DESIGN PATTERNS

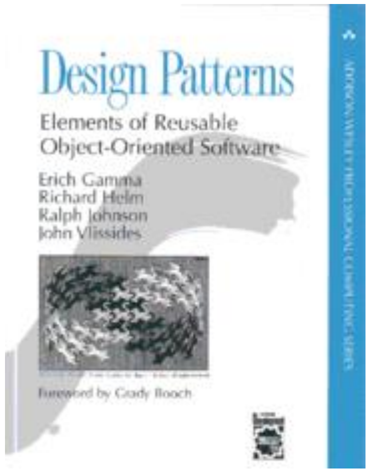
RAJA SEKHAR TADEPALLI

HISTORY OF PATTERNS

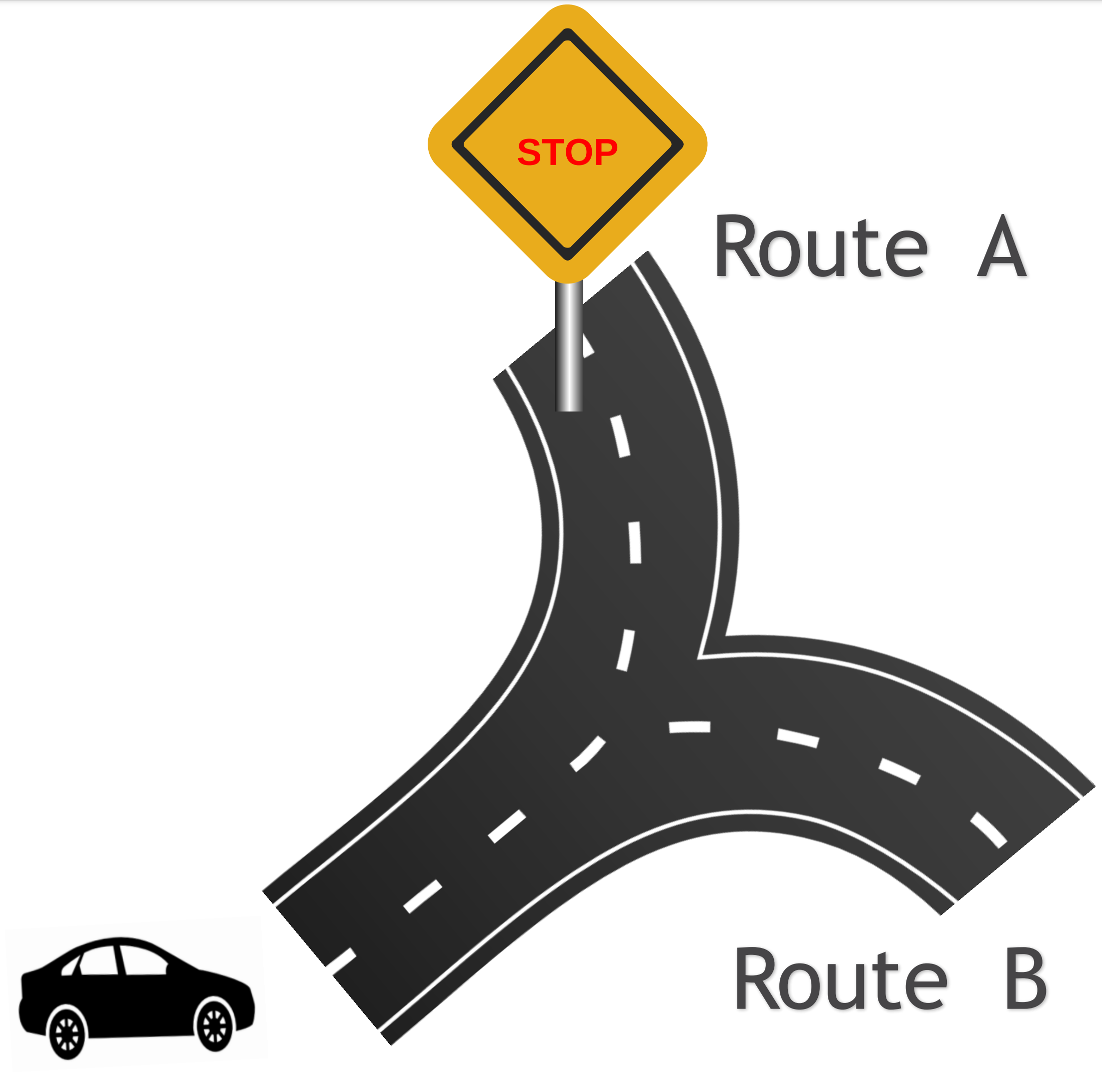
- The concept of a "pattern" was first expressed in Christopher Alexander's work *A Pattern Language* in 1977 (2543 patterns)

- In 1990 a group called the Gang of Four or "GoF" (Gamma, Helm, Johnson, Vlissides) compile a catalog of design patterns

- 1995 book *Design Patterns: Elements of Reusable Object-Oriented Software* is a classic of the field



WHAT IS A PATTERN



Destination – EPAM Office

WHAT IS DESIGN PATTERN

Template to solve a problem

Obtained by trial and error

Solution to general problems

Best Practices

Development process is fast because of tested & proven development paradigms

Common Language among Developers/Architects

WHAT IS A DESIGN PATTERN

“Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice”

- *Christopher Alexander*

Definition

Design patterns are repeatable / reusable solution to commonly occurring Problems in a certain context in Software Design

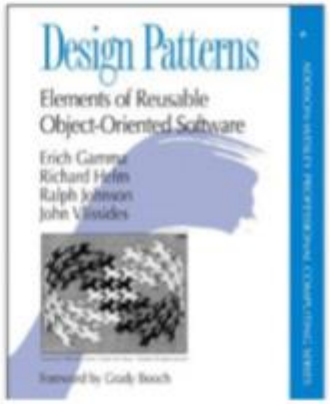
Gang of Four (GOF)

John Vlissides

Erich Gamma

Ralph Johnson

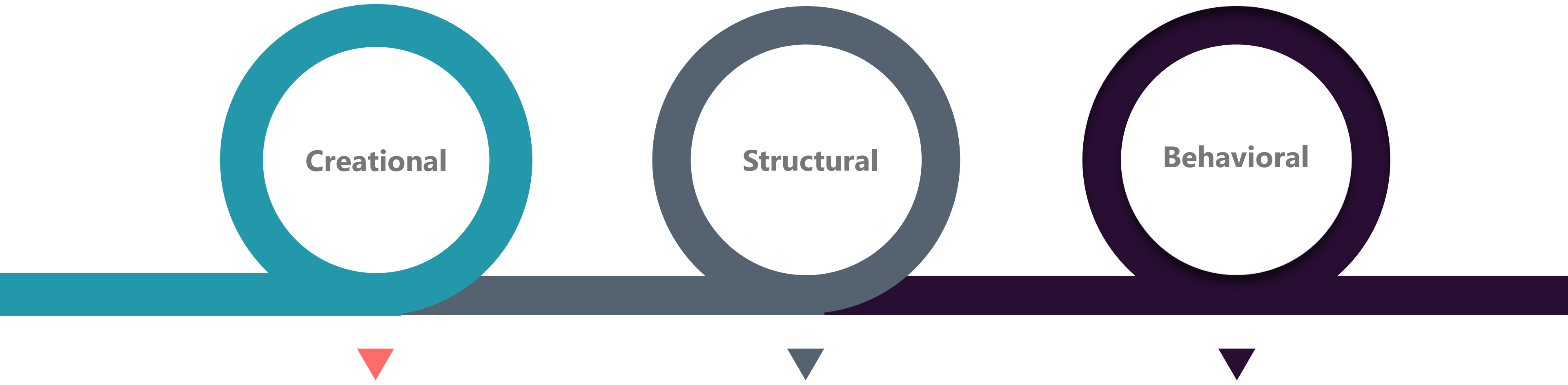
Richard Helm



Program to an interface not an implementation

Favor object composition over inheritance

Gang of Four (GOF)



- Way to Create Objects while hiding the Creation Login
- Flexibility to decide the way in which objects will be created for a given use case.

- Focused on how classes and objects can be composed, to form larger structures.
- Simplifies the structure by identifying the relationship

- Concerned with the assignment of responsibilities between objects or encapsulating behavior in an object and delegating requests to it.

GOF Design Patterns

Types of GOF Patterns

1. **Creational patterns**

    1. Abstract factory pattern groups object factories that have a common theme.
    2. Builder pattern constructs complex objects by separating construction and representation.
    3. Factory method pattern creates objects without specifying the exact class to create.
    4. Prototype pattern creates objects by cloning an existing object.
    5. Singleton pattern restricts object creation for a class to only one instance.

2. **Structural**

    1. Adapter allows classes with incompatible interfaces to work together by wrapping its own interface around that of an already existing class.
    2. Bridge decouples an abstraction from its implementation so that the two can vary independently.
    3. Composite composes zero-or-more similar objects so that they can be manipulated as one object.
    4. Decorator dynamically adds/overrides behavior in an existing method of an object.
    5. Facade provides a simplified interface to a large body of code.
    6. Flyweight reduces the cost of creating and manipulating a large number of similar objects.
    7. Proxy provides a placeholder for another object to control access, reduce cost, and reduce complexity.

GOF Design Patterns

3. Behavioral Patterns

1. Chain of responsibility delegates commands to a chain of processing objects.
2. Command creates objects which encapsulate actions and parameters.
3. Interpreter implements a specialized language.
4. Iterator accesses the elements of an object sequentially without exposing its underlying representation.
5. Mediator allows loose coupling between classes by being the only class that has detailed knowledge of their methods.
6. Memento provides the ability to restore an object to its previous state (undo).
7. Observer is a publish/subscribe pattern which allows a number of observer objects to see an event.
8. State allows an object to alter its behavior when its internal state changes.
9. Strategy allows one of a family of algorithms to be selected on-the-fly at runtime.
10. Template method defines the skeleton of an algorithm as an abstract class, allowing its subclasses to provide concrete behavior.
11. Visitor separates an algorithm from an object structure by moving the hierarchy of =methods into one object.

Design Principle vs Design Pattern

- **Design Principle**
  - It provides high level guidelines to design better software applications. Design principles do not provide implementation and not bound to any programming language. E.g. SOLID (SRP, OCP, LSP, ISP, DIP) principles.
  - For example, Single Responsibility Principle (SRP) suggests that a class should have only one and one reason to change. This is high level statement which we can keep in mind while designing or creating classes for our application. SRP does not provide specific implementation steps but it's on you how you implement SRP in your application.
- **Design Pattern**
  - It provides low level solution (implementation) for the commonly occurring object-oriented problem. In another word, design pattern suggest specific implementation for the specific object-oriented programming problem. For example, if you want create a class that can only have one object at a time then you can use Singleton design pattern which suggests the best way to create a class that can only have one object.
  - Design patterns are tested by others and safe to follow. E.g. Gang of Four patterns: Abstract Factory, Factory, Singleton, Command etc.

CREATIONAL PATTERNS

Abraham Became First President of States.

