Design Patterns

CSCI 420: Software Engineering

Overview

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

Aside: Criticisms

- **Targets the wrong problem**

  Peter Norvig demonstrated that 16 out of the 23 patterns in the Design Patterns book are simplified/eliminated in Lisp

- **Lacks formal foundations**

  At OOPSLA 1999, the Gang of Four were, *with their full cooperation,* subjected to a show trial, in which they were "charged" with numerous crimes against computer science. They were "convicted" by ⅔ of the "jurors" who attended the trial.

- **Leads to inefficient solutions**

  https://github.com/EnterpriseQualityCoding/FizzBuzzEnterpriseEdition

- **Does not differ significantly from other abstractions**

Categories

- Creational Design Patterns
  - These design patterns are all about class instantiation
  - Composed of class-creation (inheritance) and object-creational (delegation)
- Structural Design Patterns
  - These design patterns are all about Class and Object composition.
  - Structural class-creation patterns use inheritance to compose interfaces.
- Behavioral Design Patterns
  - All about Class's objects communication.

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
- **Object Pool**
  Avoid expensive acquisition and release of resources by recycling objects that are no longer in use
- **Prototype**
  A fully initialized instance to be copied or cloned
- **Singleton** *
  A class of which only a single instance can exist

Structural Patterns (1/2)

- **Adapter** *
  Match interfaces of different classes
- **Bridge**
  Separates an object's interface from its implementation
- **Composite**
  A tree structure of simple and composite objects
- **Decorator** * **(Python)**
  Add responsibilities to objects dynamically

Structural Patterns (2/2)

- **Flyweight**
  A fine-grained instance used for efficient sharing
- **Facade**
  A single class that represents an entire subsystem
- **Private Class Data**
  Restricts accessor/mutator access
- **Proxy**
  An object representing another object

Behavioral Patterns (1/2)

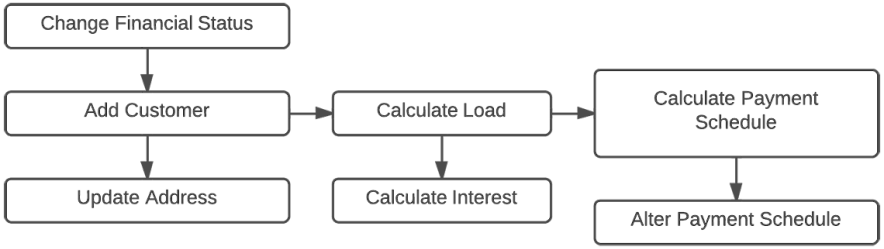
- **Chain of responsibility**
  A way of passing a request between a chain of objects
- **Command** *
  Encapsulate a command request as an object
- **Interpreter**
  A way to include language elements in a program
- **Iterator** *
  Sequentially access the elements of a collection
- **Mediator**
  Defines simplified communication between classes
- **Memento** *
  Capture and restore an object's internal state

Behavioral Patterns (2/2)

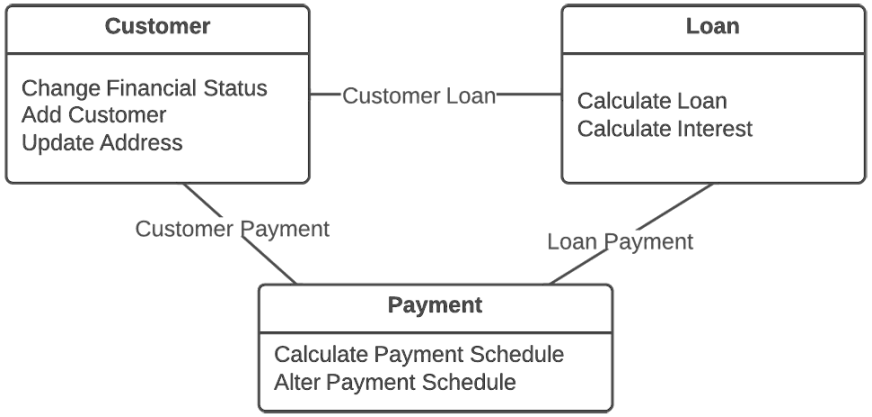
- **Null Object**
  Designed to act as a default value of an object
- **Observer** *
  A way of notifying change to a number of classes
- **State** *
  Alter an object's behavior when its state changes
- **Strategy** *
  Encapsulates an algorithm inside a class
- **Template method**
  Defer the exact steps of an algorithm to a subclass
- **Visitor** *
  Defines a new operation to a class without change

