Midterm Review

Kevin Camellini
SOEN 344
Winter 2019
Concordia University

Table of Contents

SOFTWARE ARCHITECTURE INTRO

Software Architect?

- Hierarchically somewhere between Dev and Team Lead
- Responsible for strategic decisions, e.g., which technology or architectural style to use

Components, connectors, constraints and rationale

- **Components**
  - Provide **application-specific** functionality
  - A unit of decomposition of a system
    - module package, web service
- **Connectors**
  - Provide **application independent** interaction mechanisms
  - Architectural element that models interactions among components and the rules that govern those interactions
  - Simple interactions
    - Procedure calls, Shared variable access
  - Complex interactions
    - Client-server protocols, Database access protocols, Asynchronous event multi-cast
  - Can be a converter, facilitator amongst components
- **Constraints**
  - requirements are met
  - required functionality is achieved
  - no functionality is duplicated
  - required performance is achieved
  - modularity is realized
- **Rationale**
  - Rationale document = answers why?
    - **Decomposition** into *components*
    - **Connections** between *components*
    - **Constraints** upon *components* and *connections*

4 + 1 View

- Serves to **derive** and **document** architecture

```
Logical   ➡️   Development

  ⬇️   Scenario   ⬇️

Process   ➡️   Physical
```

1. **Logical View**
    - Functional Requirements
2. **Development View**
    - Organization of modules, components and elements
    - Organization in hierarchical layers
3. **Process View**
    - Dynamic aspects of the **run-time** processes
        - Process creation
        - Synchronization
        - Concurrency
4. **Physical View**
    - Non-Functional Requirements
    - Mapping of software onto existing/ available **physical hardware**
5. **+ Scenarios**
    - User stories that run through all 4 views.

ARCHITECTURAL STYLES (7)

An architectural **style** is a common architectural **design**. It defines for ___ usage in concrete software systems:

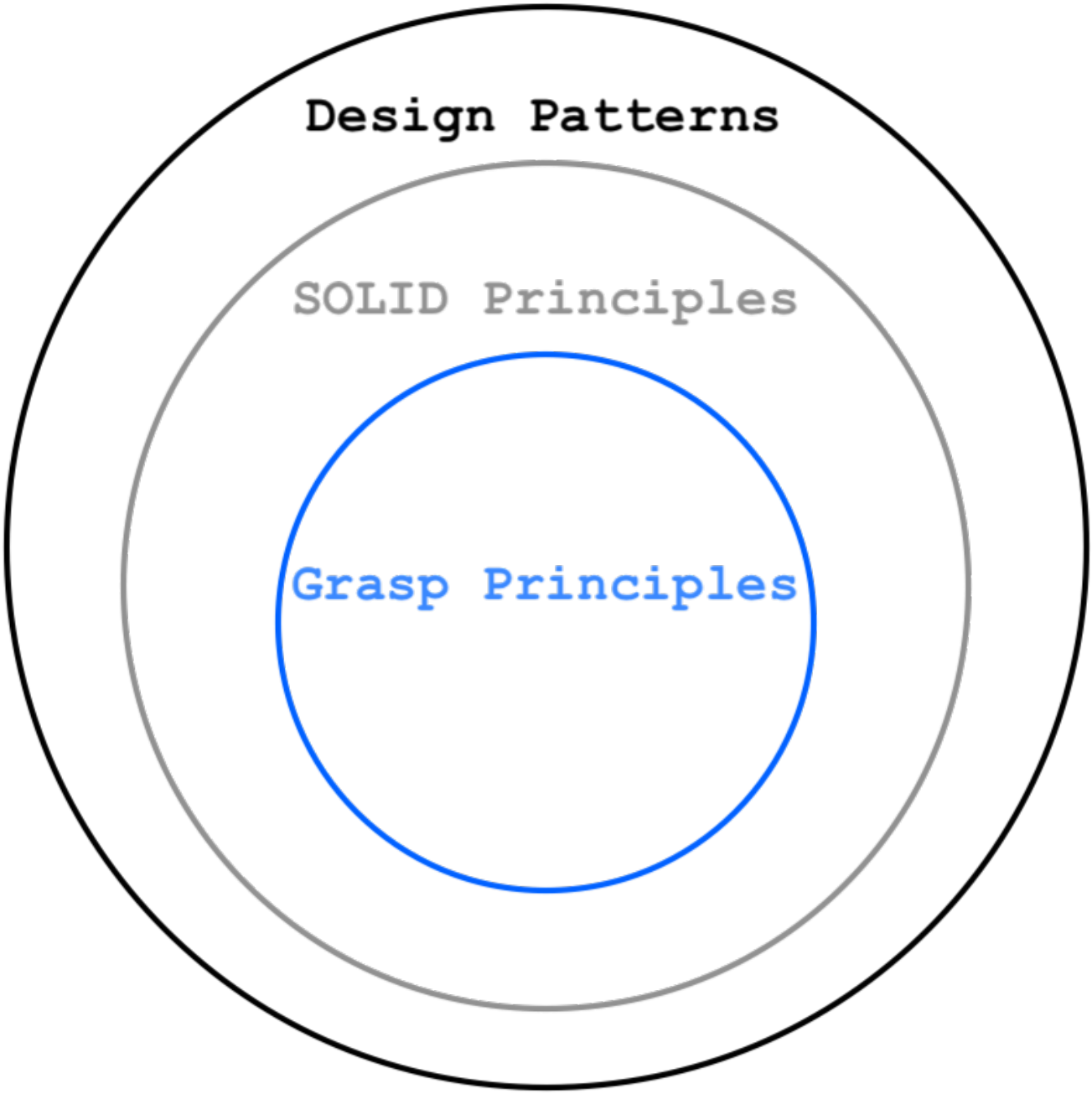
- components
- connectors
- constraints

An architectural **pattern** is a way of **solving** a common architectural **problem**.

1. Layers
    - Hierarchically organized system Layers are components
    - Layer interfaces and protocols are **connectors**
    - Application, presentation, session, transport, network, data link and physical layer
2. Client/Server
    - Servers are active all the time and passively wait for requests
    - Clients are active sporadically and trigger Servers
    - Servers are (usually) few and fat, while clients are numerous and thin
    - + Supports many users at the same time
    - - Servers are a single point of failure
3. MVC
    - Models hold data
    - Views present data
    - Controllers implement business logic
    - + Separation of concerns
    - + Supports TDD well
    - - Usually every change touches M,V and C.
4. P2P
    - Components are both **server** and **client** at the same time
    - **Everybody** is (in principle) communicating with **everybody**
    - Two styles:
        - Pure P2P
        - Hybrid P2P (central servers or super-peers)
    - + Scalability
        - Each new participant also adds new resources
        - No natural bounds to system scale.
    - + Usually few or *no single point* of **failure** or **control**
    - - message flooding is a problem
    - - Protocols tend to get complicated
5. Pipe and Filter
    - **Filters** are the components that *read an input data stream* and *transform it into an output data stream*
    - **Pipes** are the ***connectors*** that provide the output of a filter as input to another filter
    - + Simple; no complex component interactions

    - + High maintainability and reusability of individual components
    - - Filters require a common data format
    - - Redundancy in parsing/unparsing
    - - Process overhead
6. Event-Driven Architecture
    - Publisher/Subscriber (listeners)
    - **Components** are event emitters (publishers, agents) or consumers (subscribers, sinks)
    - An **event dispatcher** distributes events to Subscribers
    - fundamentally **asynchronous**
    - + Components are loosely coupled
    - + Components can be easily exchanged
    - - No guarantees regarding execution or order of event processing (asynchronous)
7. Blackboard
    - Moodle is a bloackboard
    - Central data management **component**
    - Independent **components** for computation
    - Blackboard can serve as a factory for tasks and activate workers
    - Like a **database** but doesn't only store data
    - + High changeability and maintainability
    - + support for fault- tolerance, robustness and redundancy because of loose coupling of workers
    - - Hard to test
    - - Difficult to establish good control strategy
    - - Low efficiency

OOP DESIGN PRINCIPLES (14)



The (16) **design patterns** are blueprints that *help adhere* to the (14) **design principles** (GRASP & SOLID).

Software design principles are general rules/best practices on how to organize your software to support:

- **Reusability**
- **Maintainability**
- **Stability**

GRASP Principles(9)
General **R**esponsibility **A**ssignment **S**oftware **P**atterns

1. Information Expert
2. Creator
3. High Cohesion
4. Low Coupling
5. Controller
6. Polymorphism
7. Pure Fabrication
8. Protected Variations
9. Indirection

SOLID Principles (5)

1. **S**ingle Responsibility Principle
2. **O**pen-Closed Principle
3. **L**iskov Substitution Principle
4. **I**nterface Segregation Principle
5. **D**ependency Inversion Principle

GRASP (9)

Responsibilities of an Object include two types:

- Knowing
  - Knowing about private encapsulated data (know thyself, presume not God to scan)
  - Knowing about related objects
  - Knowing about things it can derive or calculate
- Doing
  - Doing something itself, such as creating an object or doing a calculation
  - Initiating action in other objects
  - Controlling and coordinating activities in other objects

GRASP provides a representation of nine basic principles that form a foundation for designing object-oriented systems. **Methods** fulfill responsibilities.