ABSTRACT FACTORY PATTERN

BUILDER PATTERN

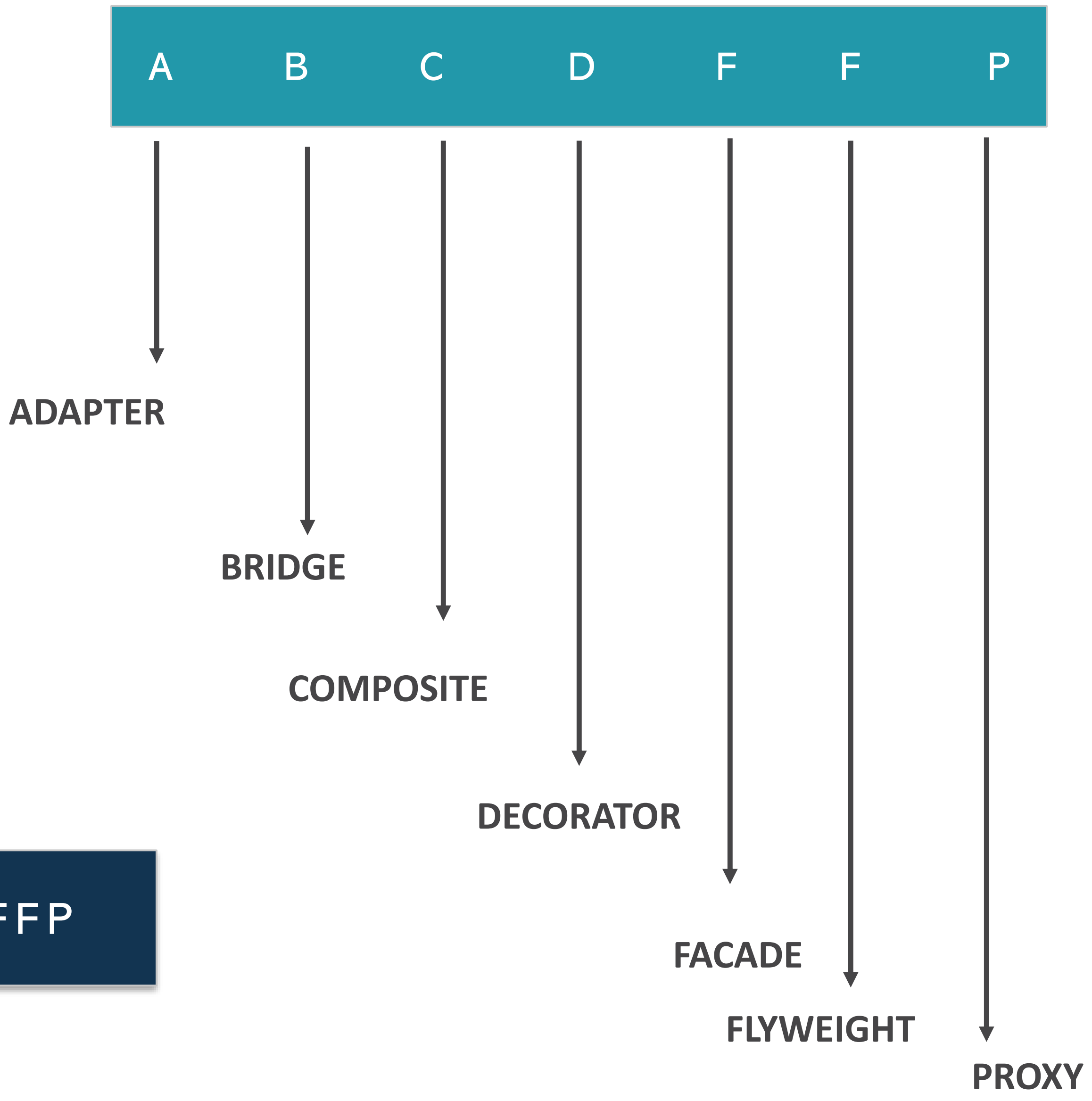
FACTORY METHOD

PROTOTYPE

SINGLETON

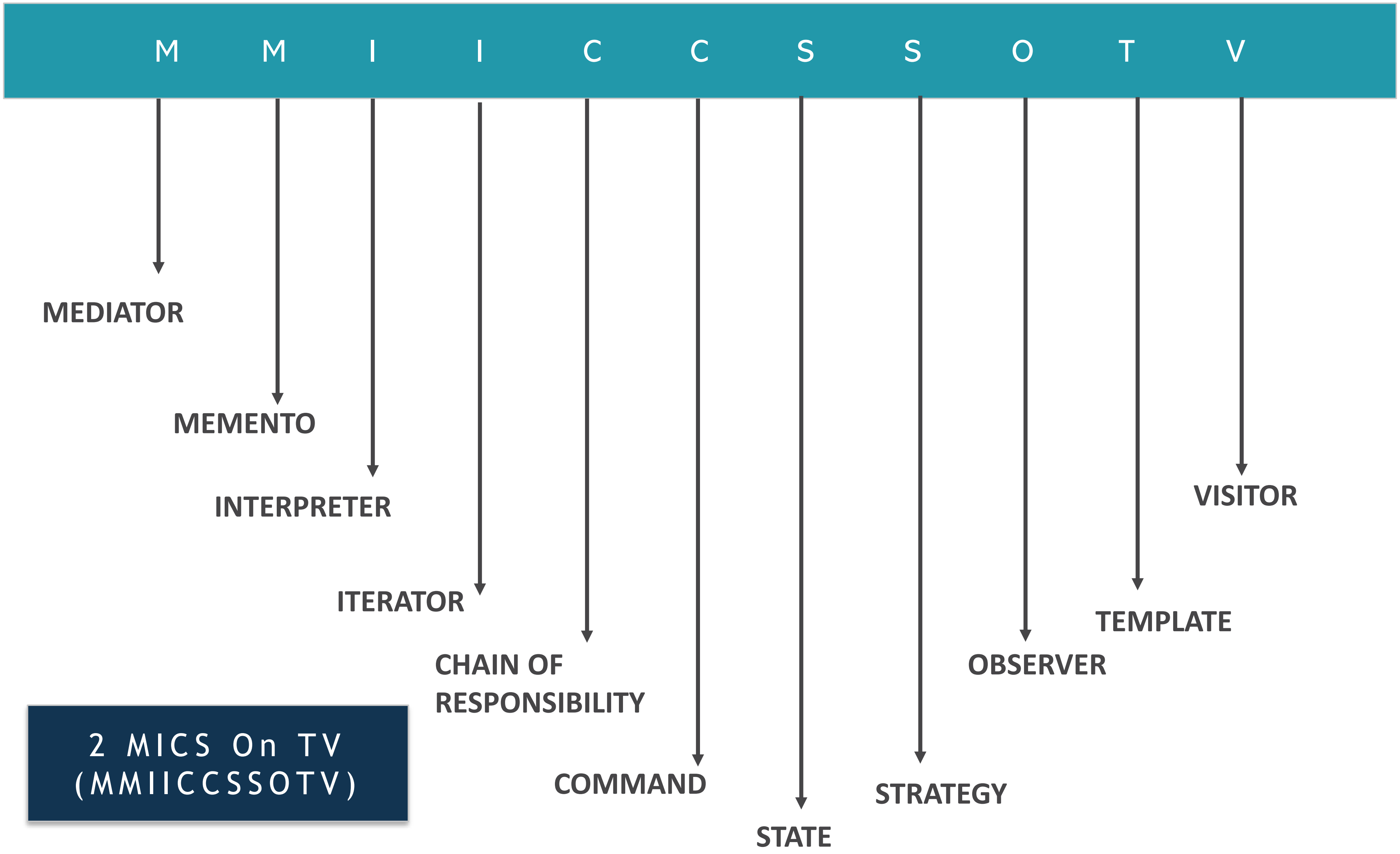
ABFPS

STRUCTURAL PATTERNS



ABCDFFP

BEHAVIORAL PATTERNS



2 MICS On TV
(MMIICCSSOTV)

CREATIONAL PATTERNS

SINGLETON

Singleton is a creational design pattern that lets you ensure that a class has only one instance, while providing a global access point to this instance.

The government is an excellent example of the Singleton pattern. A country can have only one official government.

Government

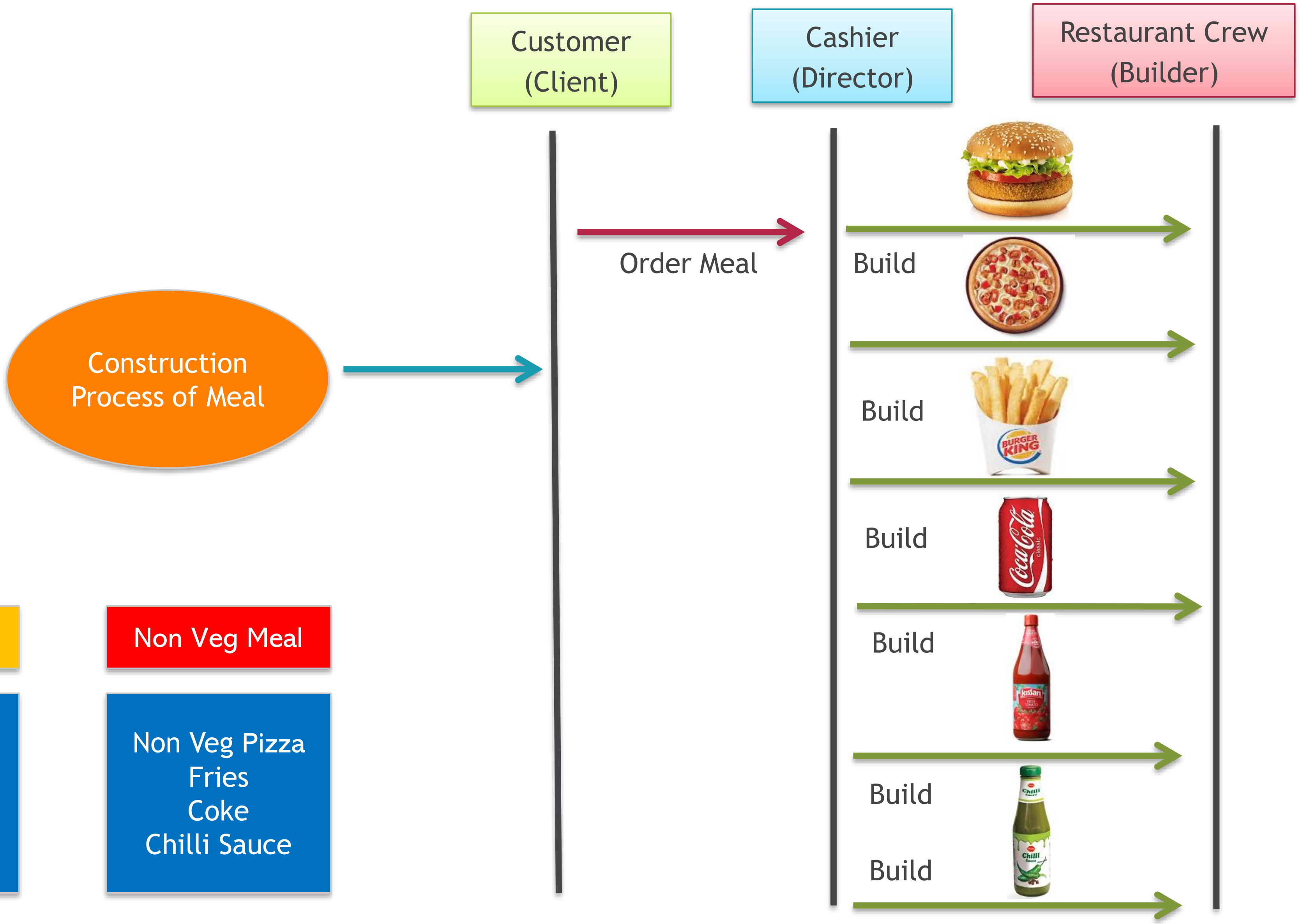
Travel Card

HOW TO ACHIEVE & WHEN TO USE SINGLETON

- **How to Achieve?**
  - Ensure that a class has just a single instance.
  - Provide a global access point to that instance.

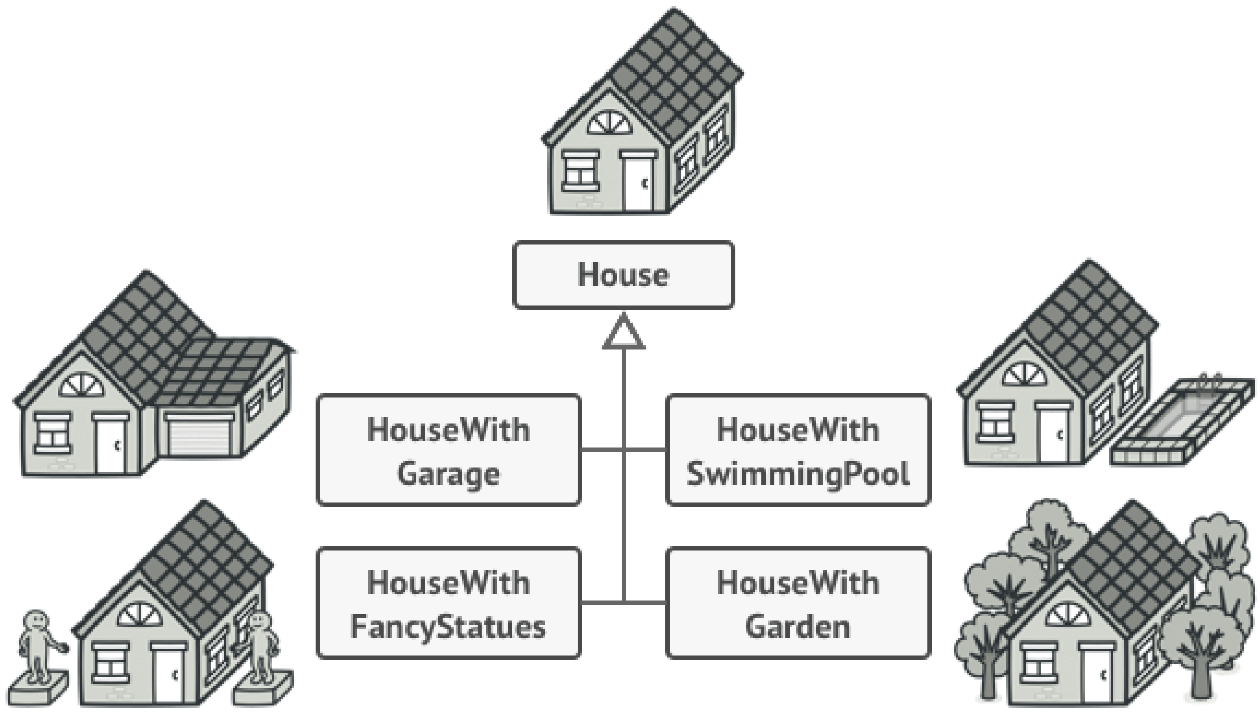
- **Best Use**
  - Logging
  - Caches
  - Registry Settings
  - Access External Resources
    - Printer
    - Device Driver
    - Database