Examples of When to Apply

Functional Decomposition (bad)



Object-Oriented Programming (good)

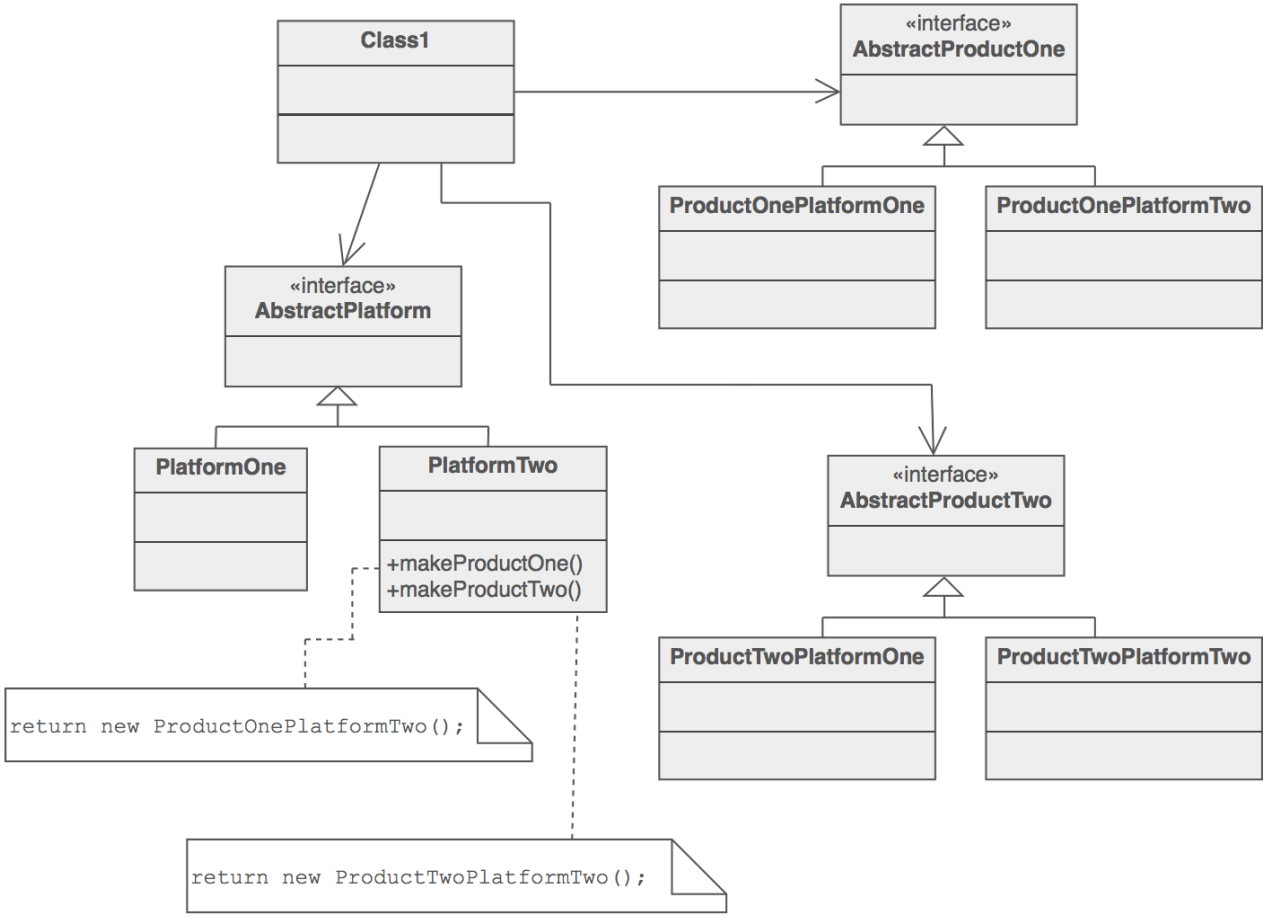


Abstract Factory*

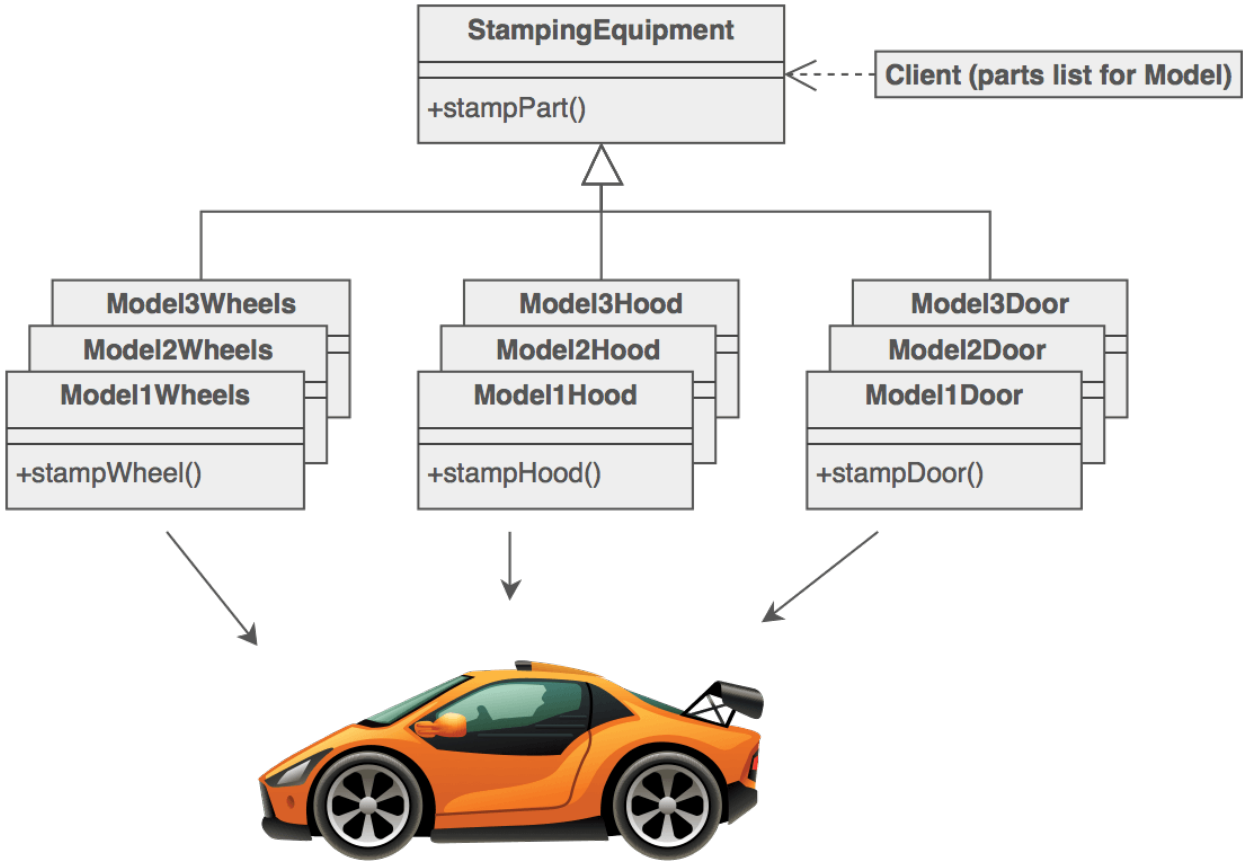
Intent

- Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
- A hierarchy that encapsulates: many possible "platforms", and the construction of a suite of "products".
- The new operator considered harmful.