1. Creator
   - Problem: Who creates an object A?
   - Solution: Assign class B the responsibility to create an instance of class A if one of these is true:
     - B “contains” or completely aggregates A
     - B records A
     - B closely uses A
     - B has the initializing data for A
     - *Then B should be the CREATOR*
2. Information Expert
   - Expert asks us to find the object that has most of the information required for the responsibility and assign

responsibility there.
    - Expert is in knowing and doing.
    - **Expert Supports Low Coupling**
    - **Problem**: What is a basic principle by which to assign responsibilities to an object
    - **Solution**: Assign a responsibility to the class that has the information needed to respond to it.
3. Low Coupling
    - Coupling is a measure of **how strongly one object** is *connected to*, *has knowledge of*, or *depends upon* **other objects**.
    - An object A that calls on the operations of object B has coupling to B's services. When object B changes, object A may be affected.
    - **Problem**: How to reduce the impact of change?
    - **Solution**: Assign responsibilities so that (unnecessary) coupling remains low. Use this principle to evaluate alternatives.
4. High Cohesion
    - Cohesion measures **how functionally related** the operations of a software element are. It also measures how much work an object is doing.
    - Note: Low Cohesion and High Coupling often go together
    - **Problem**: How to keep objects focused, understandable, and manageable, and, as a side effect, support Low Coupling
    - **Solution**: Assign responsibilities so that cohesion remains high. Use this criteria to evaluate alternatives.
5. Controller
    - **Problem**: What first object beyond the UI layer receives and coordinates a System Operation?
    - **Solution**: Assign the responsibility to an object representing one of these choices:
        - Represents the overall system – a **root** object
        - Represents a use case scenario within which the system operation occurs.
6. Polymorphism
    - **Problem**: How to handle alternatives based on type.Pluggable software components -- how can you replace one server component with another without affecting the client?
    - **Solution**: When **related alternatives** or behaviours vary by type or **class**, assign responsibility for the behaviour – using polymorphic operations – to the types for which the behaviour varies. In this context, polymorphism means giving the same name to similar or related services
7. Pure Fabrication
    - **Problem**: What object should have responsibility when you do not want to violate *High Cohesion* and *Low Coupling*, or other goals, but solutions offered by Expert (for example) are not appropriate?
        - Sometimes assigning responsibilities only to domain layer software classes leads to problems like low cohesion, high coupling, or low reuse potential.
    - **Solution**: Assign a highly cohesive set of responsibilities to an artificial or convenience class that does not represent a domain concept.
8. Indirection (intermediary)
    - **Problem**: Where do we assign responsibility if we want to **avoid direct coupling** between two or more objects?
    - **Solution**:Assign responsibility to an intermediate object to mediate between the other components.
9. Protected Variations (predicted instability)
    - **Problem**: How do we design objects and systems so that instability in them does not have undesirable effects on other elements?

- **Solution**: Identify points of predicted instability (variation) and assign responsibilities to create a stable interface around them.

SOLID (5)

Don't be afraid to break shit up into many classes

1. **S**ingle Responsibility
   - Every class should have a single responsibility. There should never be more than one reason for a class to change.
2. **O**pen-Closed (easily extendable)
   - A class should be **easily extendable** without modifying the class itself.
   - Use inheritance to to extend functionalities.
   - **Strategy Pattern** is a great demonstration of the open/closed principle.
3. **L**iskov Substitution (substitutable derived subclasses)
   - Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.
   - Every sub/derived class should be substitutable for their base/parent class.
4. **I**nterface Segregation
   - Clients should not be forced to depend on interfaces they do not use.
5. **D**ependency Inversion (dependencies depend on abstractions)
   - High-level modules should not depend on low-level modules. Both should depend on **abstractions**.
   - Abstractions should not depend on details. Details should depend on abstractions.

DESIGN PATTERNS (16)

Creational Patterns (5/21)

Often, designs start out using **Factory Method** (less complicated, more customizable, subclasses proliferate) and evolve toward **Abstract Factory, Prototype**, or **Builder** (more flexible, more complex) as the designer discovers where more flexibility is needed.

Abstract Factory, Builder, and Prototype define a factory object that's responsible for knowing and creating the class of product objects, and make it a parameter of the system.

- **Abstract Factory** has the factory object producing objects of several classes.
- **Builder** has the factory object building a complex product incrementally using a correspondingly complex protocol.
- **Prototype** has the factory object (aka prototype) building a product by copying a prototype object.

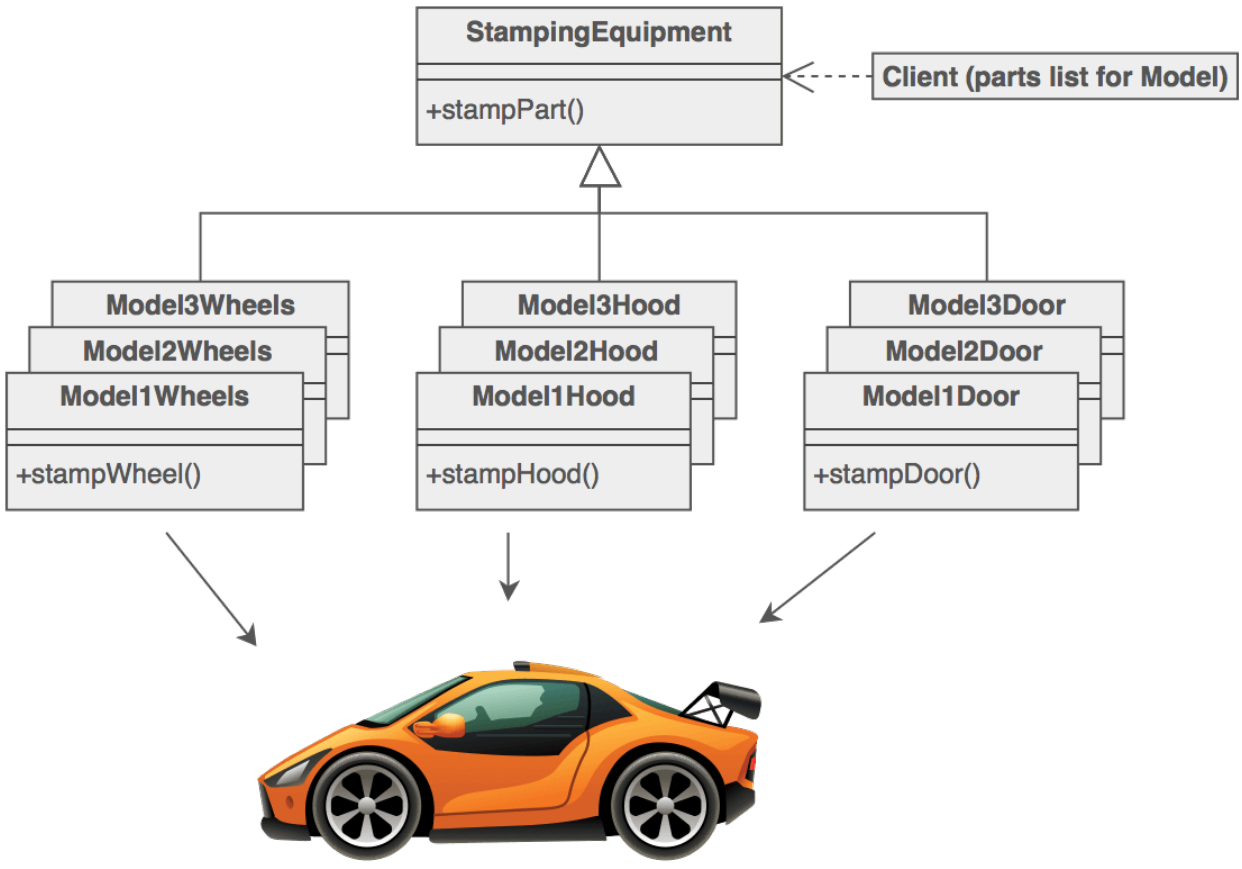
Builder focuses on constructing a *complex object step by step*. **Abstract Factory** emphasizes a *family of product objects* (either simple or complex). **Builder** returns the product *as a final step*, but as far as the **Abstract Factory** is concerned, the product gets returned *immediately*.

1. Factory
    - **Intent**:
        - Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class **defer instantiation to subclasses**.
        - The `new()` operator considered harmful.
    - **Problem**: A framework needs to standardize the architectural model for a range of applications, but allow for individual applications to define their own domain objects and provide for their instantiation.
    - **Example**:

      ```
      Color.make_RGB_color(float red, float green, float blue)
      Color.make_HSB_color(float hue, float saturation, float brightness)
      ```

    - **Notes**:
        - **Abstract Factory** classes are often implemented with **Factory Methods**, but they can be implemented using **Prototype**.
        - **Factory Methods** are usually called within **Template Methods**.
        - **Factory Method**: creation through *inheritance*. **Prototype**: creation through *delegation*.
2. Abstract Factory
    - **Intent**:
        - Abstract Factory patterns work around a super- factory which creates other factories.
        - The `new()` operator considered harmful.
        - An interface is responsible for creating a factory of related objects **without explicitly specifying their classes**.

