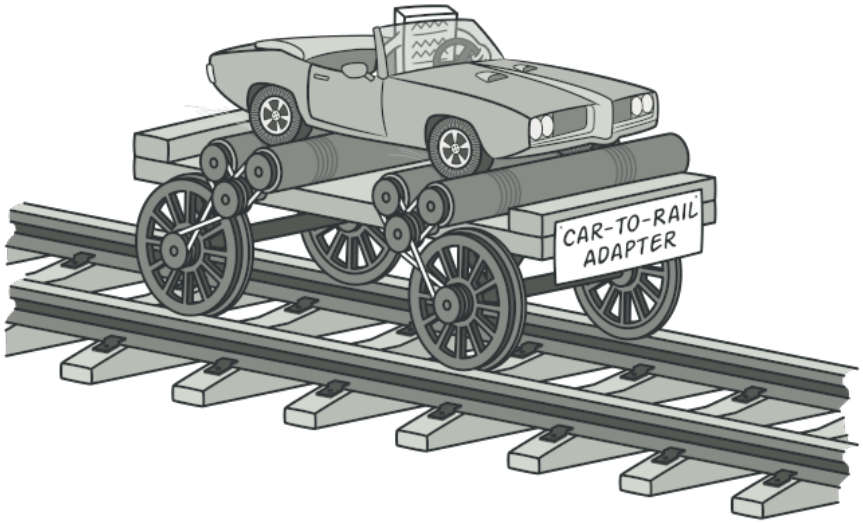
Topic 6 - Structural Design Patterns (Gamma et. al)

Adapter

Intent:

- Convert one class/interface to another expected type so we can reuse old classes in a new context



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

Applicability:

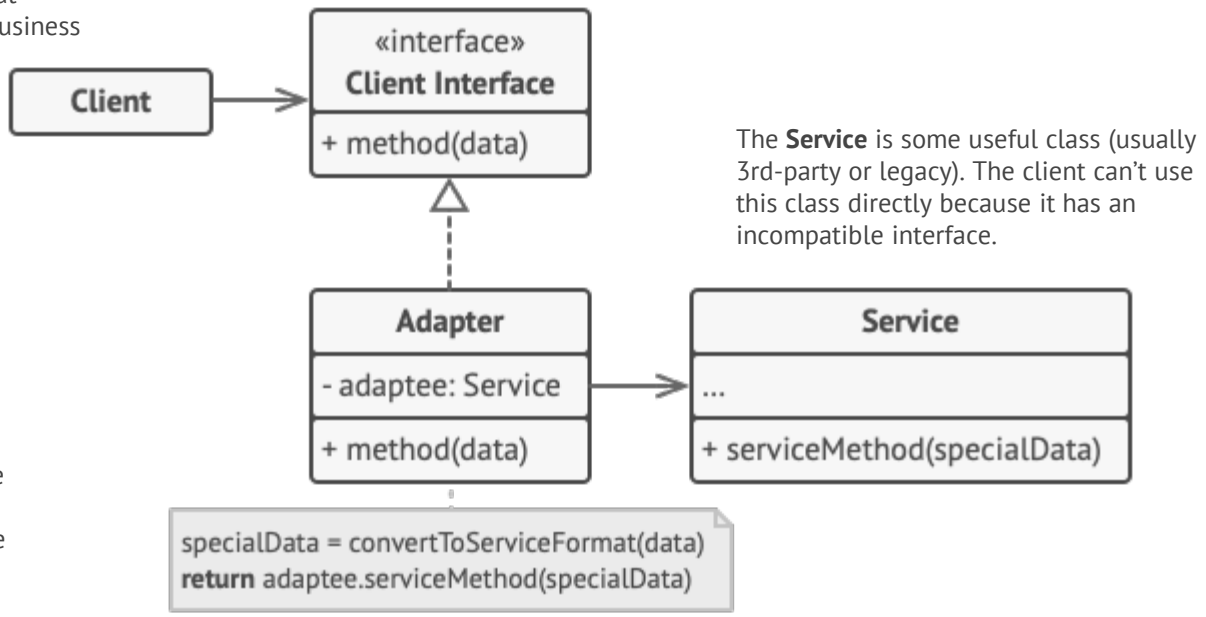
- When a class provides the required functionality but has an interface that is not suitable
  - especially relevant when source of existing class not available
  - but really better to leave existing source as is where possibly anyway!

Object Adapter

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

The **Client Interface** describes a protocol that other classes must follow to be able to collaborate with the client code.