Abstract Factory



Abstract Factory



Builder*

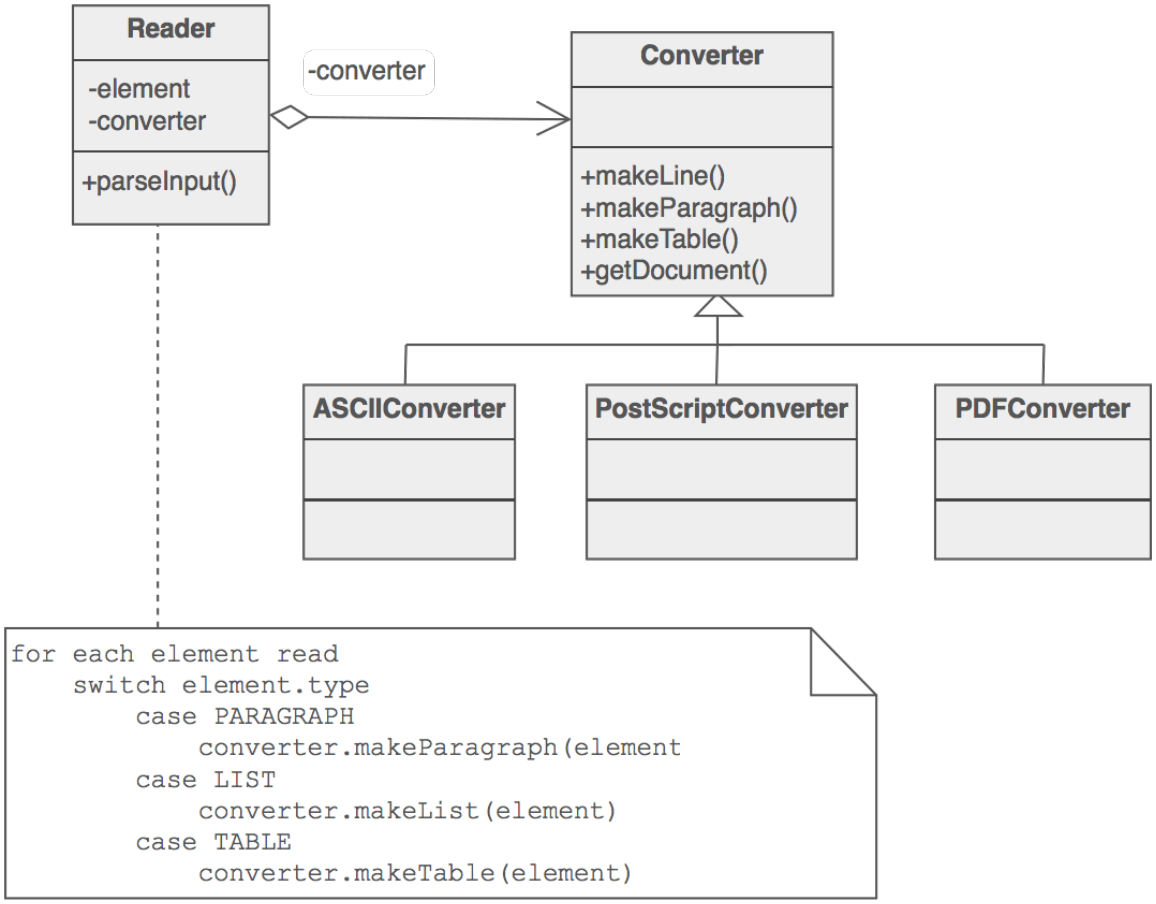
Intent

- 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.