- **Design Goals**: Correctness and Reusability
- **Problem**: Provide an interface for creating families of related objects, without specifying concrete classes.
- **Example**:



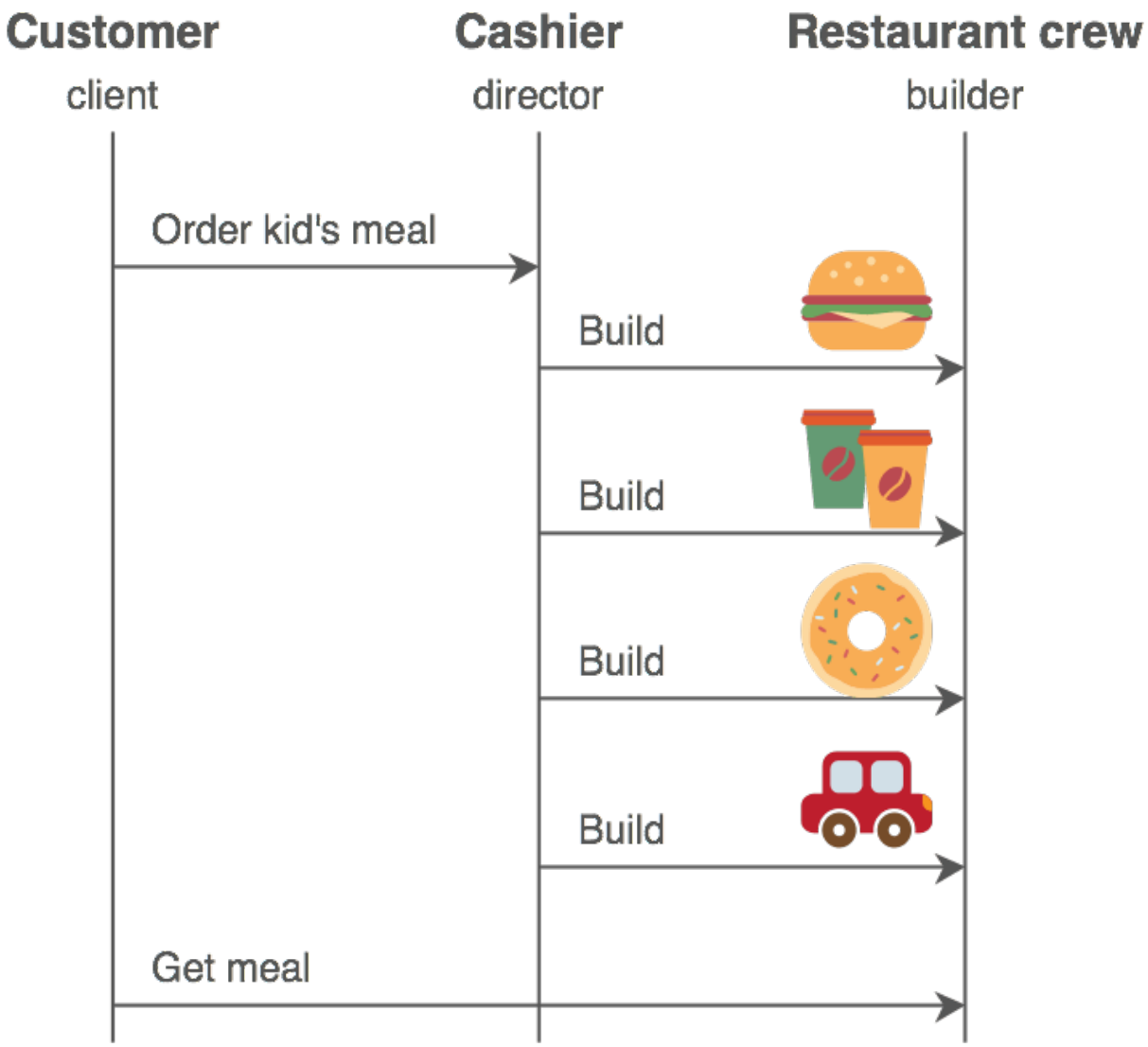
Sheet metal stamping equipment used in the manufacture of automobiles. The stamping equipment is an Abstract Factory which creates auto body parts

- **Notes**:
  - **Abstract Factory** classes are often implemented with **Factory Methods**, but they can also be implemented using **Prototype**.
  - **Abstract Factory** can be used as an alternative to **Facade** to hide platform-specific classes.
  - Sometimes creational patterns are competitors: there are cases when either **Prototype** or **Abstract Factory** could be used profitably.

3. Builder

- **Intent**: Separate the construction of a *complex object* from its representation so that the same construction process can create different representations. Parse a complex representation, create one of several targets.
- **Problem**: An application needs to create the elements of a complex aggregate. The specification for the aggregate exists on secondary storage and one of many representations needs to be built in primary storage.

- **Example**: A ship yard.



- **Notes**:

- **Builder** often builds a Composite.

4. Prototype (clone)
   - **Intent**:
     - Prototype Pattern is about cloning an existing object instead of creating a new one from scratch and then customize it as per the requirement.
     - The `new()` operator considered harmful.
     - Creational pattern that focuses on the performance during object creation.
   - **Problem**: Performance
   - **Example**: `clone()`
   - **Notes**:
     - **FactoryMethod**: creation through *inheritance*. **Prototype**: creation through *delegation*.
     - Developers tend to use **Prototype** with **Abstract Factory** to improve the *performance* and reduce the

overhead of creating different objects.
        - Designs that make heavy use of the **Composite** and **Decorator** patterns often can benefit from Prototype as well.
5. Singleton 🤴
    - Ensure that a class has only one instance, while providing global access.
    - Singletons are intended to be used when a class must have **exactly one instance**, no more, no less
    - Violates Single Responsibility Principle
    - Maybe difficult to unit test as many test frameworks rely on inheritance when producing mocked objects
    - Solves 2 problems:
        - Ensures that a class has just a single instance
        - Provide a global access point to that instance, while protecting that instance from being overwritten by other code.
    - **Notes**:
        - **Facade** objects are often Singletons because only one Facade object is sufficient.
        - **State** objects are often Singletons.
        - **Flyweight** would resemble Singleton if you somehow managed to reduce all shared states of the objects to just one flyweight object. But there are two fundamental differences between these patterns:
            - There should be only one Singleton instance, whereas a **Flyweight** class can have multiple instances with different intrinsic states.
            - The Singleton object can be mutable. **Flyweight** objects are immutable.
            - **Abstract Factories**, **Builders** and **Prototypes** can all be implemented as Singletons.

Behavioral Patterns (9/21)

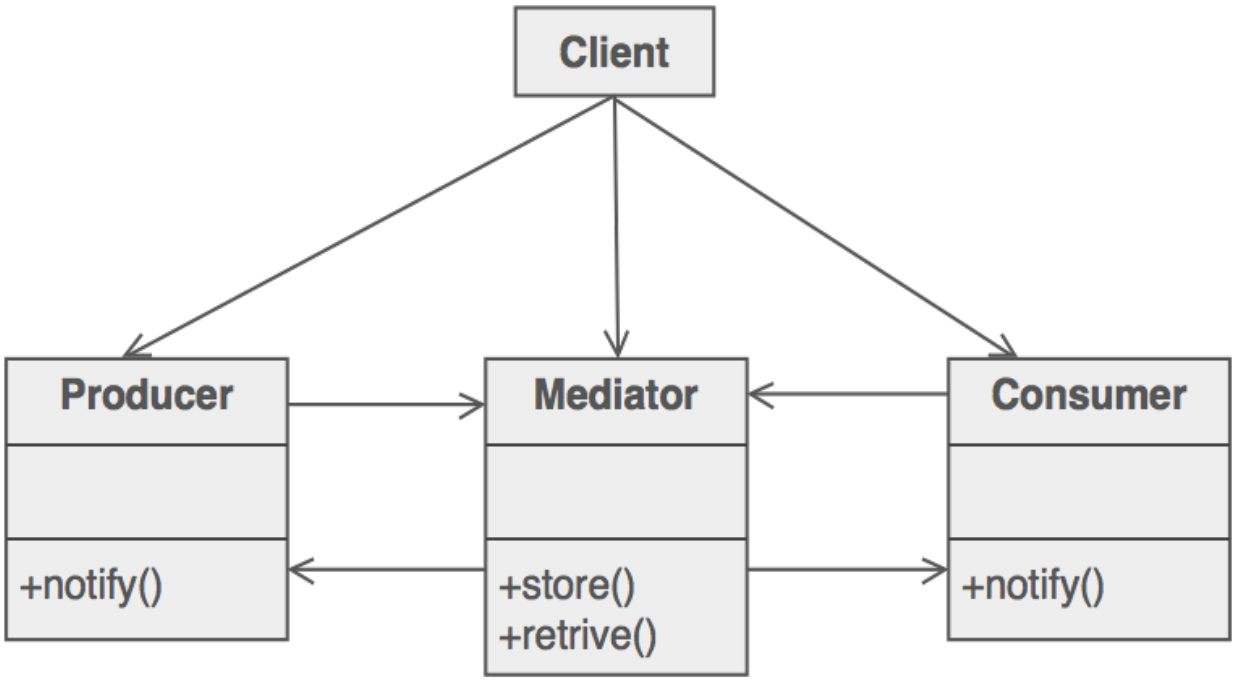
Chain of Responsibility, **Command**, **Mediator**, and **Observer**, address how you can decouple ***senders*** and ***receivers***, but with different trade-offs. - **Chain of Responsibility** passes a sender request along a chain of potential receivers. - **Command** normally specifies a sender-receiver connection with a subclass. - **Mediator** has senders and receivers reference each other indirectly. - **Observer** defines a very decoupled interface that allows for multiple receivers to be configured at run-time.

1. Mediator 🛩🛫🛬

    - Reduce chaotic dependencies between objects and forces them to collaborate only via mediator object

    - Promotes loose coupling by keeping objects from referring to each other and lets you vary their interaction independently

    - **Notes**:
        - **Mediator** and **Observer** are competing patterns. The difference between them is that Observer distributes communication by introducing "observer" and "subject" objects, whereas a Mediator object encapsulates the communication between other objects. We've found it easier to make reusable Observers and Subjects than to make reusable Mediators.