Builder Pattern



Veg Meal	Non Veg Meal
Veg Burger Fries Coke Tomato Sauce	Non Veg Pizza Fries Coke Chilli Sauce

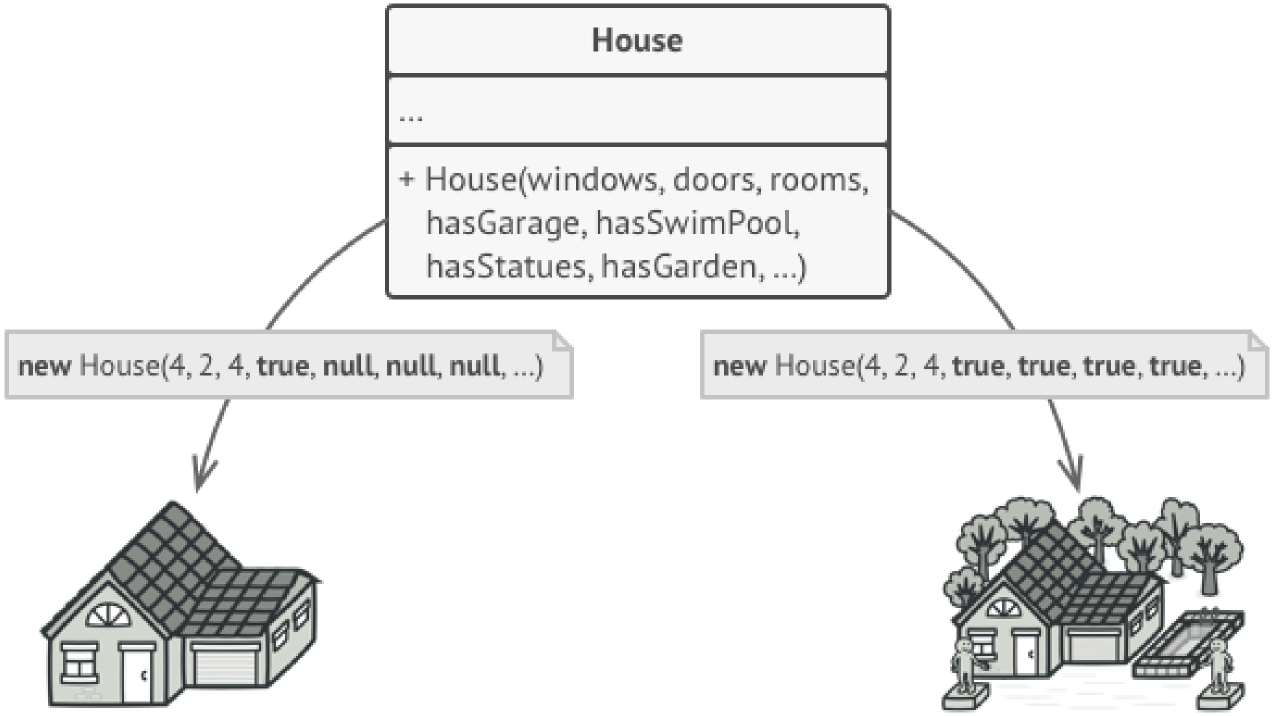
Builder Pattern



You might make the program too complex by creating a subclass for every possible configuration of an object.

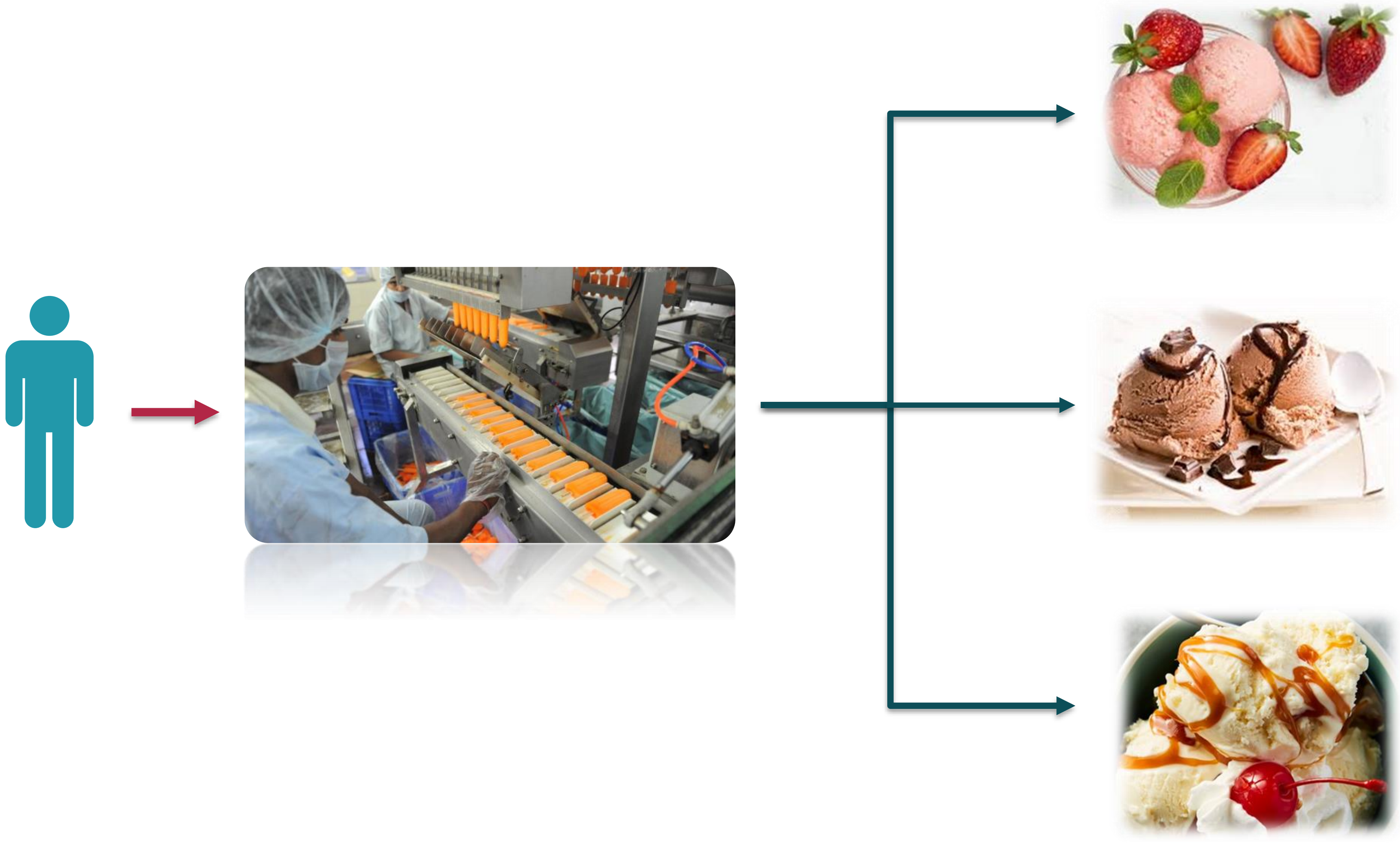
Builder Pattern

Extend the **base** House class and create a set of **subclasses** to cover all combinations of the parameters.


House
...
+ House(windows, doors, rooms, hasGarage, hasSwimPool, hasStatues, hasGarden, ...)
new House(4, 2, 4, true, null, null, null, ...)
new House(4, 2, 4, true, true, true, true, ...)

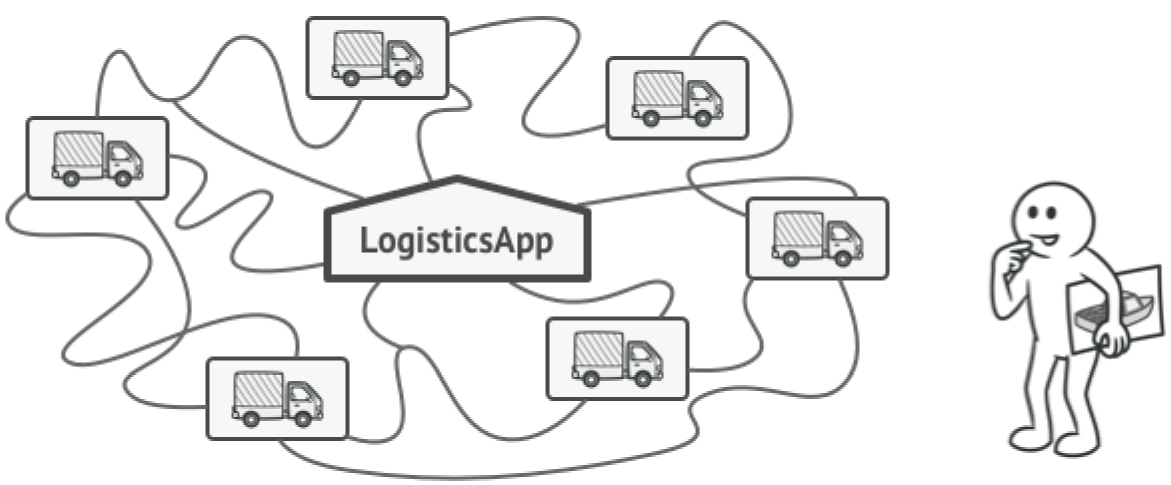
The constructor with lots of parameters has its downside: not all the parameters are needed at all times.

FACTORY



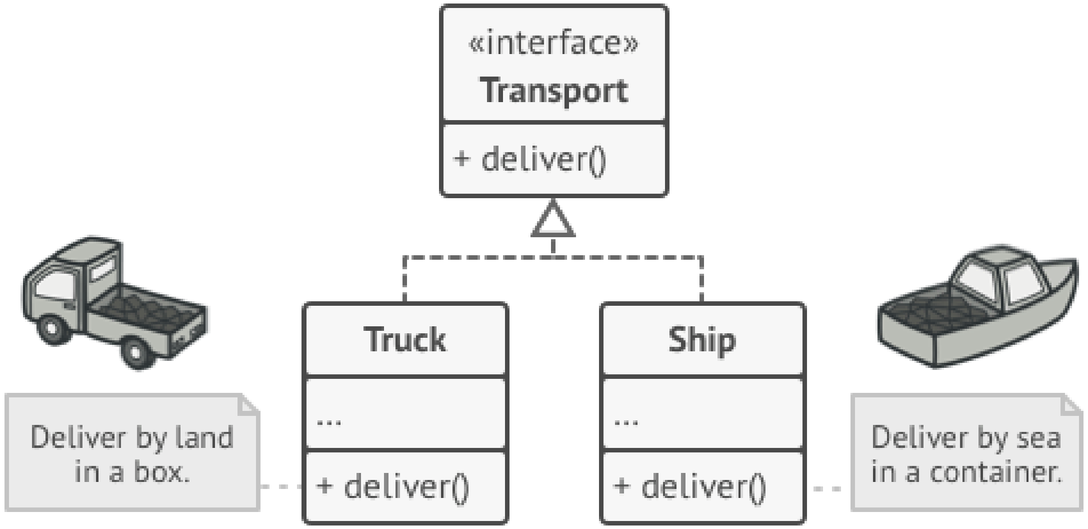
FACTORY

Factory Method is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.