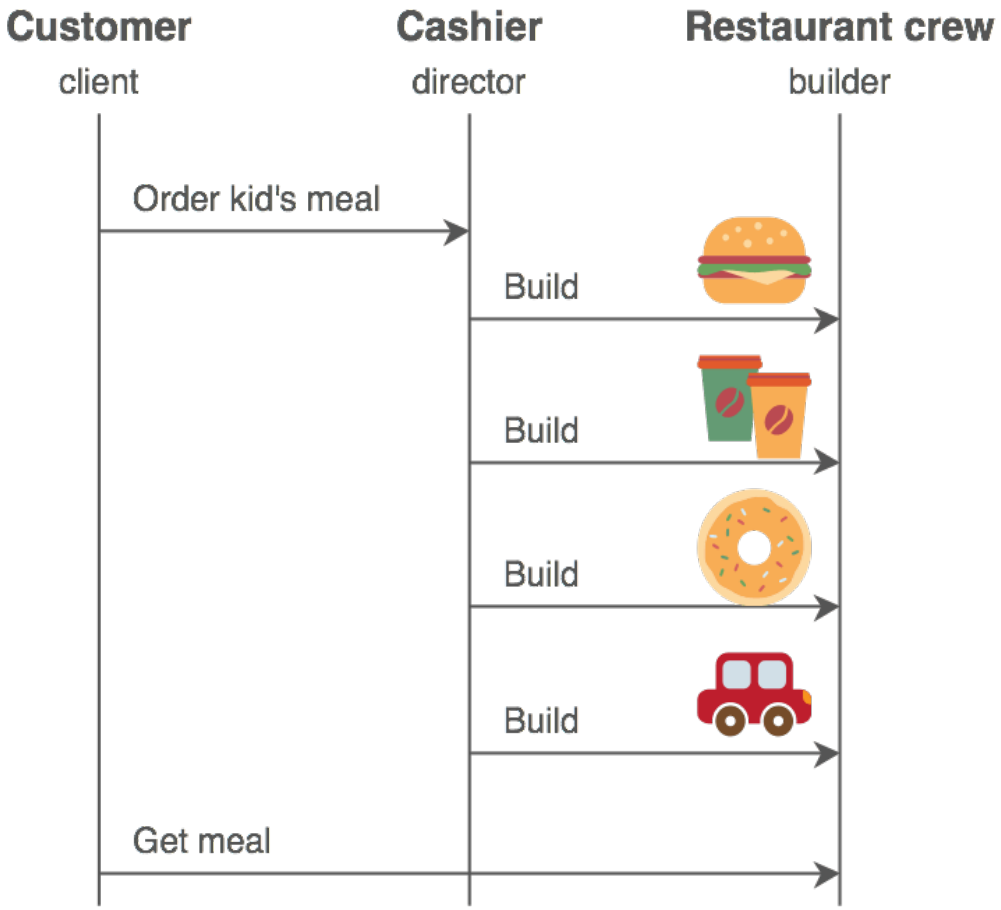
Problem

- 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.

Builder*


Reader
-element
-converter
+parseInput()
-converter
Converter
+makeLine()
+makeParagraph()
+makeTable()
+getDocument()
ASCIIConverter
PostScriptConverter
PDFConverter
for each element read
switch element.type
case PARAGRAPH
converter.makeParagraph(element
case LIST
converter.makeList(element)
case TABLE
converter.makeTable(element)

Builder


Customer
client
Cashier
director
Restaurant crew
builder
Order kid's meal
Build
Build
Build
Build
Get meal