The **Client** is a class that contains the existing business logic of the program.



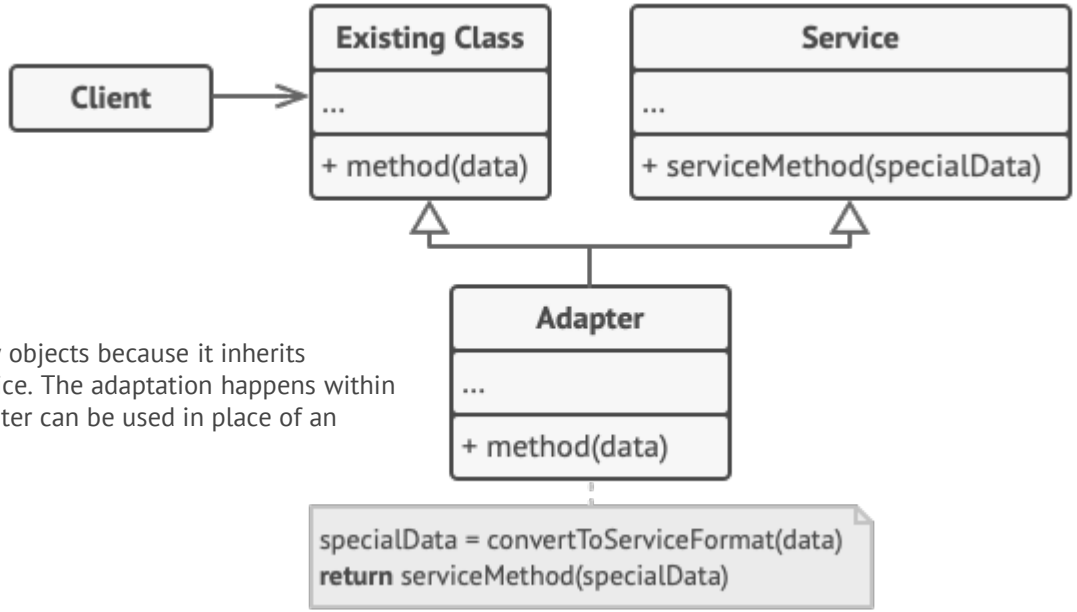
The **Service** is some useful class (usually 3rd-party or legacy). The client can't use this class directly because it has an incompatible interface.

The client code doesn't get coupled to the concrete adapter class as long as it works with the adapter via the client interface. Thanks to this, you can introduce new types of adapters into the program without breaking the existing client code. This can be useful when the interface of the service class gets changed or replaced: you can just create a new adapter class without changing the client code.

The **Adapter** is a class that's able to work with both the client and the service: it implements the client interface, while wrapping the service object. The adapter receives calls from the client via the adapter interface and translates them into calls to the wrapped service object in a format it can understand.

Class Adapter

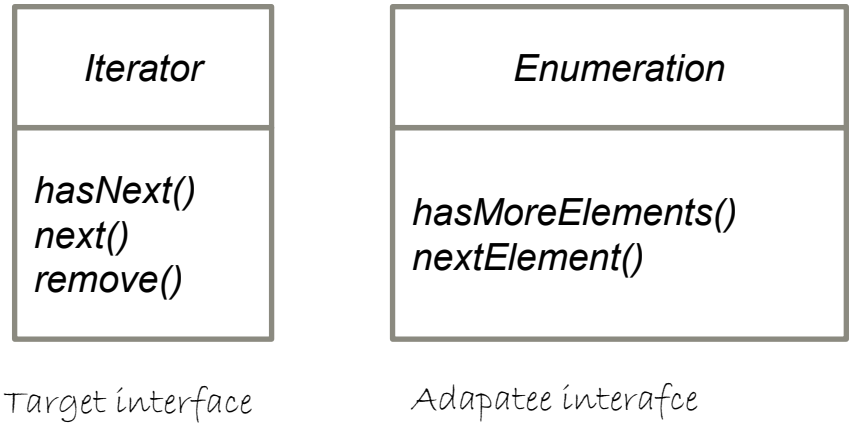
https://refactoring.guru/design-patterns/adapter



The **Class Adapter** doesn't need to wrap any objects because it inherits behaviors from both the client and the service. The adaptation happens within the overridden methods. The resulting adapter can be used in place of an existing client class.