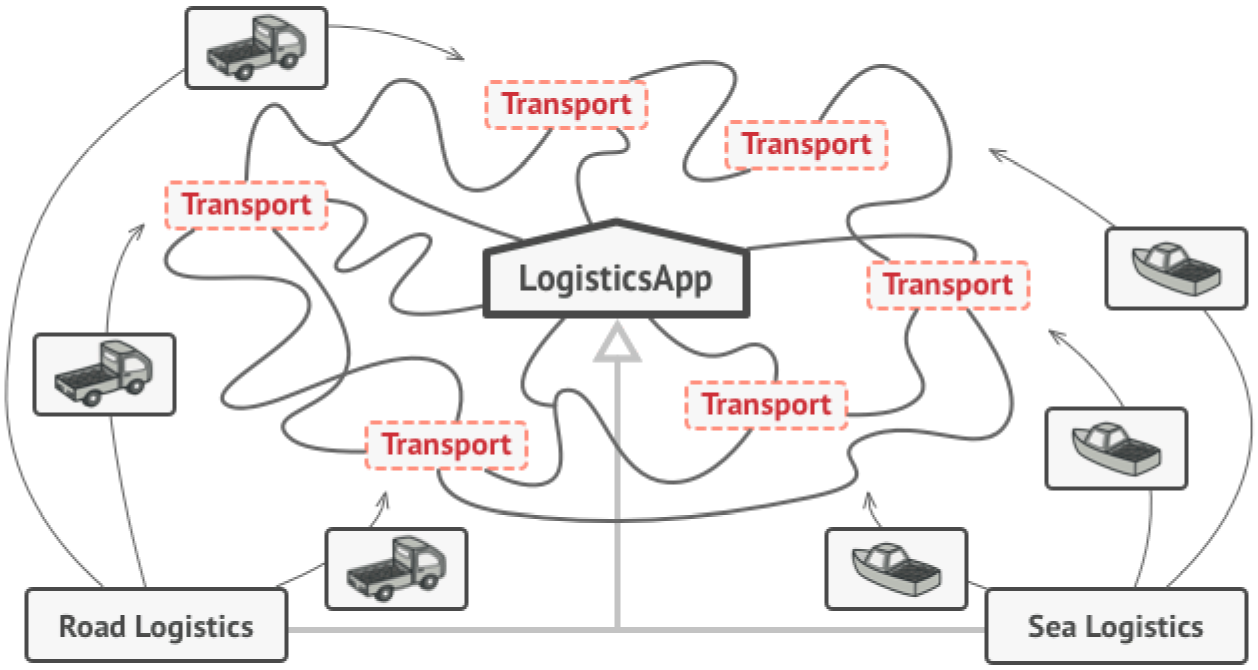
Adding a new class to the program isn't that simple if the rest of the code is already coupled to existing classes.

HOW TO ACHIEVE FACTORY



All products must follow the same interface.

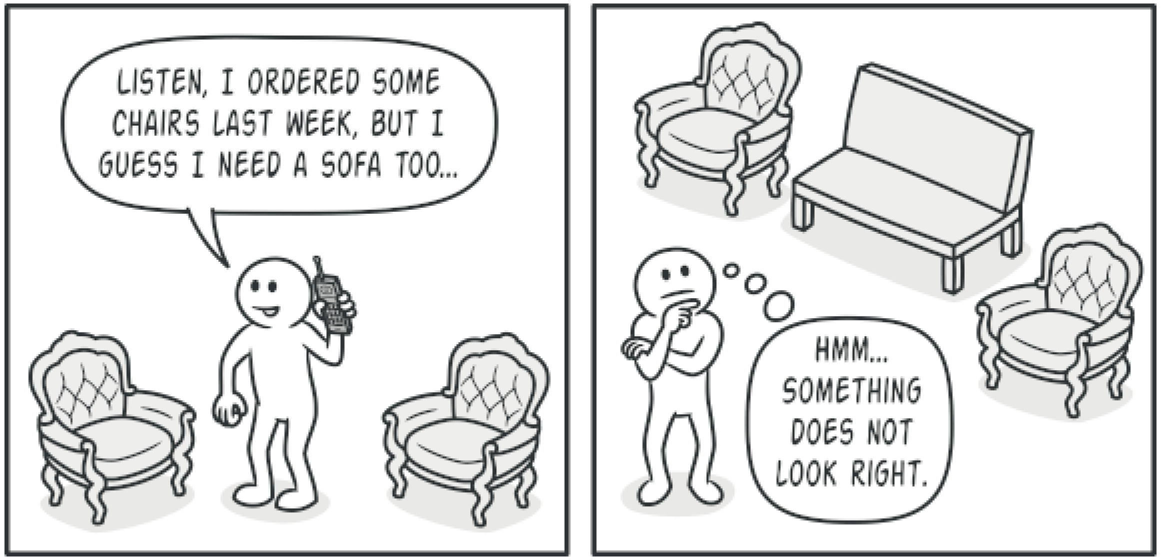
HOW TO ACHIEVE FACTORY



As long as all product classes implement a common interface, you can pass their objects to the client code without breaking it.

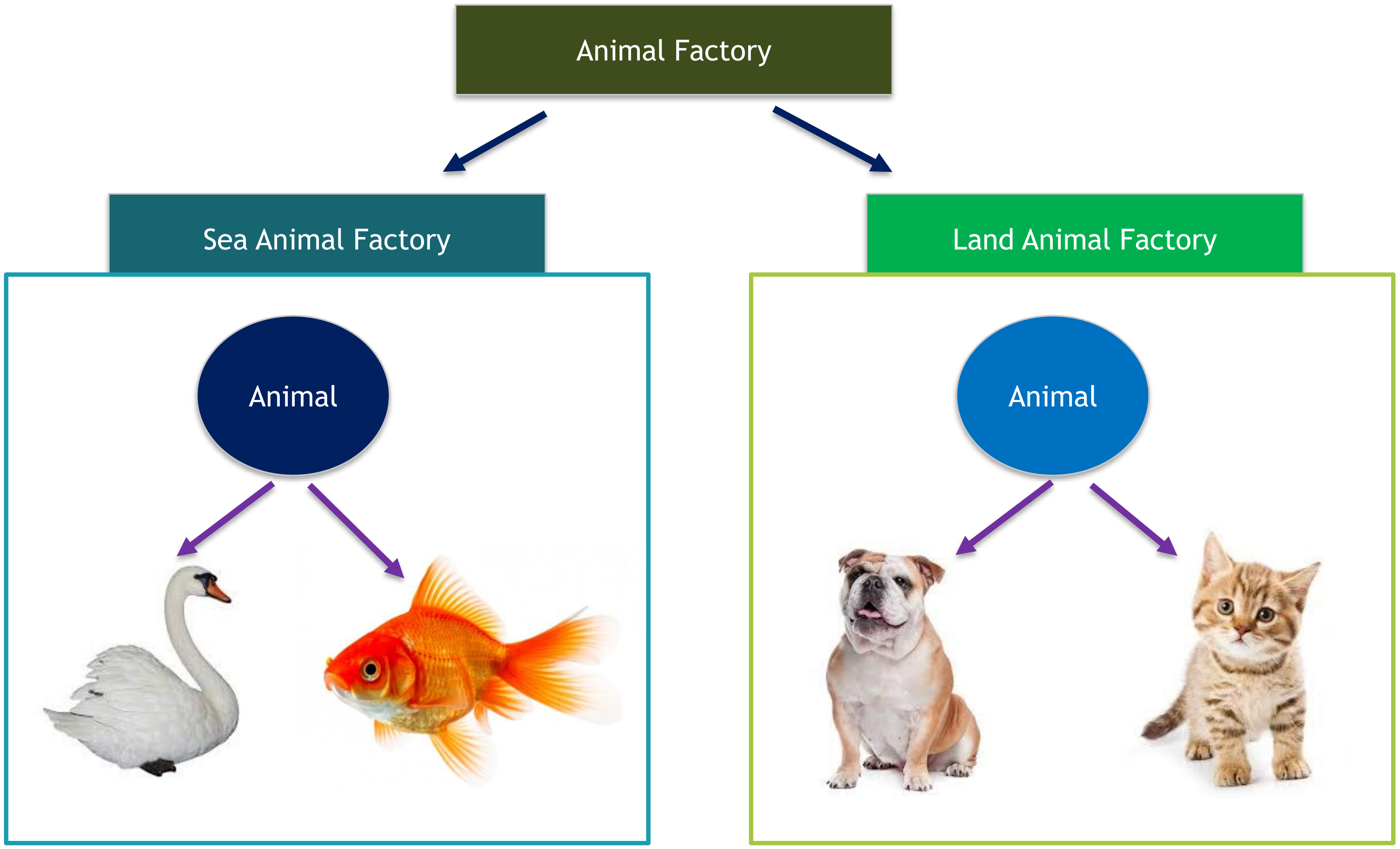
Abstract Factory

Abstract Factory is a creational design pattern that lets you produce **families of related objects** without specifying their concrete classes.

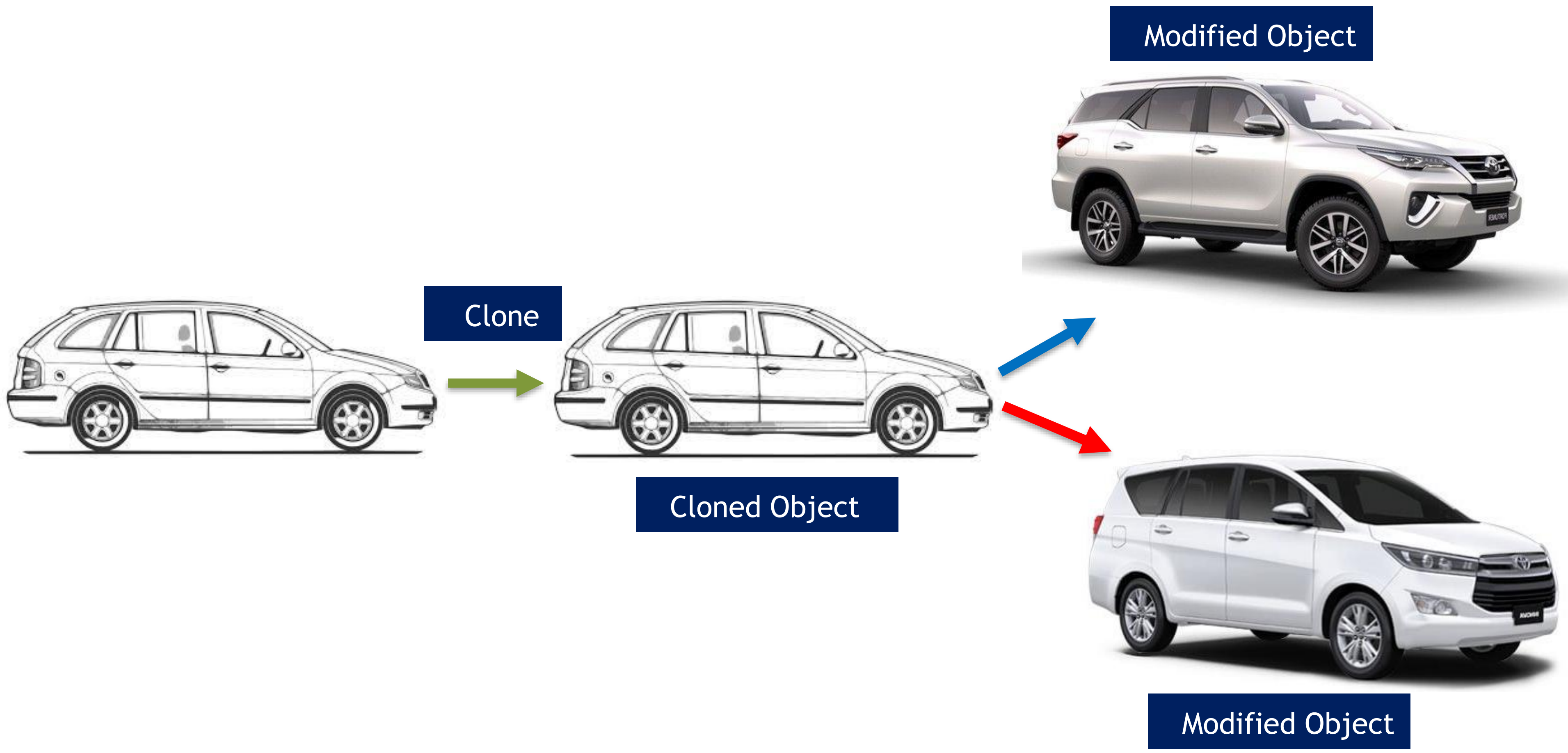

LISTEN, I ORDERED SOME CHAIRS LAST WEEK, BUT I GUESS I NEED A SOFA TOO...
HMM... SOMETHING DOES NOT LOOK RIGHT.

A Modern-style sofa doesn’t match Victorian-style chairs.