Factory Method

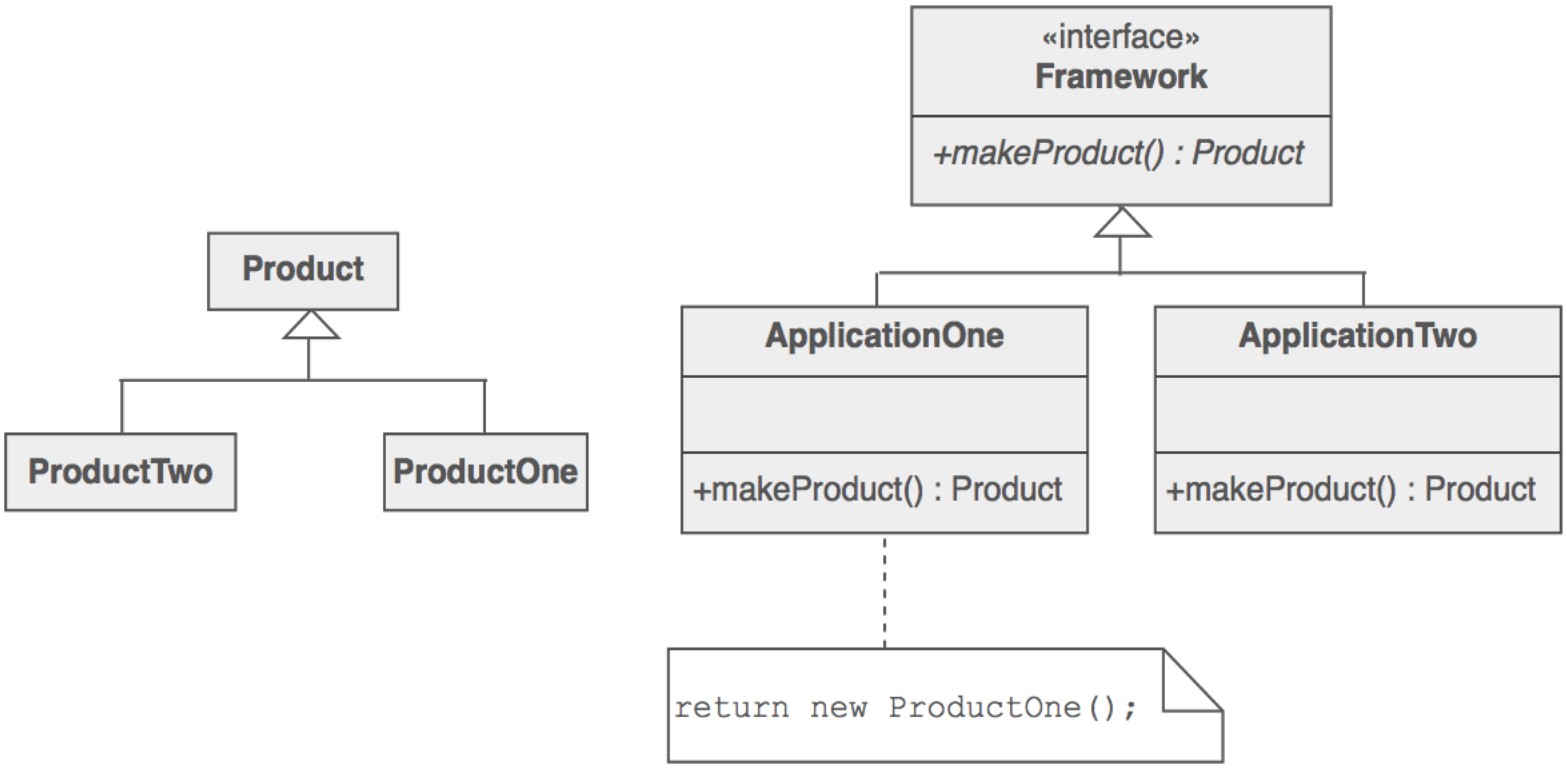
Intent

- Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
- Defining a "virtual" constructor.
- The new operator considered harmful.

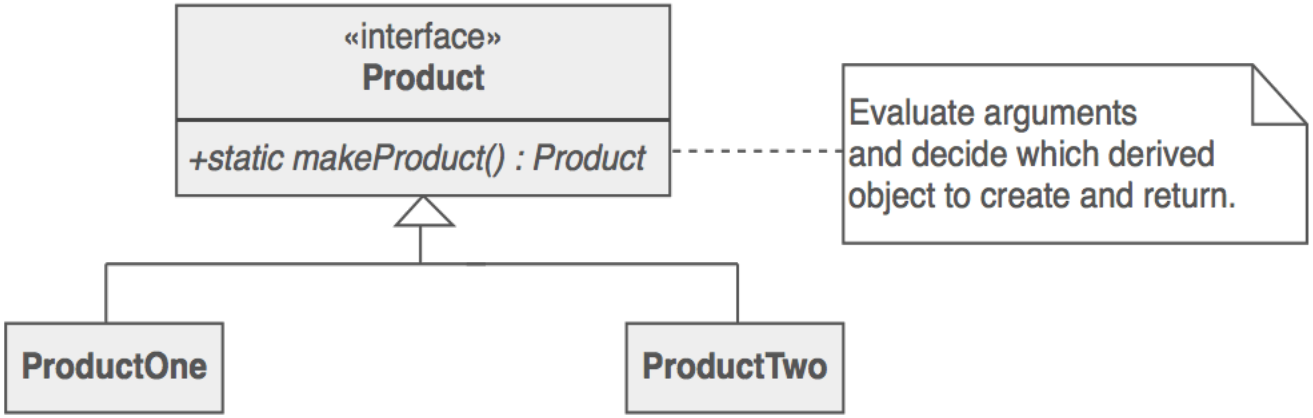
Problem

- 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.