Adapters: Example

- The early (pre Java 1.2) collection types (Vector, Stack) methods returned a `java.util.Enumeration` interface.
- `Enumeration` interface allows stepping through the elements of a collection without knowing the specifics of how they are managed.
- More recent versions use Iterator instead.
- Sometimes legacy code using `Enumeration` must be adapted to use `java.util.Iterator` with code that does not support `Enumeration`.
- See `adapter.enumeration.*` example



Consequences

- A Class Adapter may override an Adaptee's behaviour, since Adaptor is a subclass and calls Adaptee methods directly
- Class Adapter requires no additional indirection (references)
- Class Adapter requires you to choose a specific concrete adaptee subclass to extend
- Object adapter is more flexible since you can easily change the adaptee to be any subclass (e.g. by passing the adaptee to the adapter constructor)
- Java does not support a full class based adapter where we extend two classes so we can have a hybrid where we extend the Adaptee and implement an existing interface which we adapt
  - not that useful in practice since you have to delegate anyway
  - but allows you to treat the adapted object as the old type as well if desired
- The amount of work required by the adapter can range from very little (forwarding calls) to very much (implementing functionality not available in the adaptee)

Bridge Pattern

Intent:

- Decouple an abstraction from its implementation so that the two can vary independently
- Decouple multiple changing facets into separate hierarchies (nested generalisations/parallel hierarchies)

Applicability: Use it when

- you want to avoid a permanent binding between an abstraction and its implementation – for example when implementation must be selected or switched at run-time.
- ***both abstractions and implementations must be extensible by subclassing*** and your hierarchy is getting bloated with combinations (similar to problem that Decorator addresses)
- changes in implementation should be invisible to clients

- JAVA API example, JDBC has a bridge between `java.sql.Connection` and `java.sql.Driver` (via `the java.sql.DriverManager` class and `getConnection()` method)

Bridge Pattern

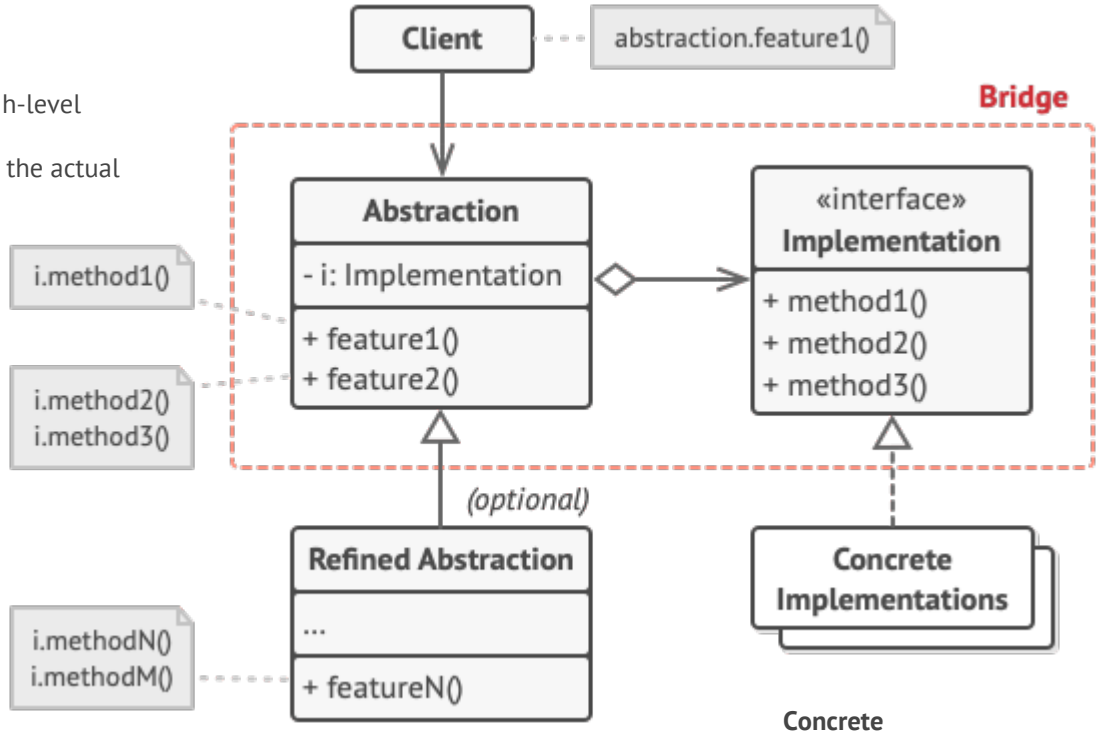
- Problem – How to decouple interface from implementation allowing substitution at runtime?
- Solution
  - Define the interface required by client in the Abstraction class.
  - Define the methods required by Abstraction class in the abstract Implementor class.
  - Implementor object can be accessed by Abstraction object using an internal reference.
  - Abstraction and Implementor subclasses can be altered independently.
  - Methods of the abstraction can be written in terms of or using methods of the implementor class/interface (i.e. **low level** methods)
  - **High level** methods are written in terms of only the abstraction
- Advantage
  - Client need not be aware of implementation details. For example data may be stored using flat files, JDBC or XML.
  - Decoupling of interface and implementation as they can be altered independently.