Abstract Factory



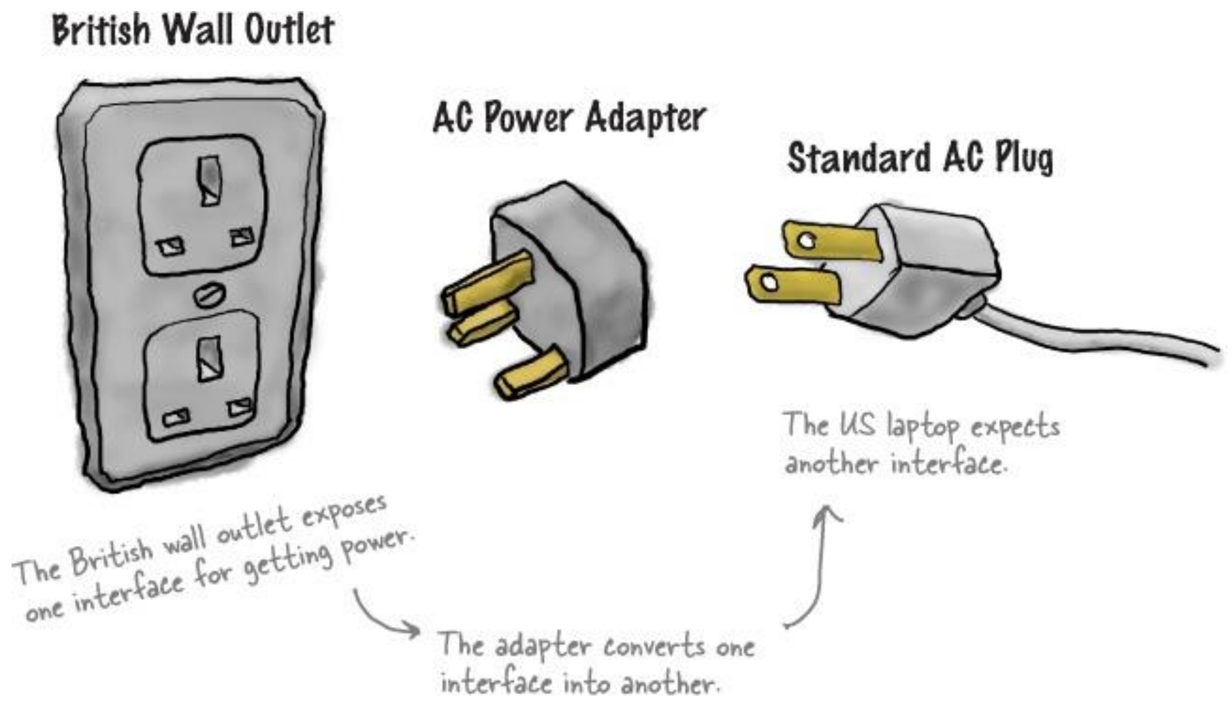
Prototype Pattern



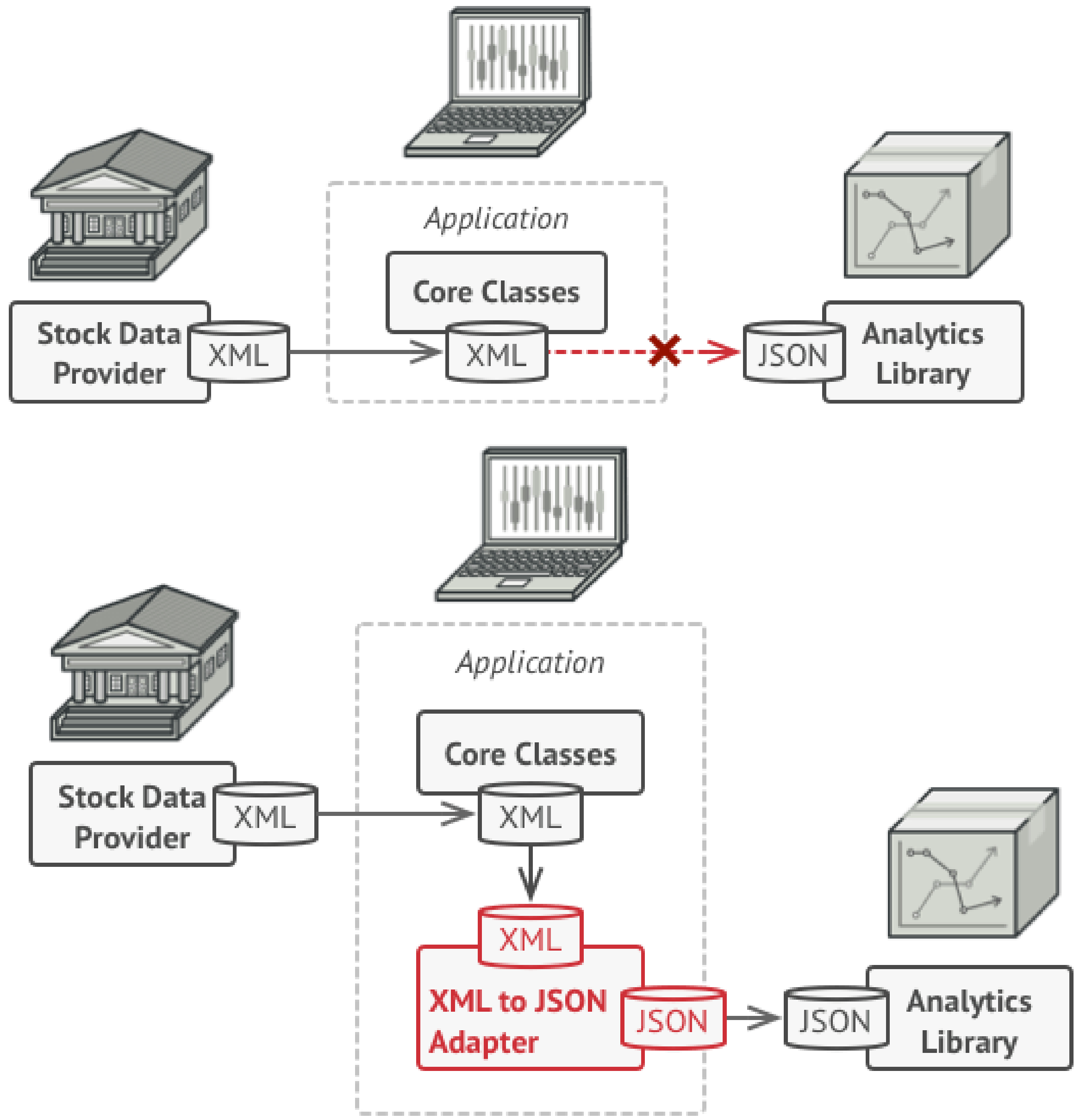
STRUCTURAL PATTERNS

ADAPTER PATTERN

Adapter is a structural design pattern that allows objects with **incompatible** interfaces to collaborate.

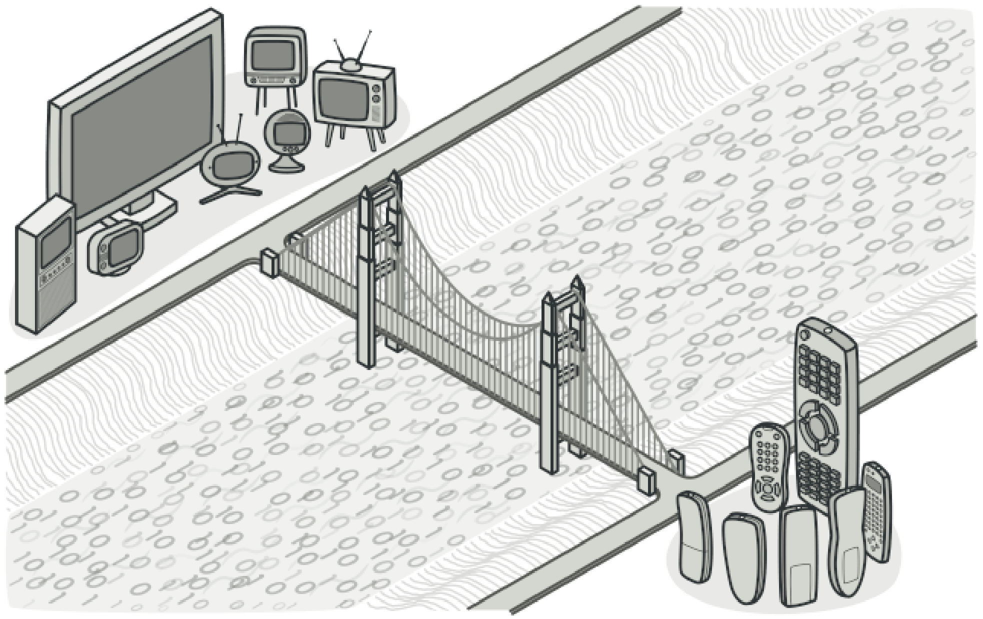

British Wall Outlet
AC Power Adapter
Standard AC Plug
The US laptop expects another interface.
The British wall outlet exposes one interface for getting power.
The adapter converts one interface into another.

ADAPTER PATTERN



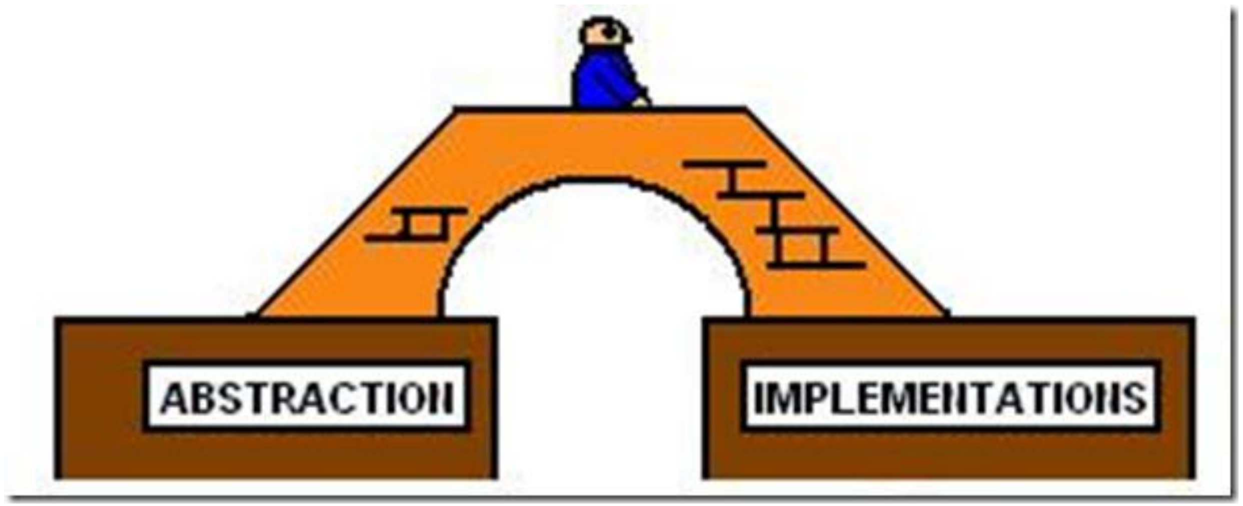
BRIDGE PATTERN

Bridge is a structural design pattern that lets you split a large class or a set of closely related classes into two separate hierarchies—abstraction and implementation—which can be developed independently of each other.



TV and remote are 2 separate entities. As an architect of remote one way is to abstract all remote in an interface and develop it, however the challenge is in the way both TV and remote development is getting mixed or interlocked – in technical terms getting tightly coupled. Changes in one might effect the another. So to solve it, We have created a bridge between TV and remote. Thus decoupled TV and Remote. Please download example and try on your own.