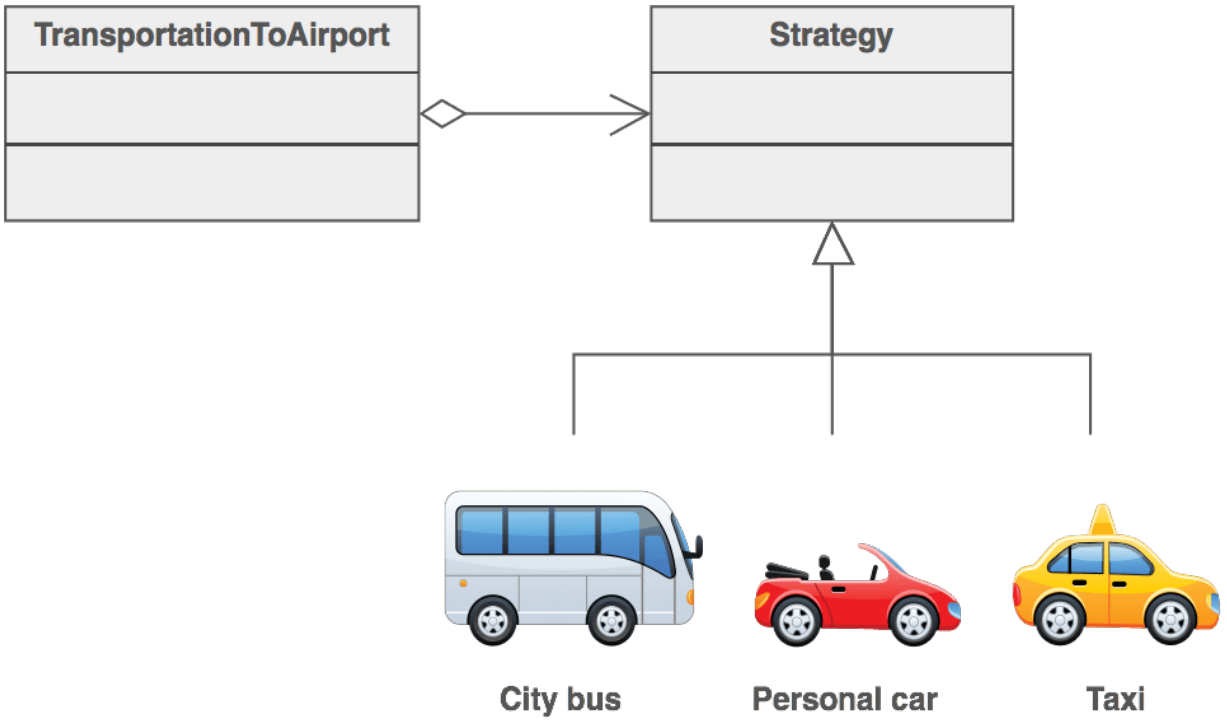
2. Strategy (🚗🚕🚌 for 🛫)

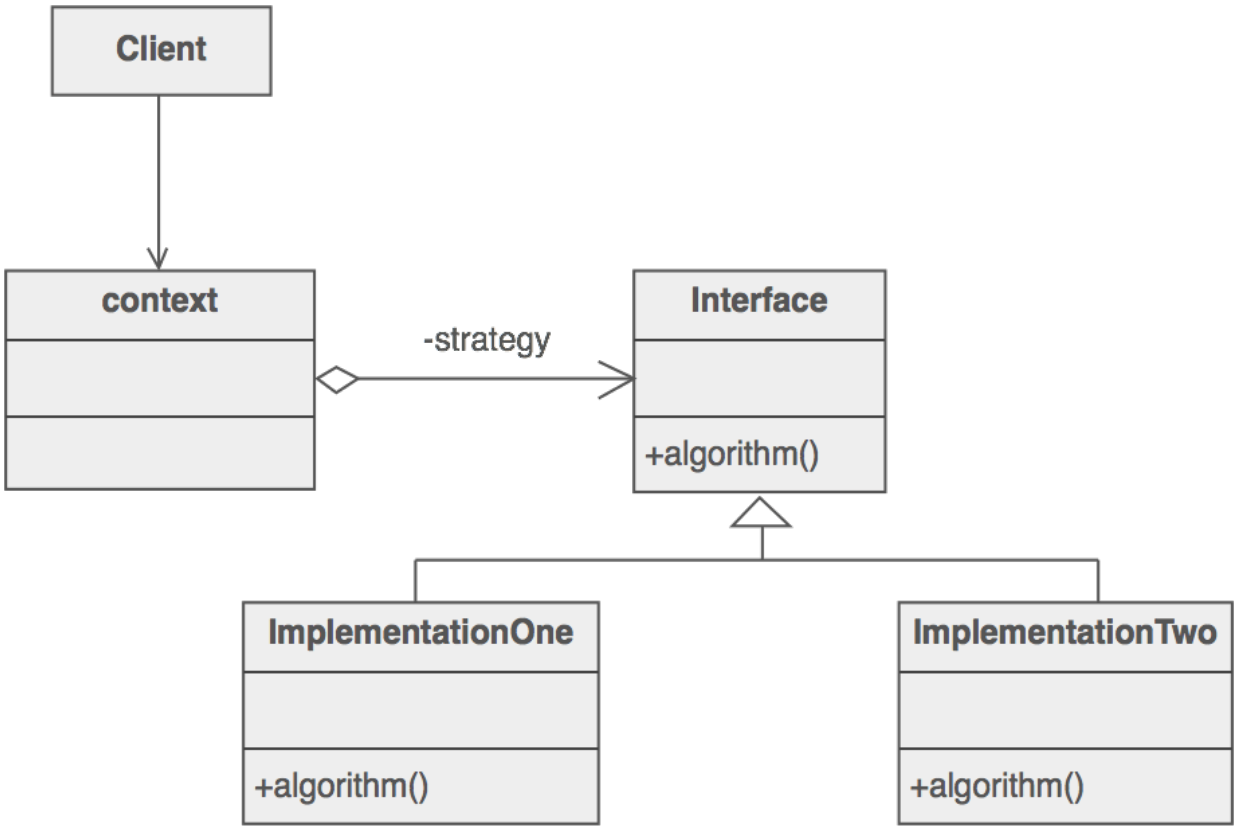
    - **Intent**: Define a family of algorithms, encapsulate each one, and make them interchangeable. Essentially, the strategy pattern allows us to **change the behaviour of an algorithm at runtime.**

    - **Problem**: Maximize **cohesion** and minimize **coupling**.

    - **Notes**:
        - **Strategy** lets you change the *guts* of an object. **Decorator** lets you change the *skin*.

- **Strategy** is like **Template** Method except in its *granularity*.
- **State** is like **Strategy** except in its *intent*.
- **Strategy** objects often make good **Flyweights**.

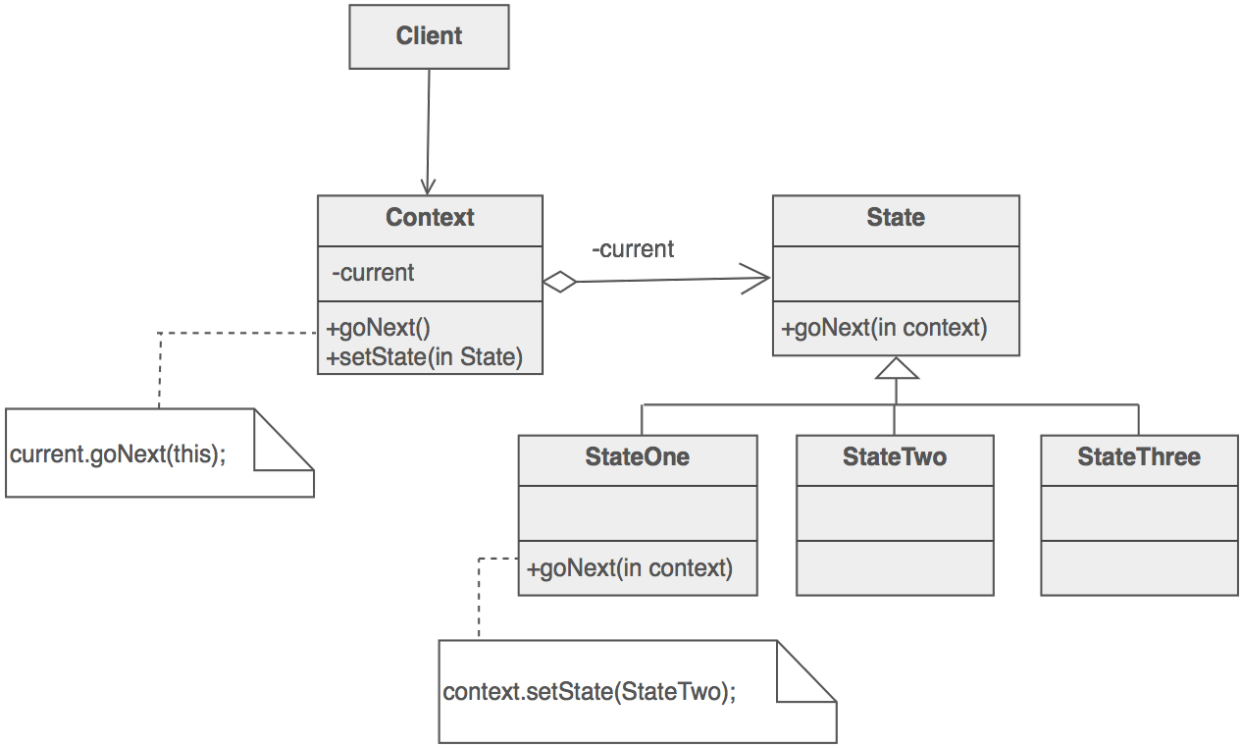




3. State (vending machine)

    - Allow an object to alter its behaviour when its internal state changes state. The object will appear to change its class.

    - In the **state** pattern, the particular states may be aware of each other and initiate transitions from the one state to another, whereas **strategies** almost never know about each other.