Bridge Pattern: General Case

https://refactoring.guru/design-patterns/bridge

Usually, the **Client** is only interested in working with the abstraction. However, it's the client's job to link the abstraction object with one of the implementation objects.

The **Abstraction** provides high-level control logic. It relies on the implementation object to do the actual low-level work.



The **Implementation** declares the interface that's common for all concrete implementations. An abstraction can only communicate with an implementation object via methods that are declared here.

The abstraction may list the same methods as the implementation, but usually the abstraction declares some complex behaviors that rely on a wide variety of primitive operations declared by the implementation.

Refined Abstractions provide variants of control logic. Like their parent, they work with different implementations via the general implementation interface.

Concrete Implementations contain platform-specific code.

Consequences

- The implementation of an abstraction can be configured at run-time. It's even possible for an object to change it's implementation at run-time.
- Abstraction and Implementor eliminate compile-time dependencies on the implementation. Changing the Implementation classes does not require recompiling abstraction and its clients.
  - this could be handled by a factory (and with the help of config data/files if necessary)
  - the choice can be made either by the client (increases flexibility) or by the Abstraction (decreases coupling with the client)
  - having the abstraction choose the implementation allows it to change it response to changing data structures
- The relationship between the Abstraction and the Implementor is the bridge
- Implementation details can be hidden from the clients.
- **NOTE**: The bridge patterns can be used to any two independent hierarchies, it doesn't strictly have to follow implementor semantics

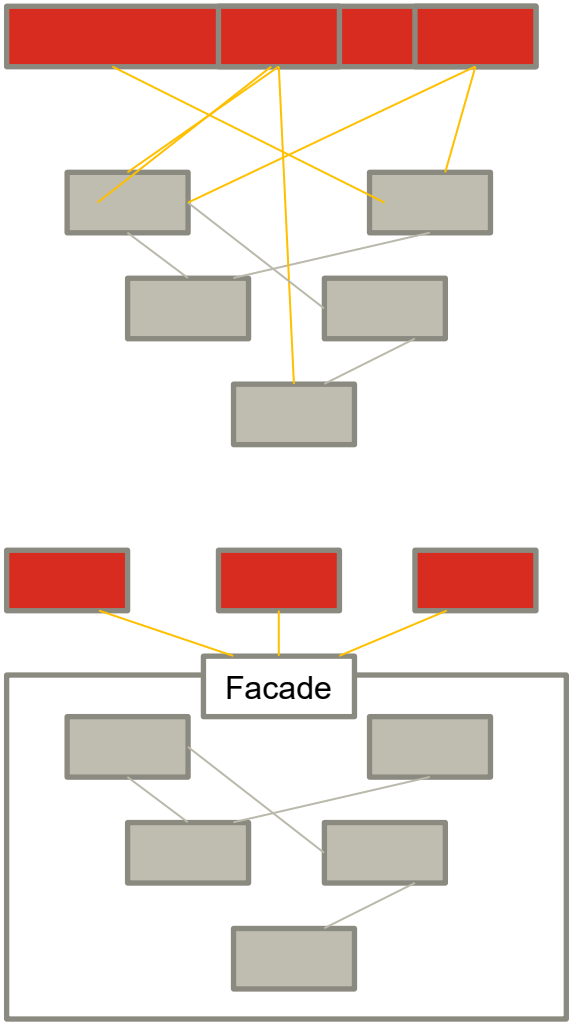
The Facade pattern

Intent

- Provide a unified interface to a set of interfaces in a subsystem

Motivation

- Structuring a system into subsystems helps reduce complexity and increases cohesion
- Facade aims to minimize the communication and dependencies between clients and sub systems
- useful for distributed and federated systems e.g. using web/cloud services
- useful for providing modern interface over legacy code
- useful for combining different coding languages