Bridge Pattern

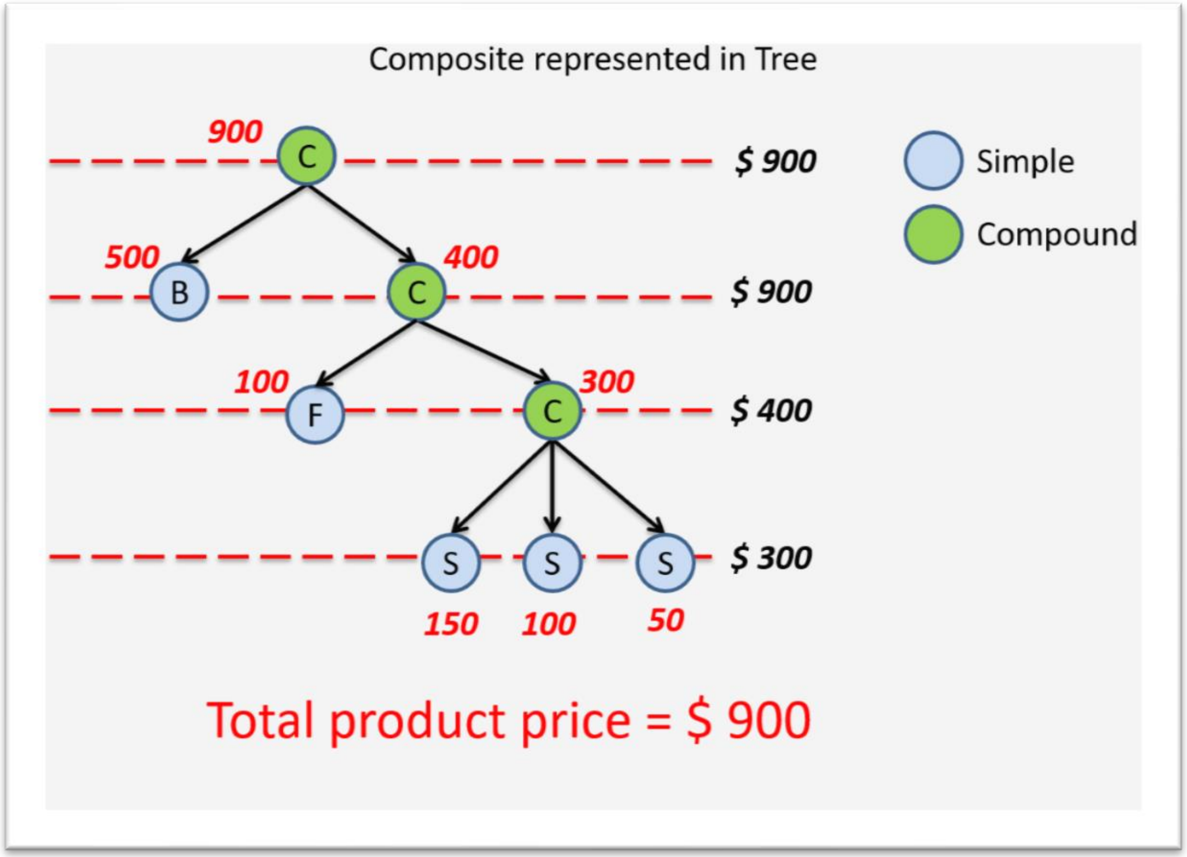


“Decouple an abstraction from its implementation so that the two can vary independently” is the intent for bridge design pattern as stated by GoF.
Bridge design pattern is a modified version of the notion of “prefer composition over inheritance”.Decouple implentation from interface and hiding implementation details from client is the essence of bridge design pattern.

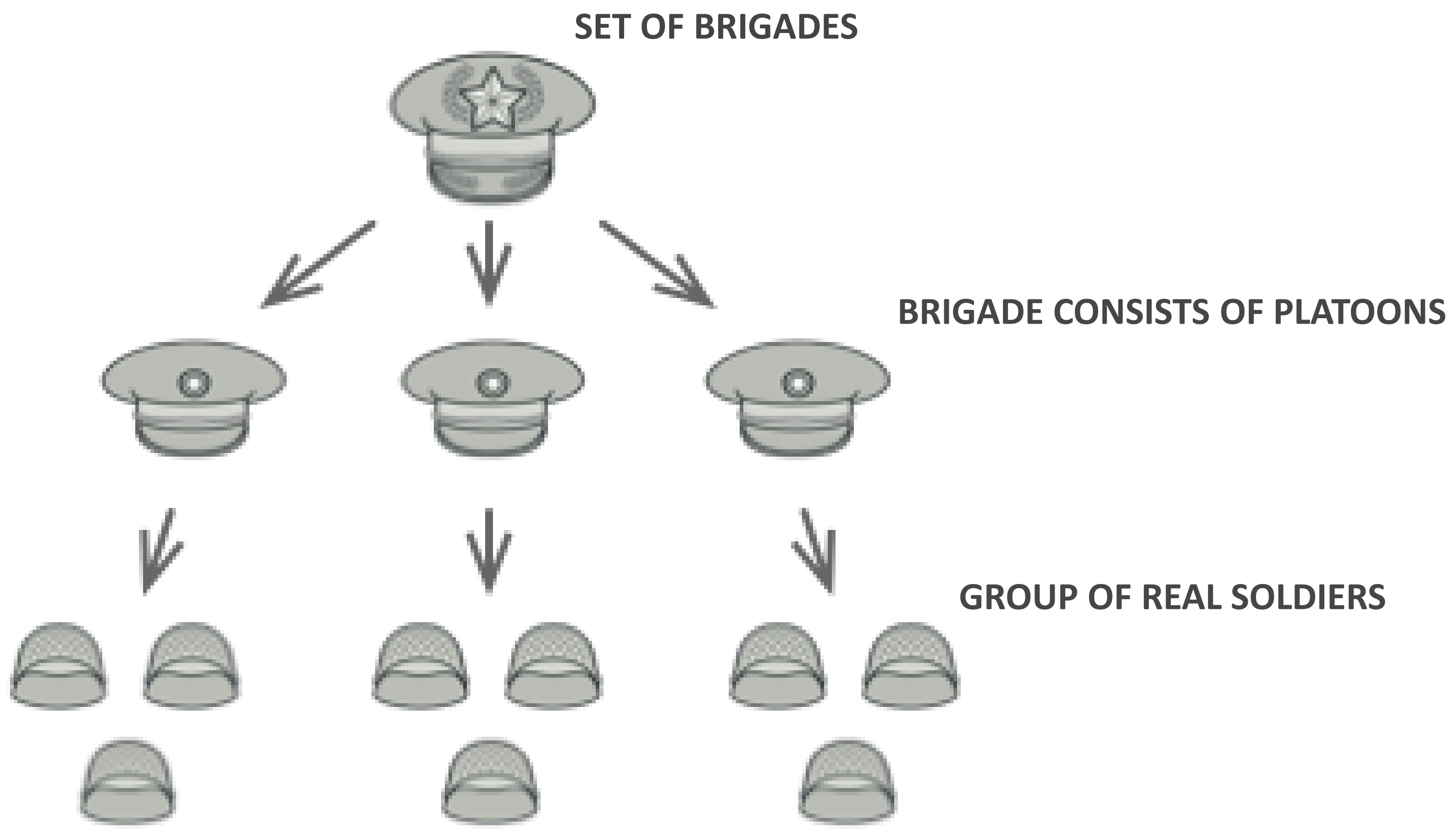
Bridge is used where we need to decouple an abstraction from its implementation so that the two can vary independently.

COMPOSITE PATTERN

Composite is a structural design pattern that lets you compose objects into tree structures and then work with these structures as if they were individual objects.



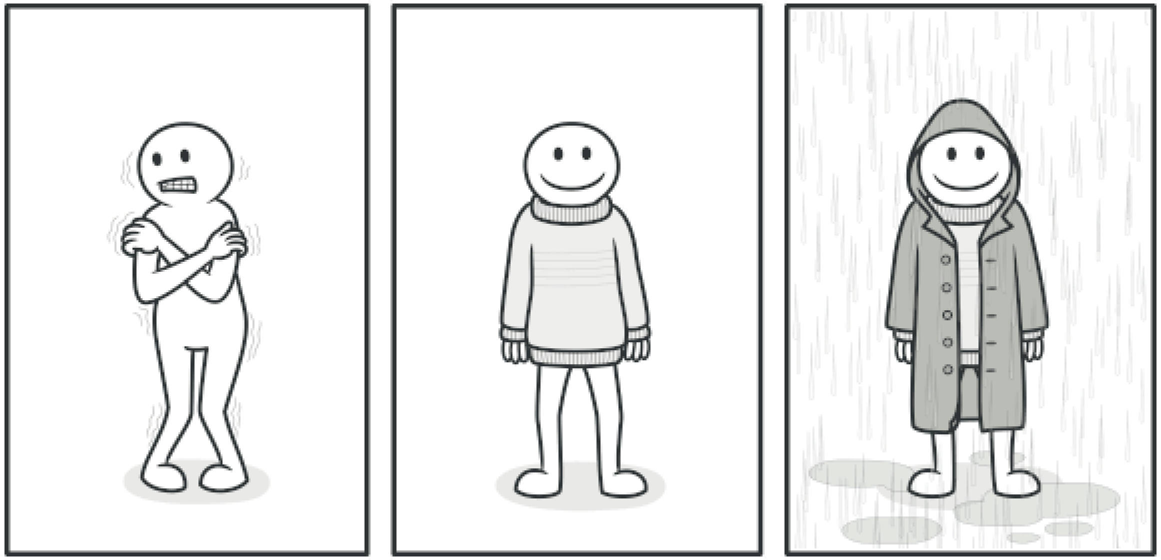
COMPOSITE PATTERN



An example of a military structure.

DECORATOR PATTERN

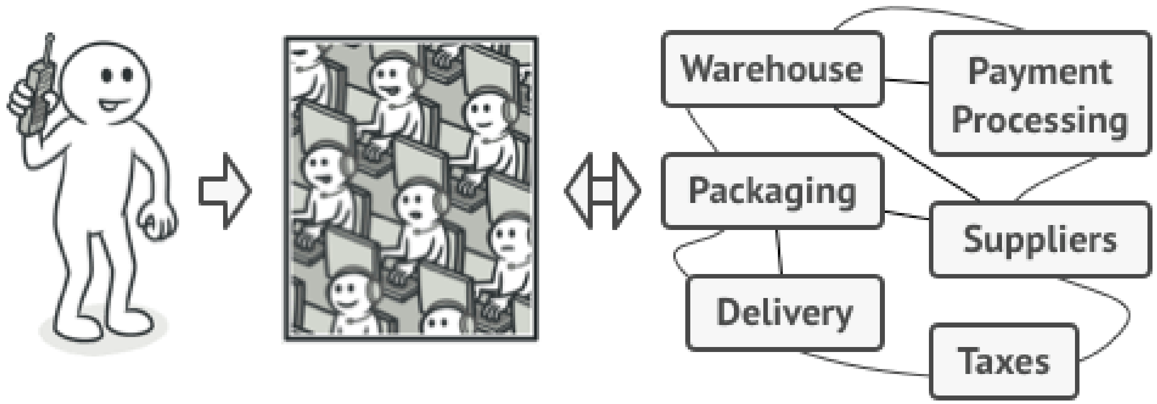
Decorator is a structural design pattern that lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.



You get a combined effect from wearing multiple pieces of clothing.

FACADE PATTERN

Facade is a structural design pattern that provides a simplified interface to a library, a framework, or any other complex set of classes.

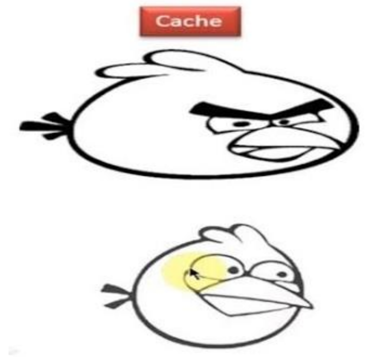


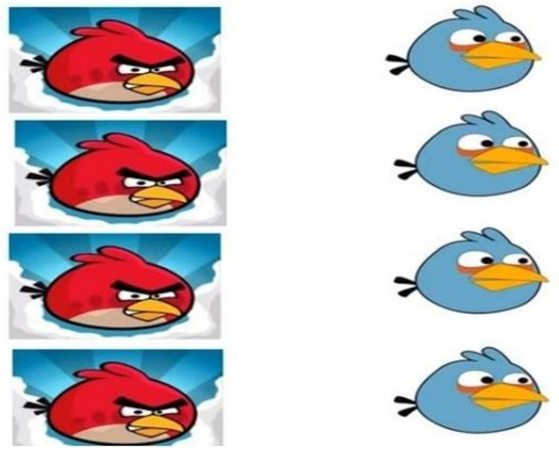
Placing orders by phone.

When you call a shop to place a phone order, an operator is your facade to all services and departments of the shop. The operator provides you with a simple voice interface to the ordering system, payment gateways, and various delivery services.

FLYWEIGHT PATTERN

- As much as possible **reduction used memory** wasted for servicing many similar objects.
- Replacing so-called **heavy objects** on light objects.
- The use of **object sharing** for effective management of many objects, i.e. we do not create every object from the beginning, we only base on already created objects thanks to this **we increase the application speed**.