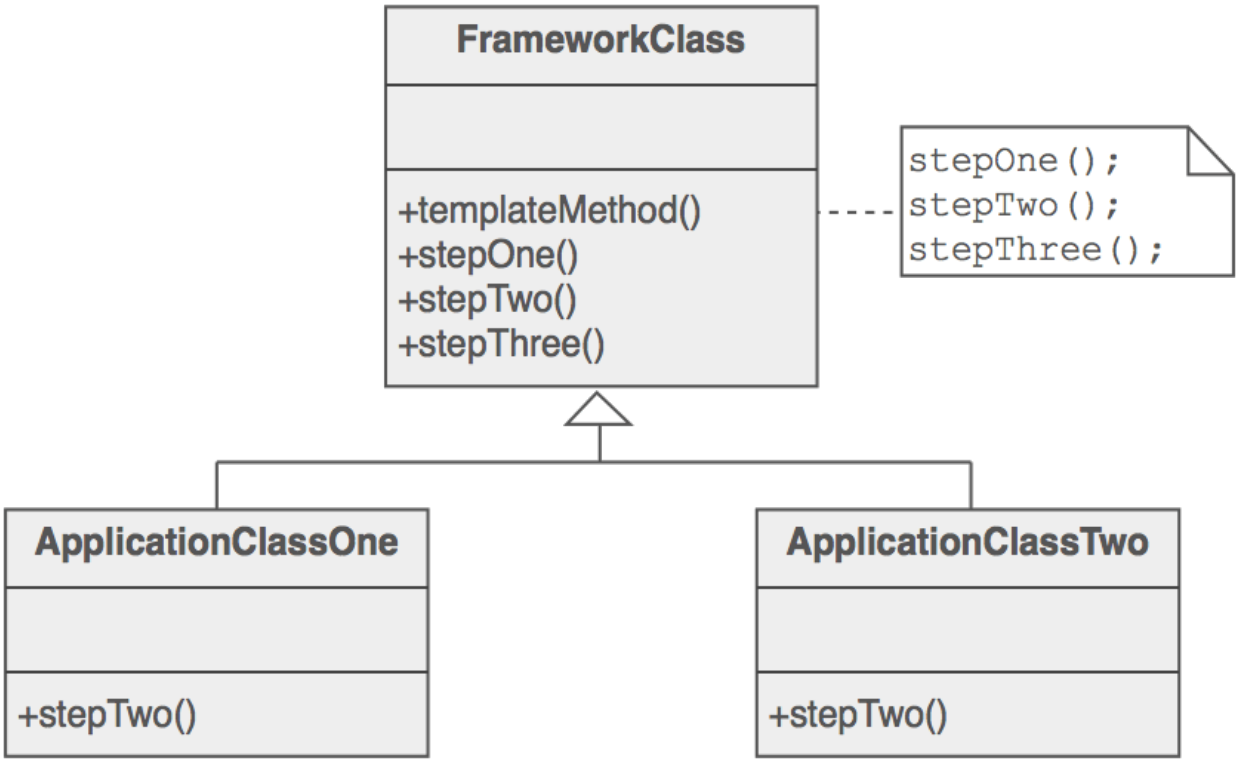


4. Template Method 🏠🏗️👷

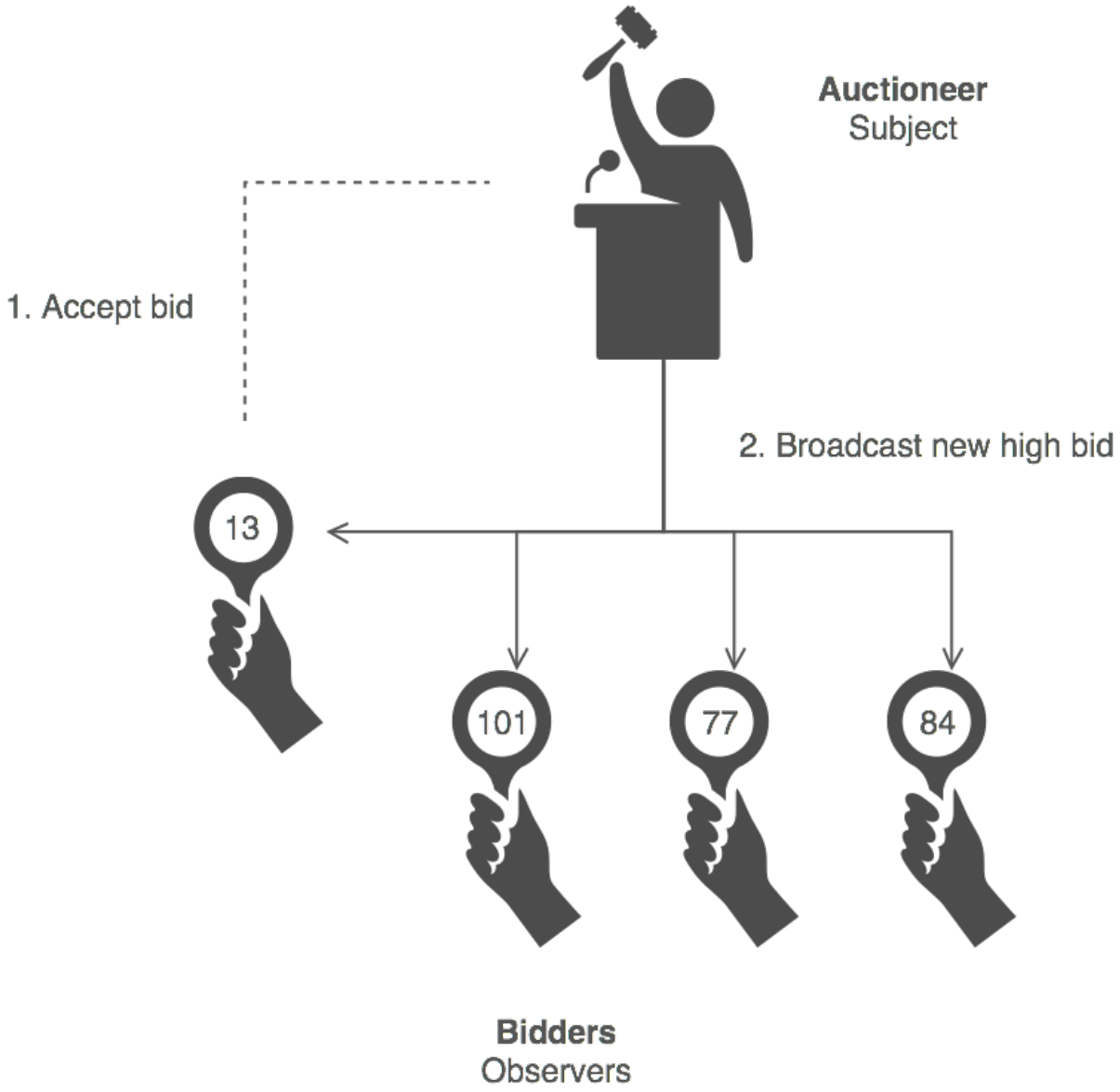
    - Defines the skeleton of an algorithm in the superclass but lets subclasses override specific steps of the algorithm w/o changing its structure.

    - Break down an algorithm into a series of steps, turn these steps into methods and put a series of calls to these methods inside a single template method.

    - **Notes**:
        - Clients may be limited by the provided skeleton
        - Might violate Liskov Substitution Principle by suppressing a default step in implementation via subclassing.
        - More steps to Template pattern = harder to maintain.
        - **Template Method** uses ***inheritance*** to vary part of an algorithm. **Strategy** uses ***delegation*** to vary the entire algorithm.
        - **Strategy** modifies the logic of individual ***objects***. **Template Method** modifies the logic of an ***entire class***.
        - **Factory Method** is a *specialization* of the **Template Method**.

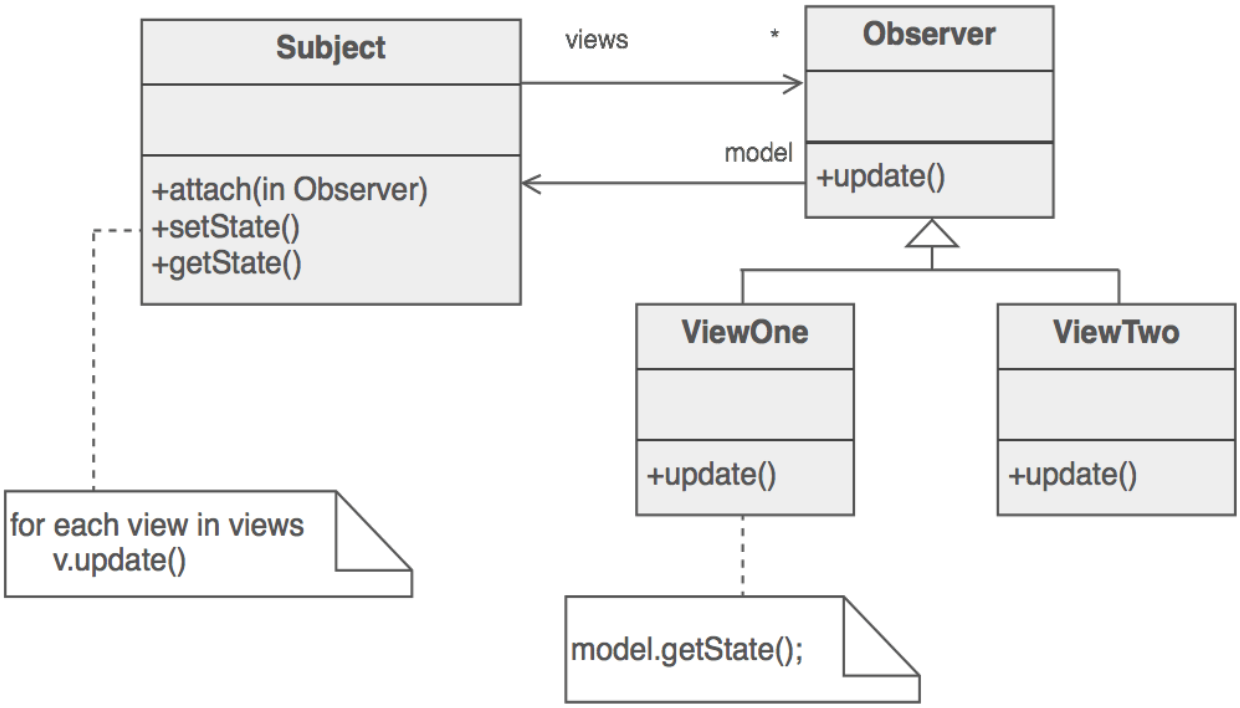


5. Observer 👀
    - **Intent**:
        - Observer pattern is used when there is **one-to-many** relationship between objects.
        - When one object changes **state**, all its *dependents* are **notified** and updated automatically.
    - **Example**:

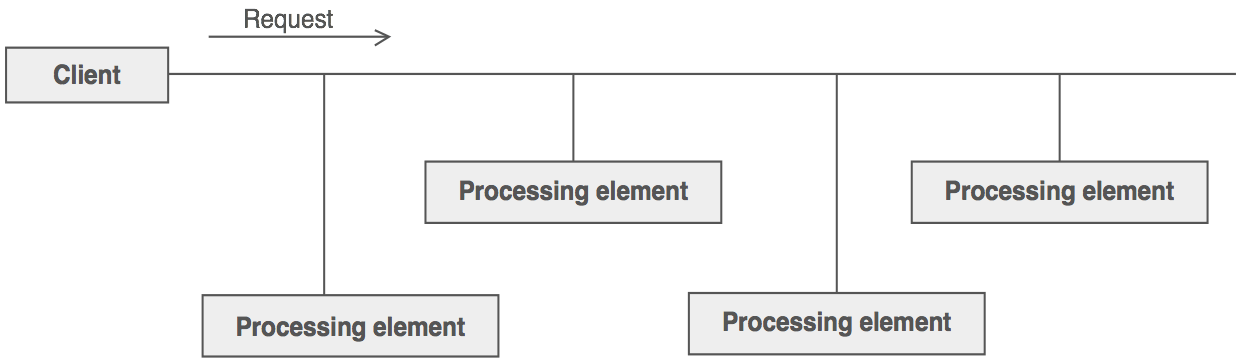


Subject is an object having methods to attach and detach observers. Concrete class **Subject** extends class **Observer**

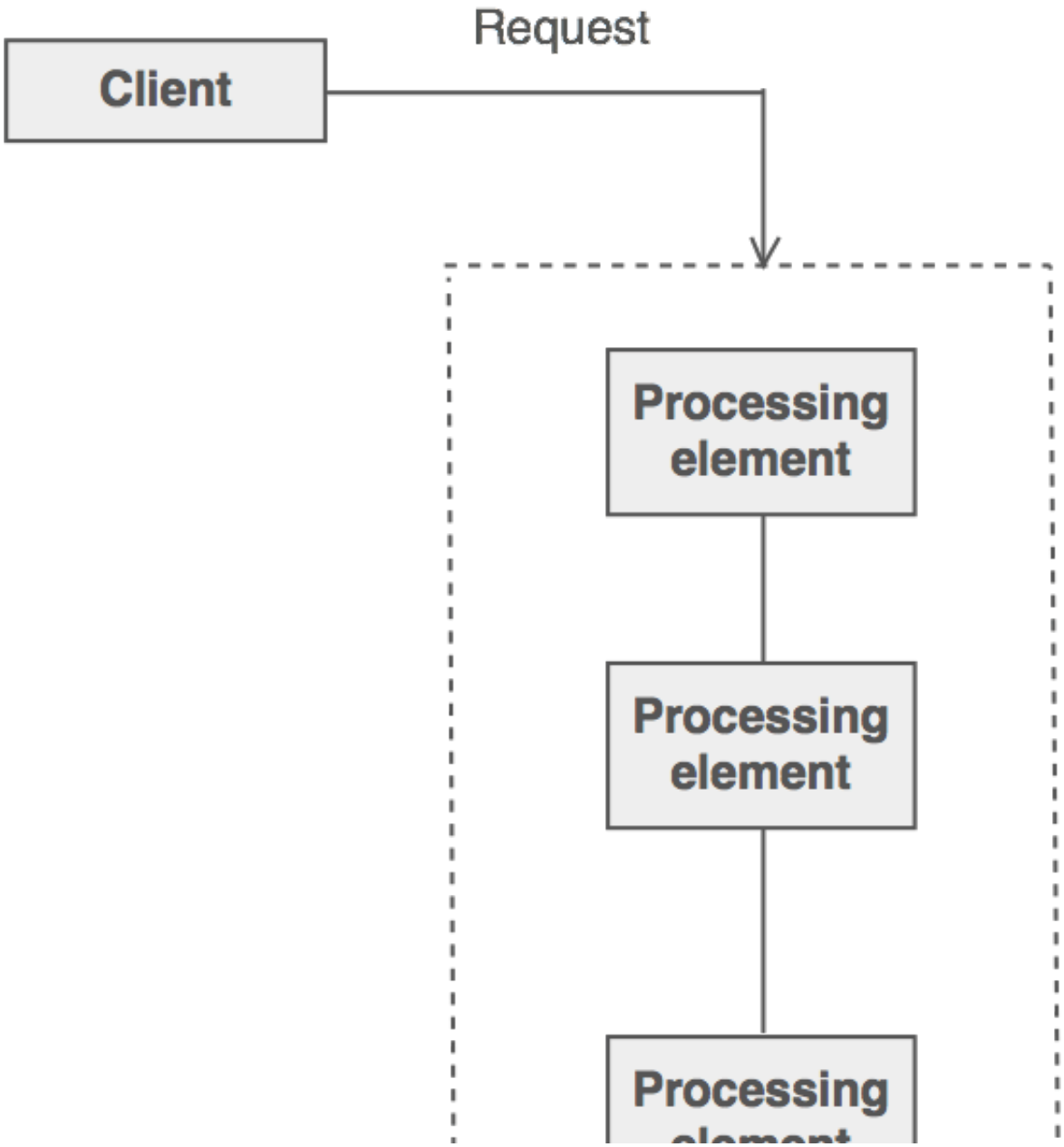
- **Notes**:
  - The *View* part of an MVC

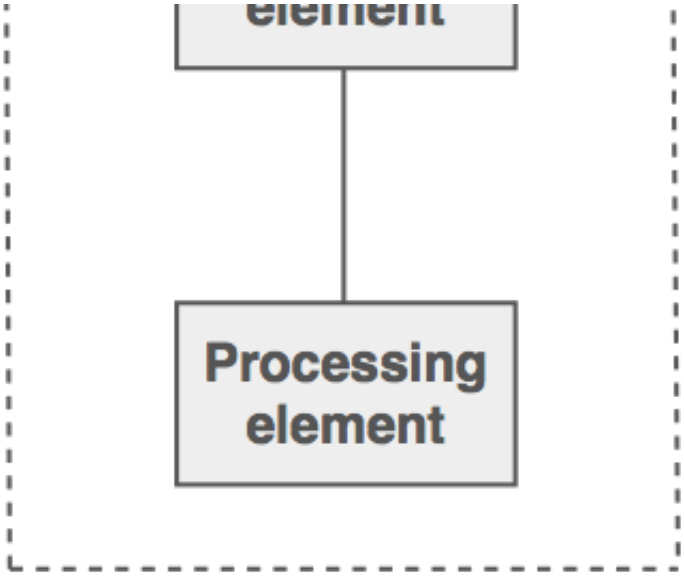


6. Command 📩
   - **Intent**:
     - Wrap it in a command object, send to invoker, invoker sends to appropriate object.
     - A request is *wrapped* under an object as **command** and passed to **invoker** object. **Invoker** object looks for the appropriate object which can handle this command and passes the command to the corresponding object which executes the command.
     - Specify a sender-receiver connection with a subclass.
   - **Problem**: Need to **issue requests to objects** without knowing anything about the operation being requested or the receiver of the request.
   - **Example**: `command.execute()`
   - **Notes**:
     - **Chain of Responsibility**, **Command**, **Mediator**, and **Observer**, address how you can decouple senders and receivers, but with different trade-offs. Command normally specifies a sender-receiver connection with a subclass.
     - **Chain of Responsibility** can use **Command** to represent requests as objects.
7. Chain of Responsibility 🔗
   - **Intent**:
     - 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.
     - Launch-and-leave requests with a single processing pipeline that contains many possible handlers.
     - Linked list of handlers (processing elements)
   - **Problem**: There is a potentially variable number of "handler" or "processing element" or "node" objects, and a stream of requests that must be handled. Need to efficiently process the requests without hard-wiring handler relationships and precedence, or request-to-handler mappings.
   - **Example**:



Chain of Responsibility simplifies object *interconnections*. Instead of *senders* and *receivers* maintaining references to all candidate receivers, each sender keeps a *single reference* to the head of the chain, and each receiver keeps a single reference to its immediate successor in the chain.





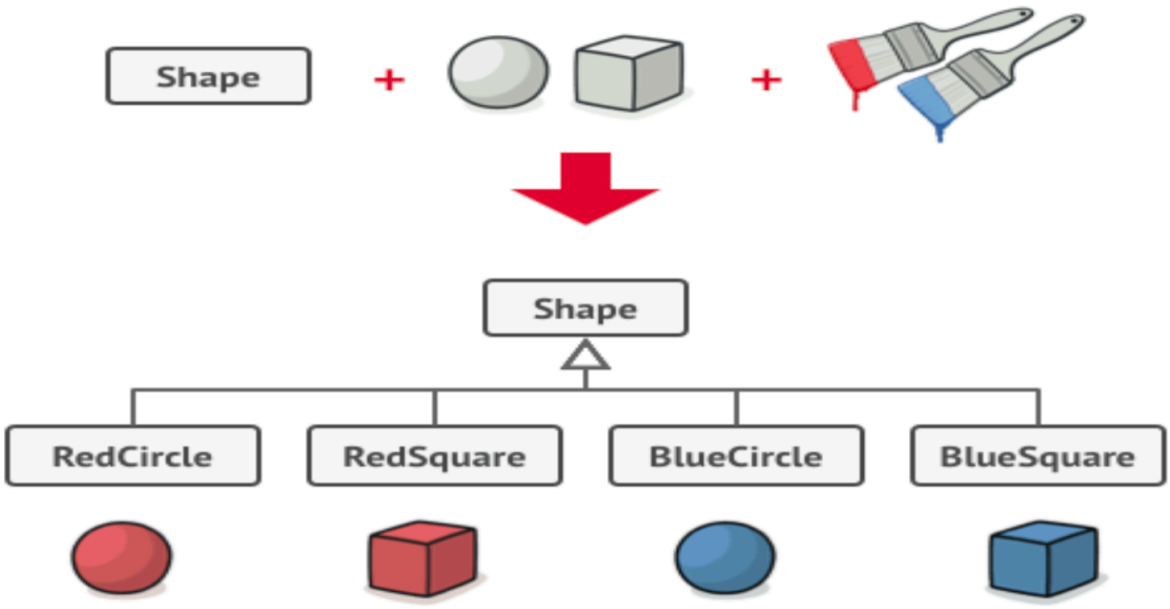
- **Notes**:
  - **Chain of Responsibility** is often applied in conjunction with **Composite**.
  - **Chain of Responsibility** can use **Command** to represent requests as objects.
  - **Chain of Responsibility**, **Command**, **Mediator** and **Observer**, address how you can decouple senders and receivers, but with different trade-offs. Chain of Responsibility passes a sender request along a chain of potential receivers
  - Make sure there exists a **safety net** to "catch" any requests which go unhandled.