Applicability

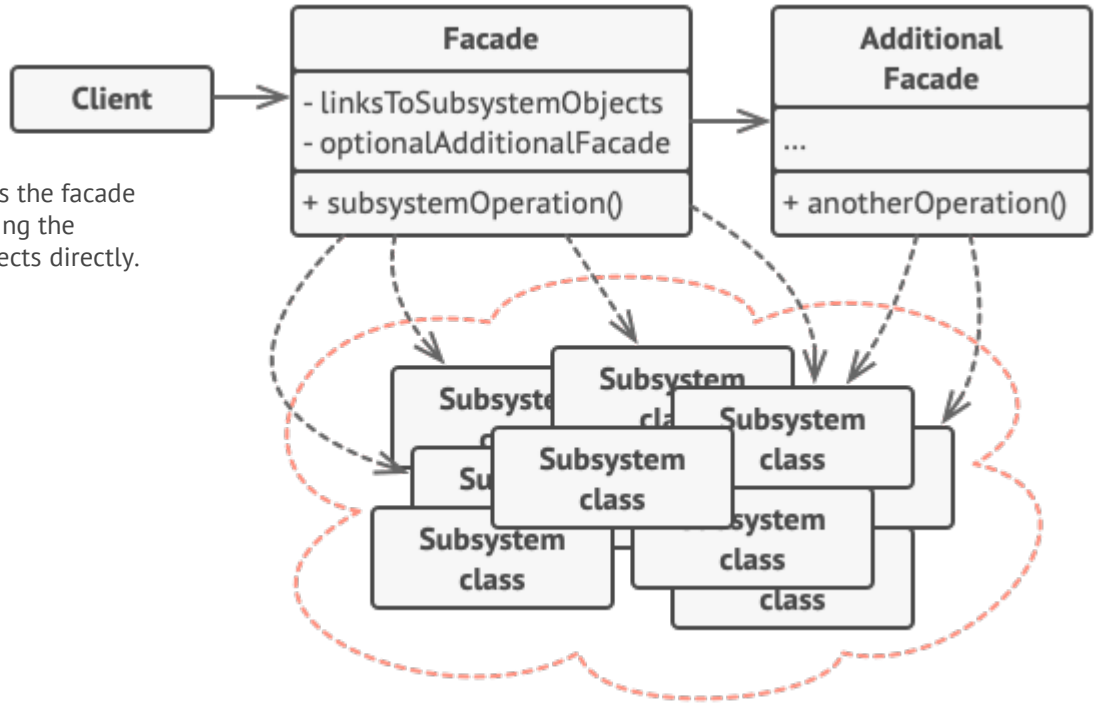
- Subsystems get more complex as they evolve.
  - a facade can provide a simple default view of the system that is simpler for most clients
  - if clients have specific needs they can still access the subsystems directly
  - related to the Interface Segregation Principle (GRASP)
- The subsystem can be decoupled from clients thereby making them more portable.
- Allows a additional layer to be created between clients and subsystems e.g. using proxy pattern

Structure

https://refactoring.guru/design-patterns/facade

The **Facade** provides convenient access to a particular part of the subsystem's functionality. It knows where to direct the client's request and how to operate all the moving parts.

An **Additional Facade** class can be created to prevent polluting a single facade with unrelated features that might make it yet another complex structure. Additional facades can be used by both clients and other facades.



The **Client** uses the facade instead of calling the subsystem objects directly.

The **Complex Subsystem** consists of dozens of various objects. To make them all do something meaningful, you have to dive deep into the subsystem's implementation details, such as initializing objects in the correct order and supplying them with data in the proper format.

Subsystem classes aren't aware of the facade's existence. They operate within the system and work with each other directly.

Example

- To provide a real source code example to the Facade we would need a complex real system
- In contrast to previous examples the example on the following slide is a conceptual example that shows how a computer could be modelled as subsystems
- A facade interface is provided to simplify interaction with the various subsystems
- public APIs are often good examples of facades, especially public APIs to web/cloud services

Facade: Consequences

- Facades often provide an interface thereby minimising coupling with the client
- Shield clients from subsystem implementation details – reducing the number of objects clients deal with.
- This allows components of the subsystems to be changed without affecting the clients.
- Facade does not prevent clients using the subsystem classes if they need to.
- DISADVANTAGE: The façade may become large and uncohesive for large subsystems and require partitioning
- Need to make the decision about the granularity of methods in the Façade and how many of the subsystem methods to expose (and at what level of implementation coupling)
- Can be useful for remote objects or web services where cohesion is sacrificed for efficiency
  - i.e. it would be inefficient to have multiple calls to separate services across the network