PROXY PATTERN

INTENT

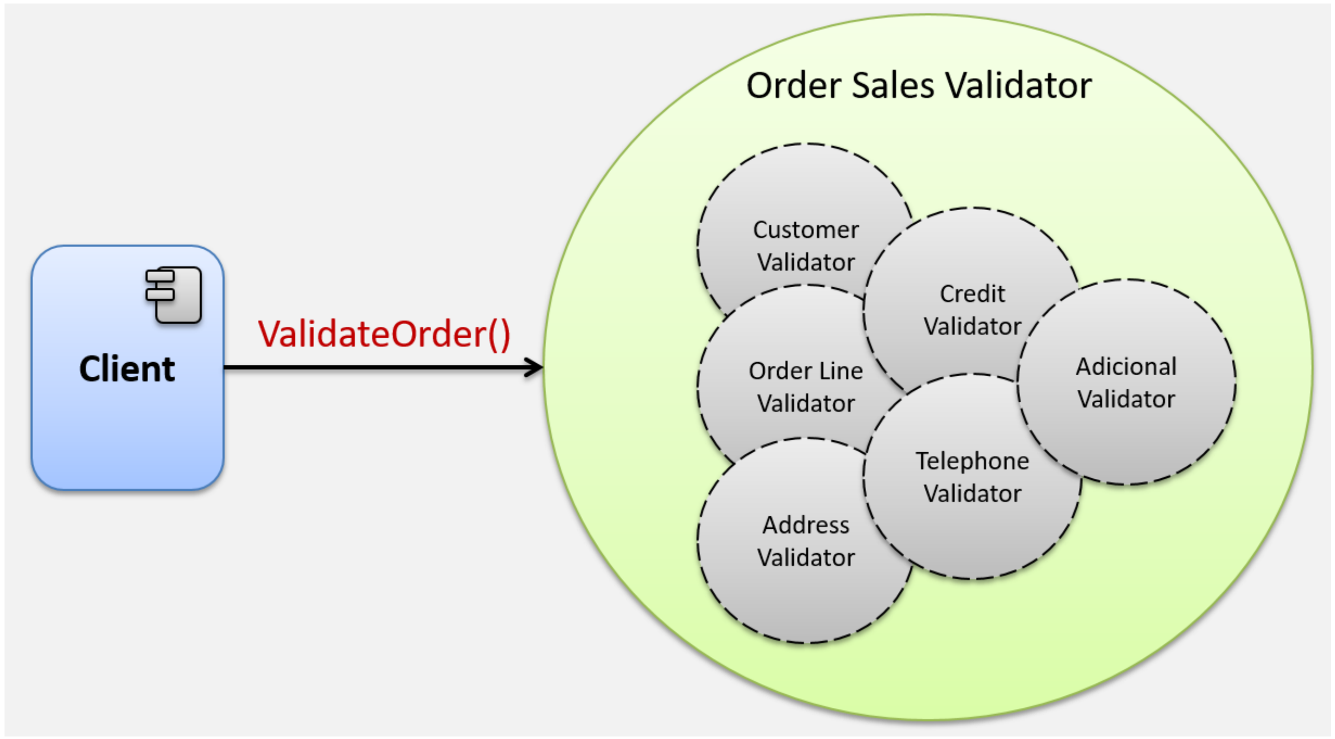
- Proxy is a structural design pattern that lets you provide a substitute or placeholder for another object. A proxy controls access to the original object, allowing you to perform something either before or after the request gets through to the original object.

BEHAVIORAL PATTERNS

CHAIN OF RESPONSIBILITY PATTERN

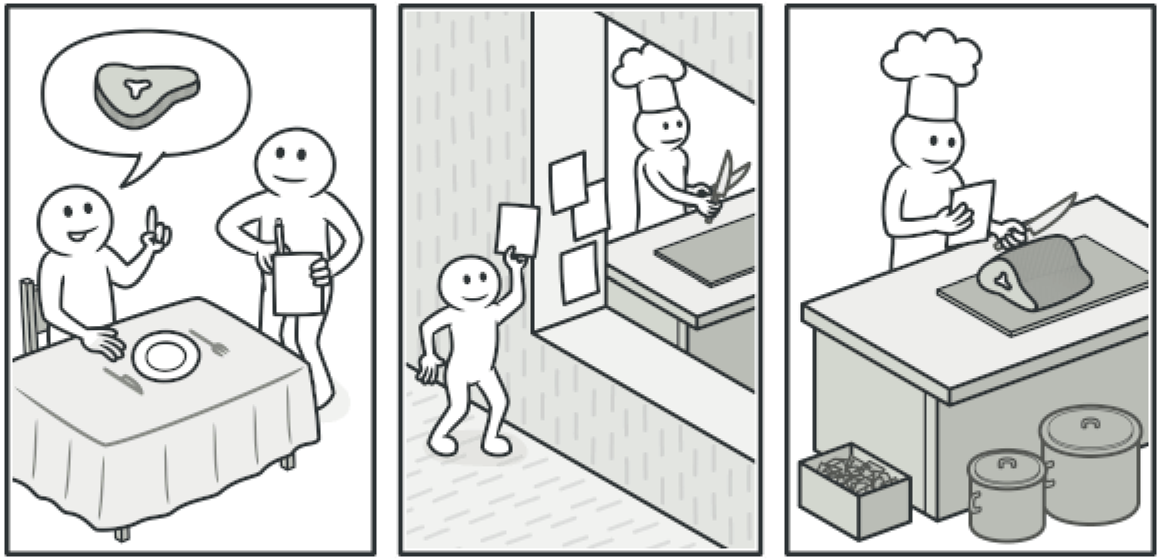
INTENT

- Chain of Responsibility is a behavioral design pattern that lets you pass requests along a chain of handlers. Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain.



COMMAND PATTERN

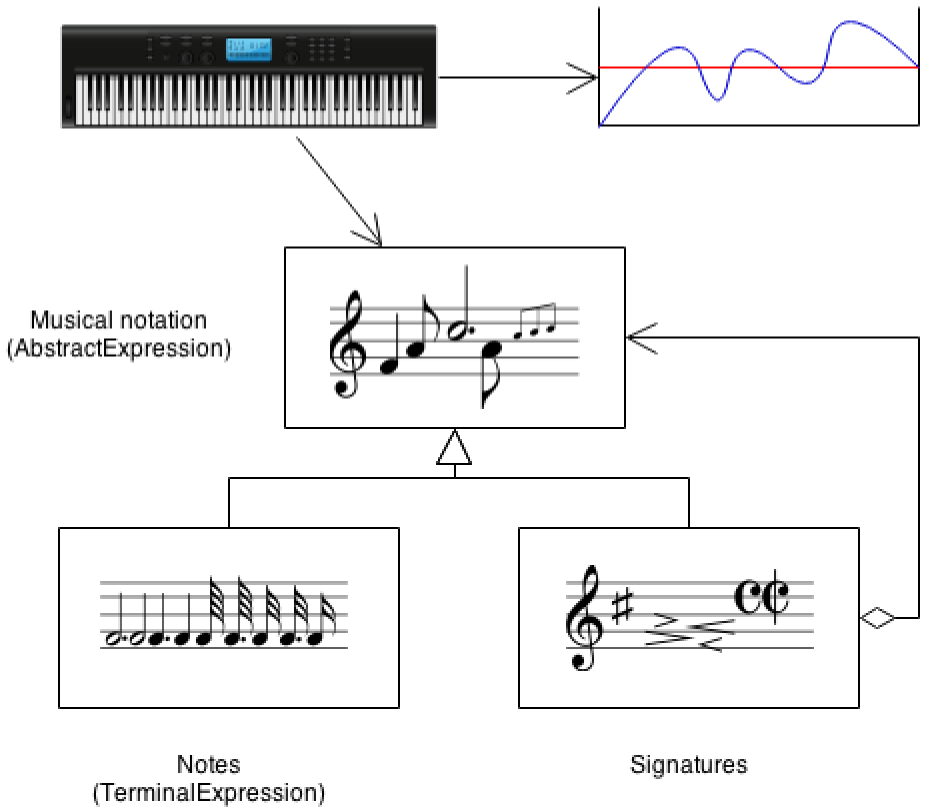
- **Command** is a behavioral design pattern that turns a request into a stand-alone object that contains all information about the request. This transformation lets you parameterize methods with different requests, delay or queue a request's execution, and support undoable operations.



Making an order in a restaurant.

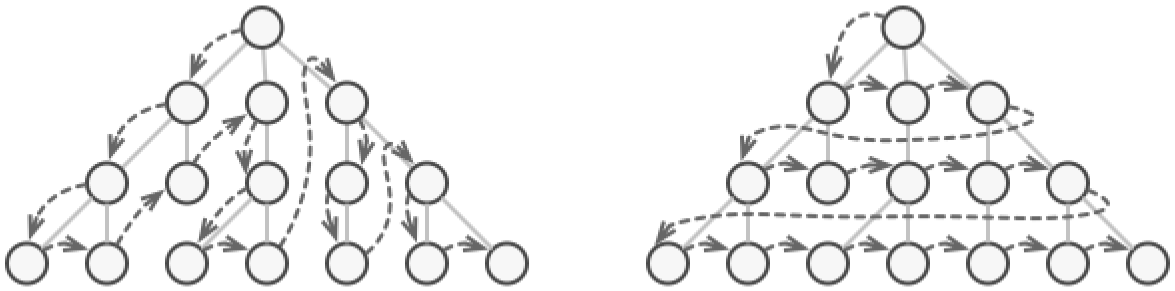
INTERPRETER PATTERN

- The Interpreter pattern defines a grammatical representation for a language and an interpreter to interpret the grammar.
- Musicians are examples of Interpreters. The pitch of a sound and its duration can be represented in musical notation on a staff.



ITERATOR PATTERN

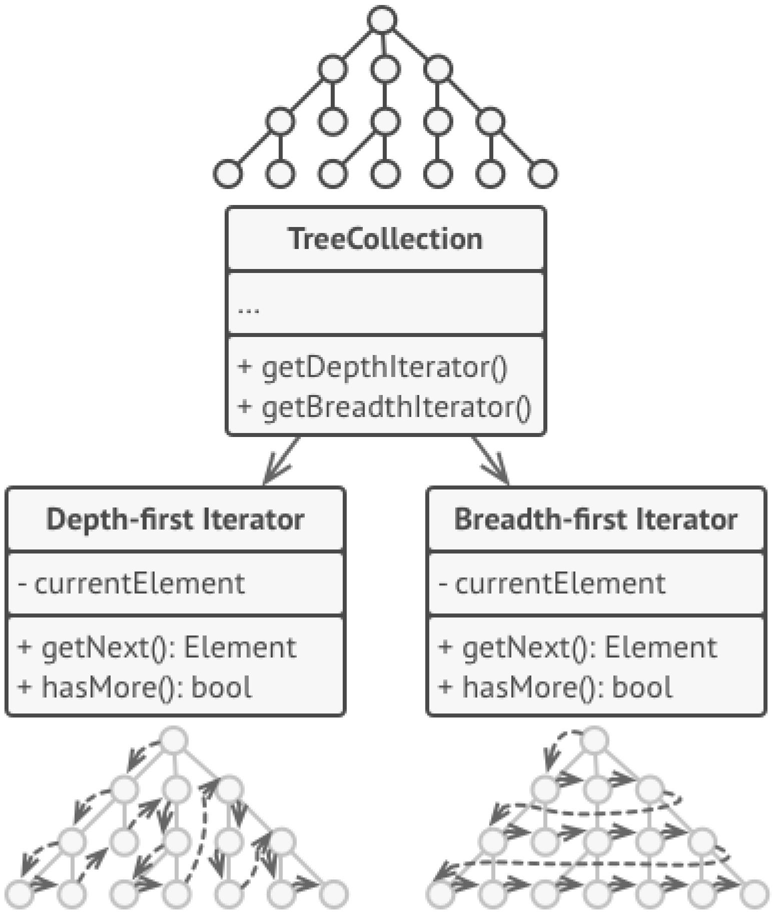
- **Iterator** is a behavioral design pattern that lets you traverse elements of a collection without exposing its underlying representation (list, stack, tree, etc.).



The same collection can be traversed in several different ways.

ITERATOR PATTERN

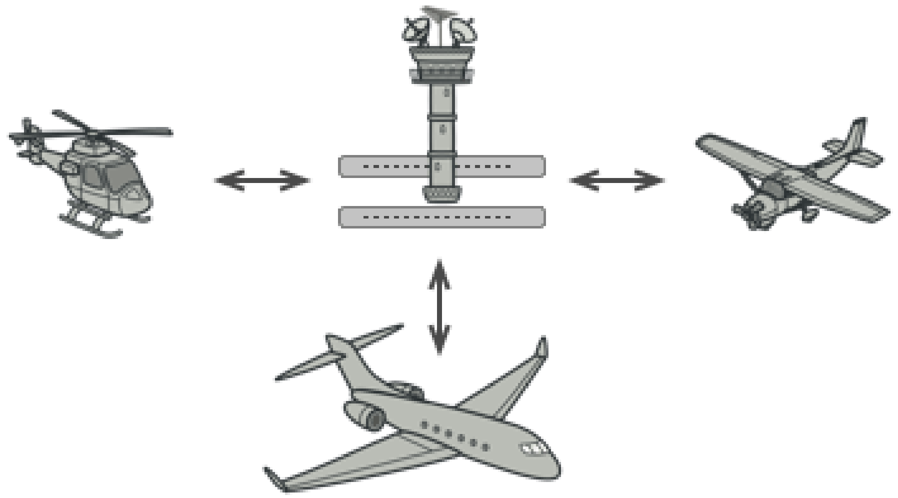
- The main idea of the Iterator pattern is to extract the traversal behavior of a collection into a separate object called an iterator.