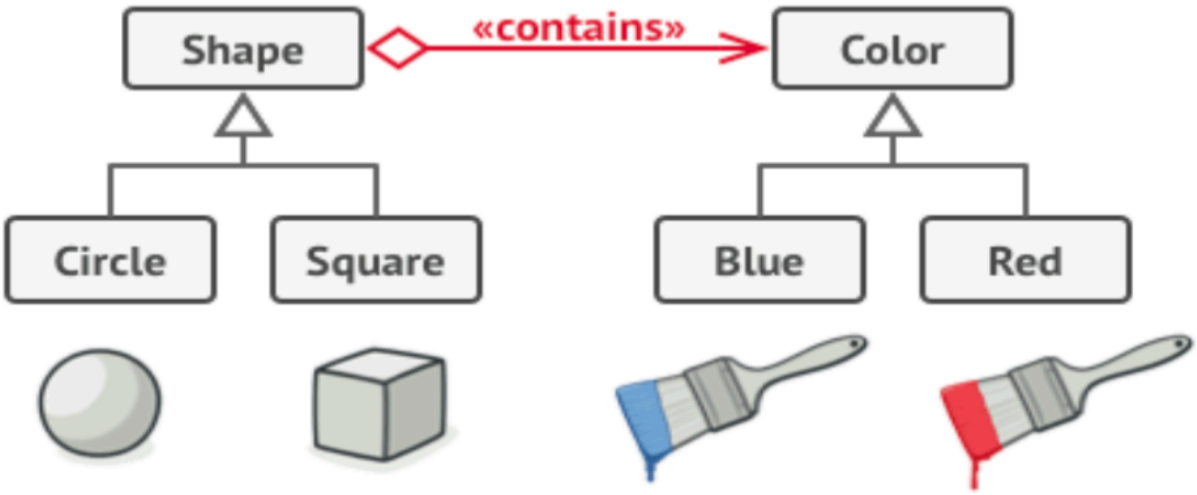
8. Memento 💣 ಽ(undo / rollback)
   - Save and restore the previous state of an object w/o revealing the details of its implementation.
     - Storing the copy of the object's state in a special object called memento
   - Content of memento aren't accessible to any other object except the one that produced it
     - Other obj must communicate with the memento using a limited interface only
   - **Notes**:
     - **Command** and **Memento** act as magic tokens to be passed around and invoked at a later time. In Command, the token represents a **request**; in Memento, it represents the internal **state** of an object at a particular time.
9. Iterator
   - **Intent**: Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
   - **Problem**: Need to **abstract** the **traversal** of wildly different **data structures** so that algorithms can be defined that are capable of interfacing with each transparently.
   - **Example**: The Iterator abstraction is fundamental to generic programming. This strategy seeks to explicitly separate the notion of algorithm from that of data structure. The motivation is to
     - Promote component-based development
     - Boost productivity
     - Reduce configuration management.
   - **Notes**:
     - Enables *polymorphic traversal*
     - **Polymorphic Iterators** rely on **Factory Methods** to instantiate the appropriate Iterator subclass.
     - **Memento** is often used in conjunction with **Iterator**. An **Iterator** can use a **Memento** to capture the state

of an iteration. The **Iterator** stores the **Memento** internally.

Structural Patterns (7/21)

1. **Adapter** 🔌🔧
    - **Problem**: Adapter lets classes work together that couldn't otherwise because of incompatible interfaces. Creating a new design but re-using an existing (old) incompatible code.
    - **Solution**: **Wrap an existing class with a new interface**.Convert the interface of a class into another interface clients expect
    - **Example**: Inserting a new three-prong electrical plug in an old two-prong wall outlet needs an adapter.
    - **Notes**:
        - Adapter makes things work after they're designed; Bridge makes them work before they are.
        - **Facade** defines a new interface, whereas **Adapter** reuses an old interface.
2. **Bridge** 🏰
    - **Intent**:
        - To decouple an abstraction from its implementation so that the two can vary independently. **Abstraction** and **implementation** can be developed **independently** of each other.
        - Abstraction and Implementation can be altered structurally without affecting each other
        - Avoids permanent binding between Abstraction and an Implementation.
    - **Problem**: "Hardening of the software arteries" has occurred by using subclassing of an abstract base class to provide alternative implementations.
    - **Solution**: Decompose the component's interface and implementation into orthogonal class hierarchies.
    - **Example**: before and after.



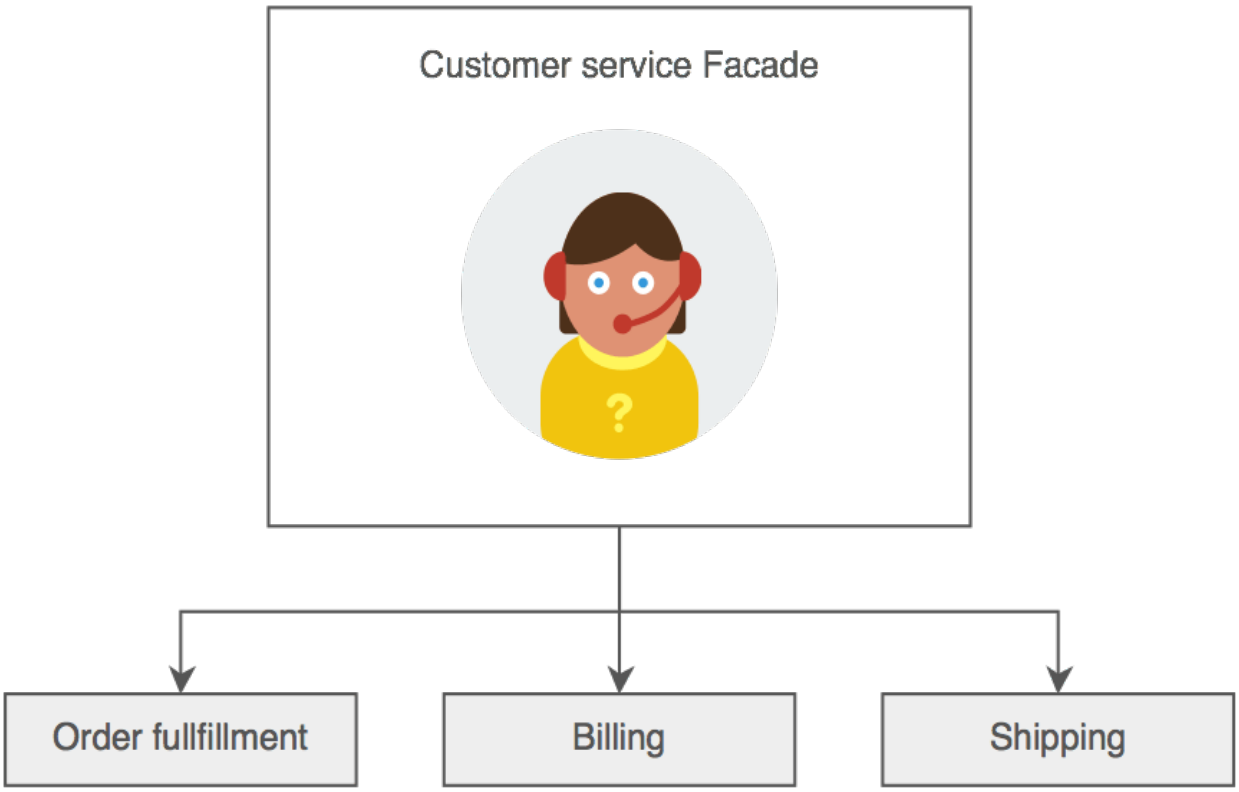


- **Notes**:
  - Abstraction = client interacts with.
  - Implementation = hidden/encapsulated (interface).
  - + Decoupling of the object's interface
  - + Improved extensibility (subclass abstractions and implementation independently)
  - + Hides details from the client
  - **Adapter** makes things work after they're designed; **Bridge** makes them work before they are.

3. **Composite** 🌳
   - **Intent**:
     - Compose objects into **tree** structures to represent **whole-part hierarchies** (a tree contains sub trees). Composite lets clients treat individual objects and compositions of objects uniformly.
     - At the heart of this pattern is the ability for a client to perform operations on an object without needing to know that there are many objects inside.
   - **Notes**:
     - Component = Root = Abstract Base Class
     - Composite = Nodes = (Possibly) Contains more composites
     - Leaf = Cannot have children
     - **Composite** and **Decorator** have similar structure diagrams, reflecting the fact that both rely on recursive composition to organize an open-ended number of objects.
     - **Composite** and **Decorator** are often used in concert.
     - **Composite** can be traversed with **Iterator**.
     - **Composite** can let you compose a **Mediator** out of smaller pieces through recursive composition
4. **Decorator** 🎁
   - **Intent**:
     - Attach additional responsibilities to an object dynamically at runtime (enhance it). Decorators provide a flexible alternative to subclassing for extending functionality.
     - Wrapping a gift, putting it in a box, and wrapping the box. -Acts as a wrapper to and ***existing*** class.
   - **Problem**: You want to add behaviour or state to individual objects at run-time. Inheritance is not feasible because it is static and applies to an entire class.
   - **Notes: Adapter** provides a ***different*** interface to its subject. **Proxy** provides the ***same*** interface. **Decorator**

provides an ***enhanced*** interface.