Iterators implement various traversal algorithms. Several iterator objects can traverse the same collection at the same time.

MEDIATOR PATTERN

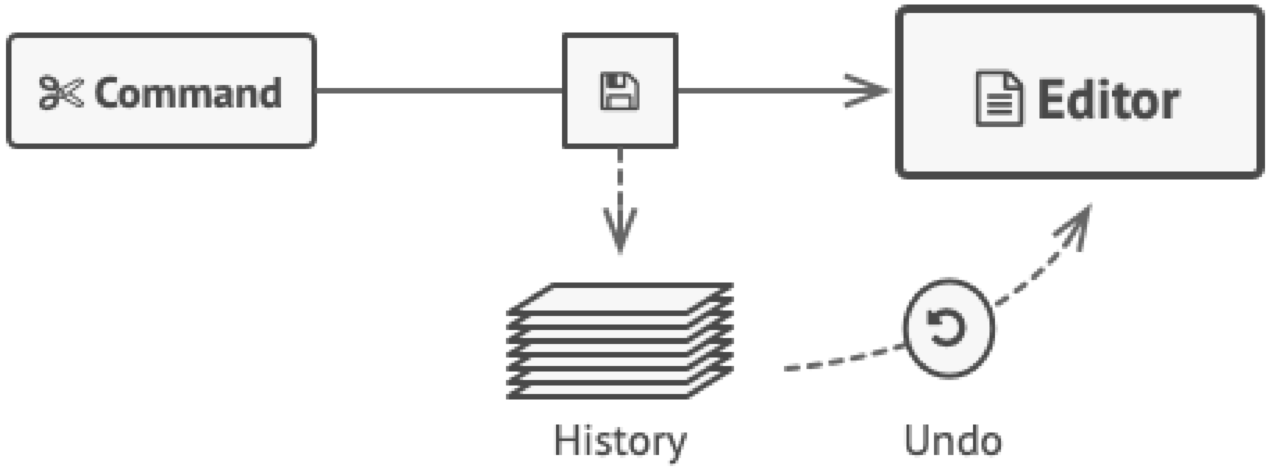
- **Mediator** is a behavioral design pattern that lets you reduce chaotic dependencies between objects. The pattern restricts direct communications between the objects and forces them to collaborate only via a mediator object.



Aircraft pilots don’t talk to each other directly when deciding who gets to land their plane next. All communication goes through the control tower.

MEMENTO PATTERN

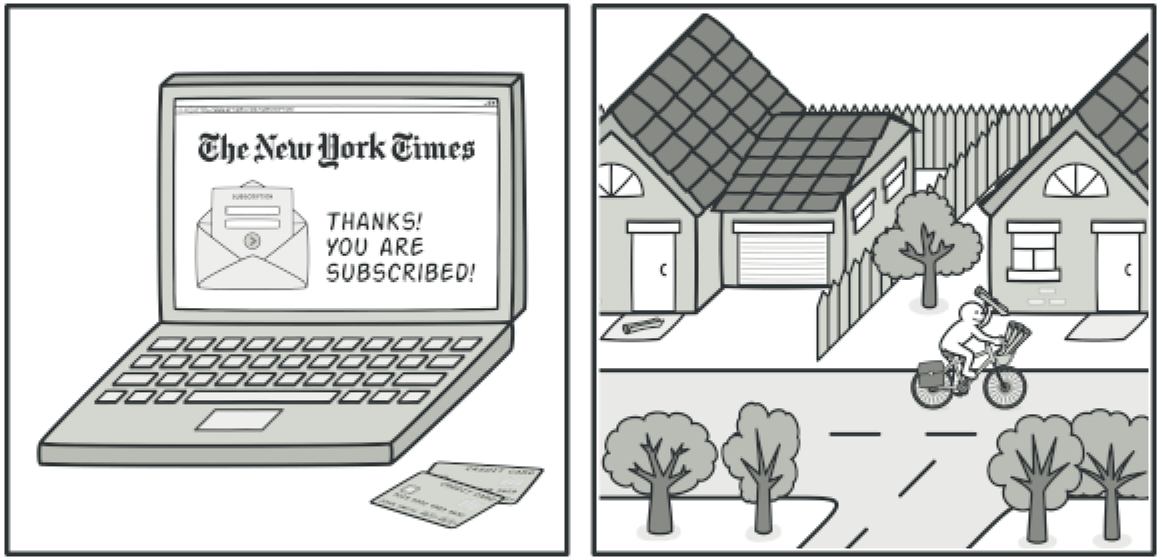
- **Memento** is a behavioral design pattern that lets you save and restore the previous state of an object without revealing the details of its implementation.



Before executing an operation, the app saves a snapshot of the objects' state, which can later be used to restore objects to their previous state.

OBSERVER PATTERN

- **Observer** is a behavioral design pattern that lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing.



Magazine and newspaper subscriptions.

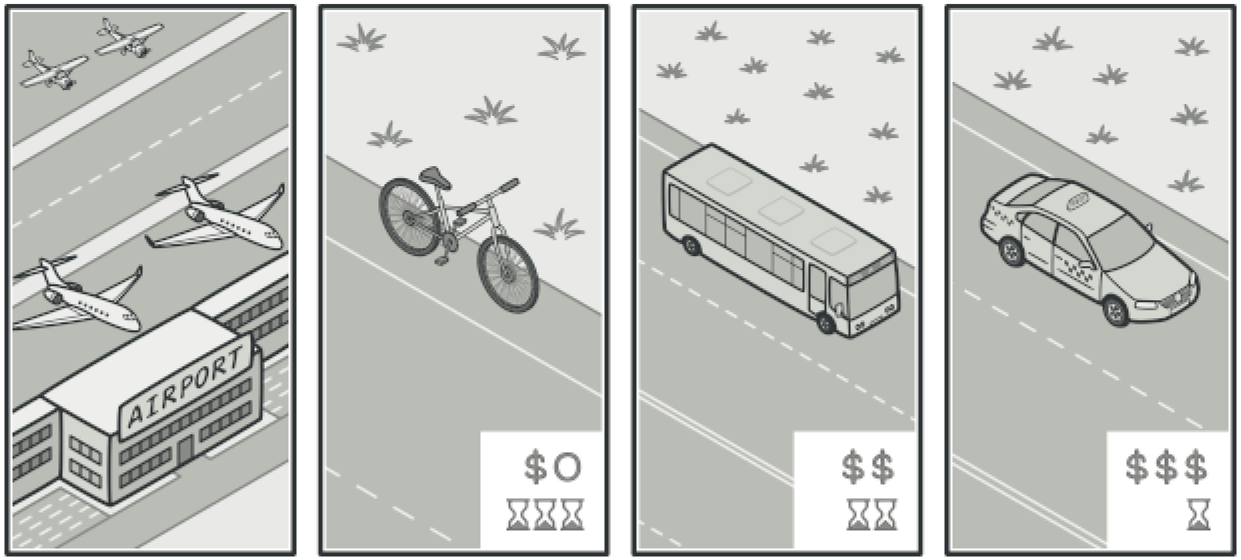
STATE PATTERN

- **State** is a behavioral design pattern that lets an object alter its behavior when its internal state changes. It appears as if the object changed its class.

- The buttons and switches in your smartphone behave differently depending on the current state of the device:
    - When the phone is **unlocked**, pressing buttons leads to executing various functions.
    - When the phone is **locked**, pressing any button leads to the unlock screen.
    - When the phone's **charge** is low, pressing any button shows the charging screen.

STRATEGY PATTERN

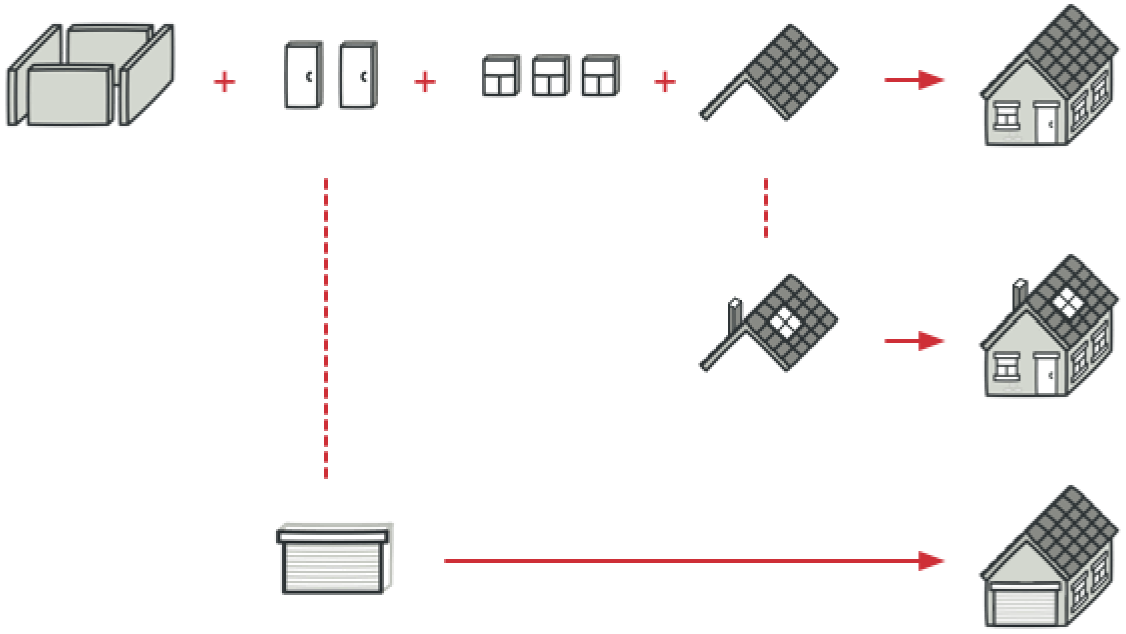
- **Strategy** is a behavioral design pattern that lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.



Various strategies for getting to the airport.

TEMPLATE PATTERN

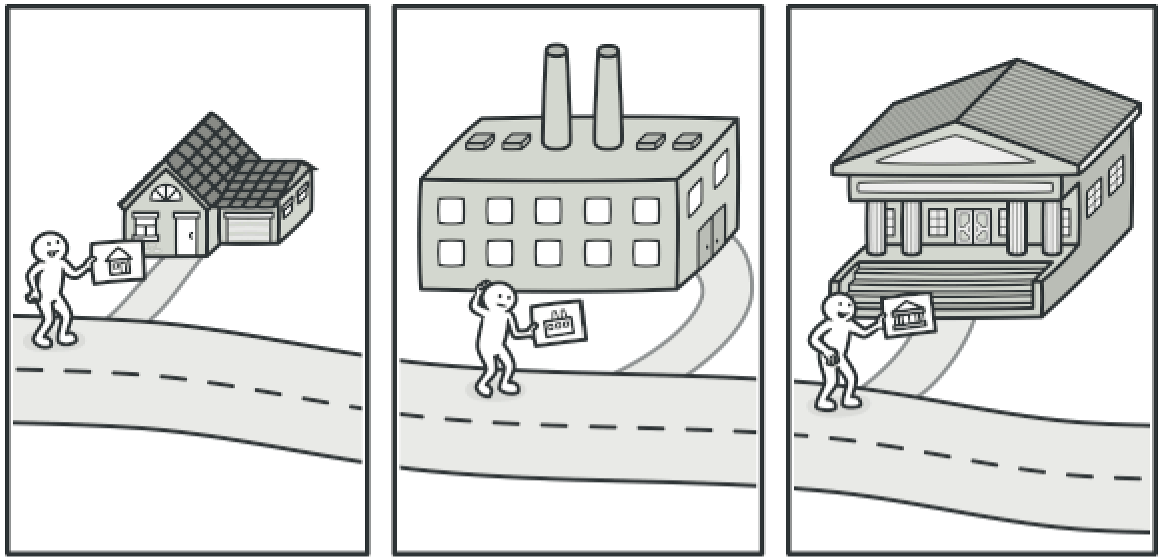
- **Template Method** is a behavioral design pattern that defines the **skeleton** of an algorithm in the superclass but lets subclasses override specific steps of the algorithm without changing its structure.



A typical architectural plan can be slightly altered to better fit the client's needs.

VISITOR PATTERN

- **Visitor** is a behavioral design pattern that lets you separate algorithms from the objects on which they operate.



A good insurance agent is always ready to offer different policies to various types of organizations.

Q & A
THANK YOU