5. **Facade** 🎭
   - **Intent**:
     - Provide a unified **interface** to a package of classes. Facade defines a higher-level interface that makes the subsystem easier to use.
     - Adds an interface to existing system to hide its complexities.
     - Define a **singleton** which is the sole means for obtaining functionality from the package.
   - **Problem**: A segment of the client community needs a simplified interface to the overall functionality of a complex subsystem.
   - **Example**:

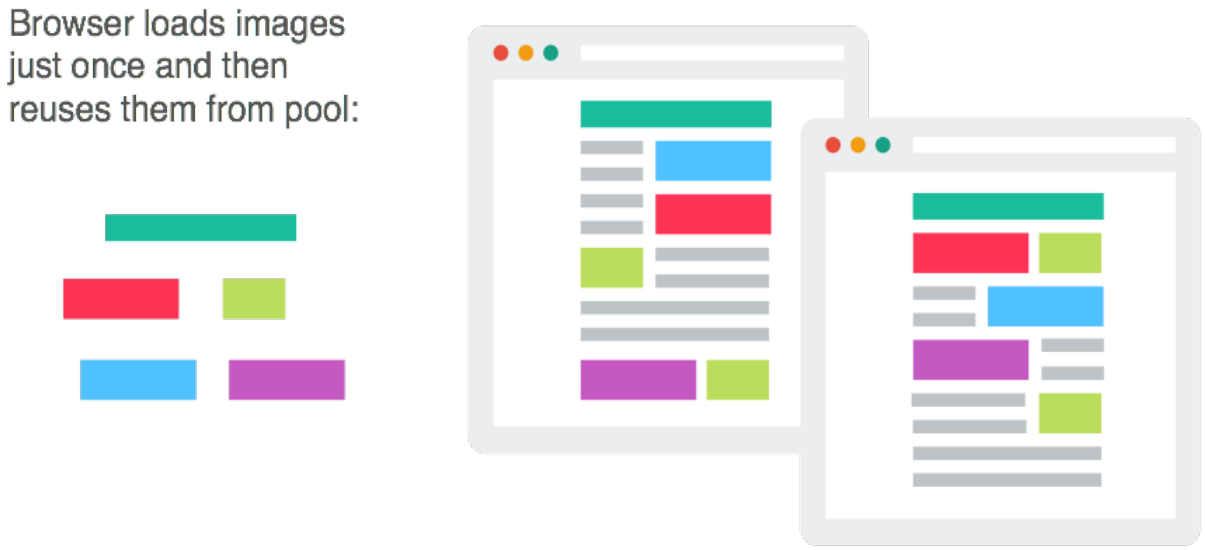


   - **Notes**:
     - +t.
     - + Shields the clients from subsystem components, thereby making the subsystem easier to use.
     - Be careful not to let the facade turn into all knowing "god" object.
     - **Abstract Factory** can be used as an alternative to **Facade** to hide platform-specific classes.
     - **Facade** defines a new interface, whereas **Adapter** uses an old interface. Adapter and Facade are both wrappers. Remember that Adapter makes two existing interfaces work together as opposed to defining an entirely new one.
     - **Flyweight** shows how to make lots of little objects, **Facade** shows how to make a single object represent an entire subsystem.
6. **Flyweight** 🥄(cache)
   - **Intent**:

- Primarily used to reduce the number of objects created and to decrease memory footprint and increase performance.
- Tries to reuse already existing similar kind objects by storing them and creates new object when no matching object is found (like a cache).
- Use ***sharing*** to support large numbers of fine-grained objects efficiently (millions of client requests).
- Divided into the state-**dependent** (*extrinsic*) parts, and the state-**independent** (*intrinsic*) part (*shared*).

- **Problem**: Designing objects down to the lowest levels of system "granularity" provides optimal flexibility, but can be unacceptably expensive in terms of performance and memory usage.

- **Example**: Modern browsers cache images when visiting a web page. When viewing the webpage a second time the cached instance is shown.



- **Notes**:
  - **Flyweight** shows how to make lots of little objects, **Facade** shows how to make a single large object represent an entire subsystem.
  - Flyweight is often combined with **Composite** to implement shared leaf nodes.

7. **Proxy**
   - **Intent**:
     - Provide a **surrogate** or **placeholder** for another object to control access to it.
     - Avoid the unnecessary execution of expensive functionality in a manner transparent to clients.
     - In proxy pattern, a class represents functionality of another class.

   - **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.

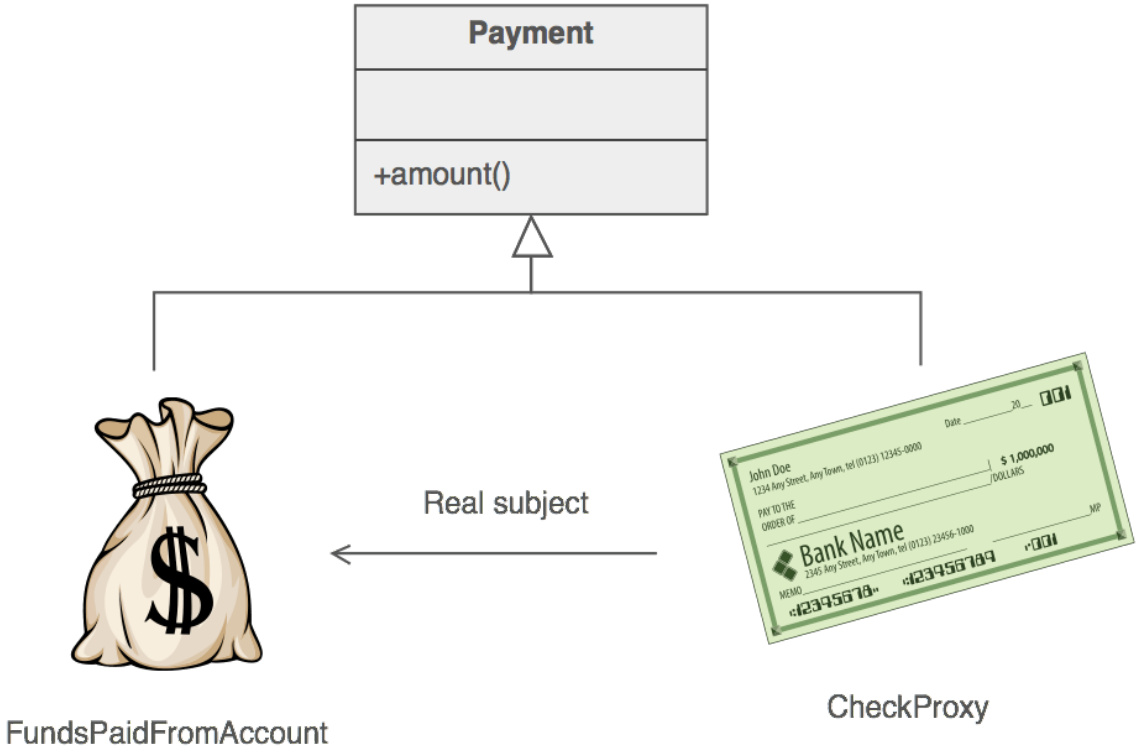
   - **Example**:

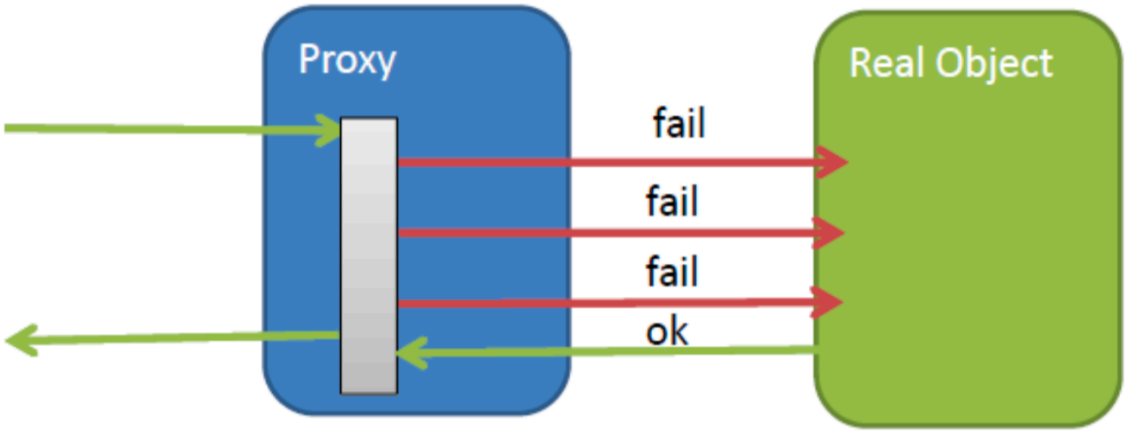
     - A ***virtual proxy*** is a placeholder for ***expensive to create*** objects. The real object is only created when a client first requests/accesses the object (Unit of Work).

     - 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.

- A ***smart proxy*** interposes additional actions.
- A ***remote proxy*** provides a local representative for an object that resides in a different address space.





- **Notes**:
  - **Adapter** provides a different interface to its subject. **Proxy** provides the same interface. **Decorator** provides an enhanced interface.
  - **Decorator** and Proxy have different purposes but similar structures. Both describe how to provide a level of *indirection* to another object, and the implementations keep a reference to the object to which they forward requests.d