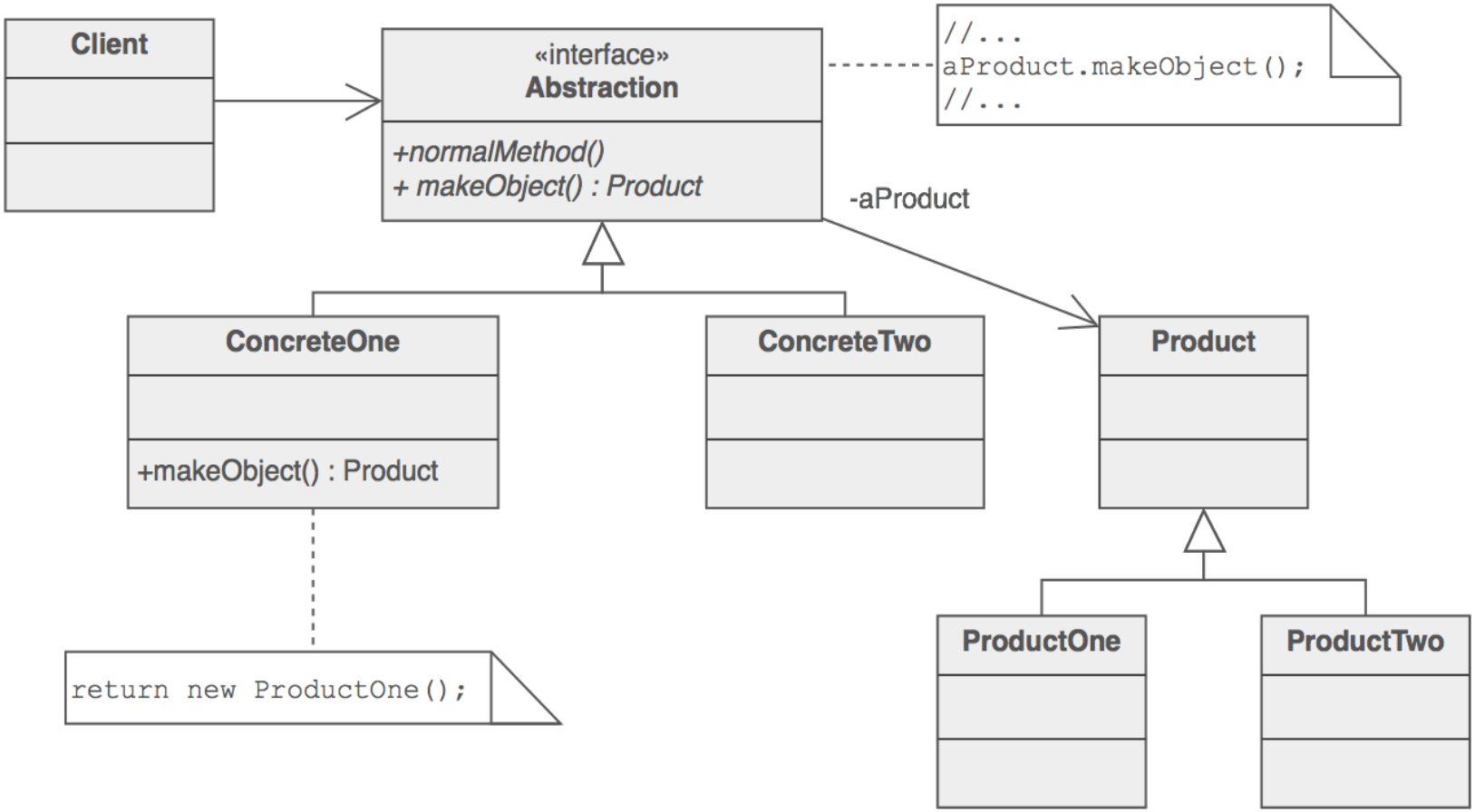
Factory Method



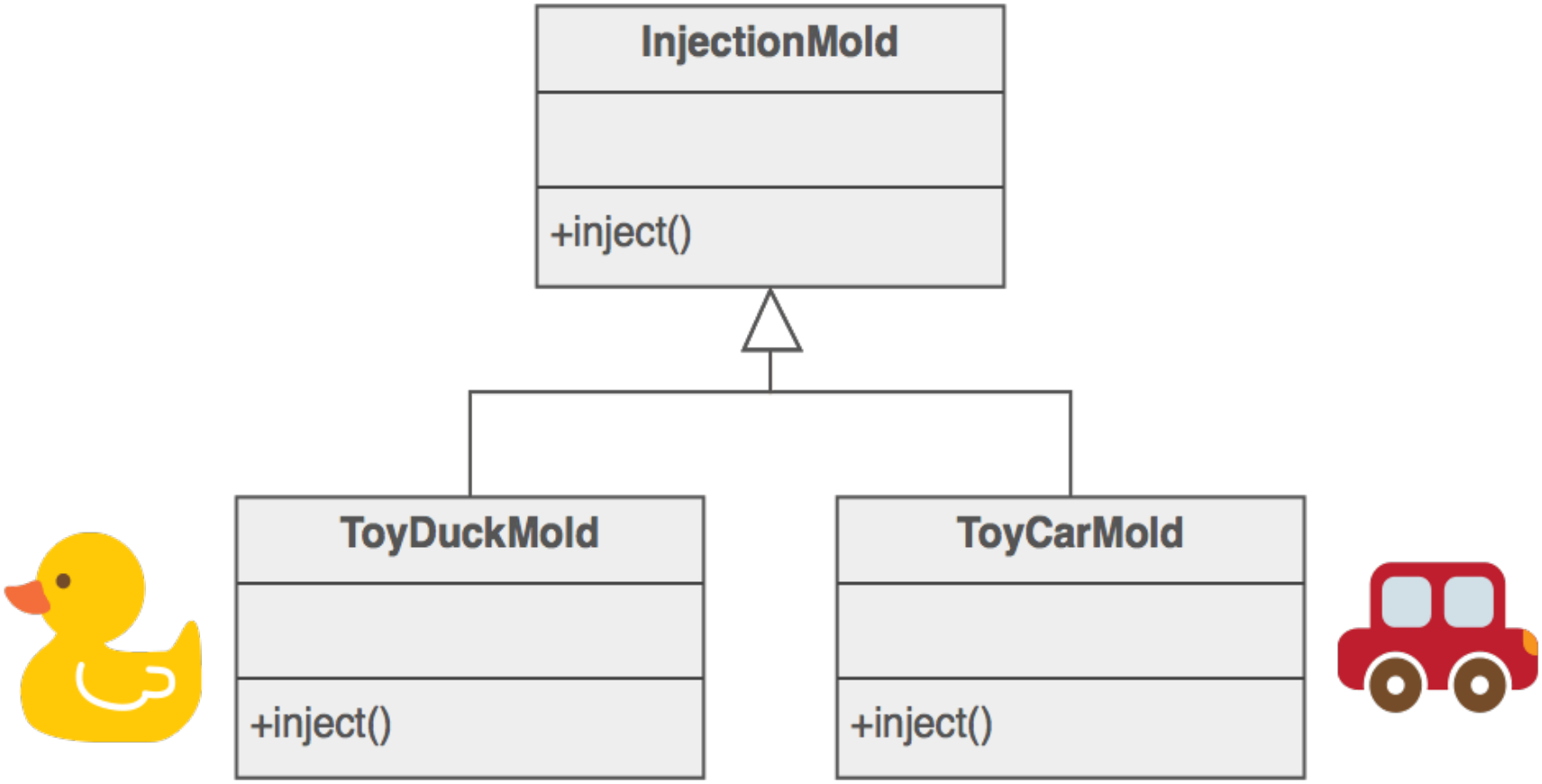
Factory Method (static member)



Factory Method



Factory Method



Object Pool

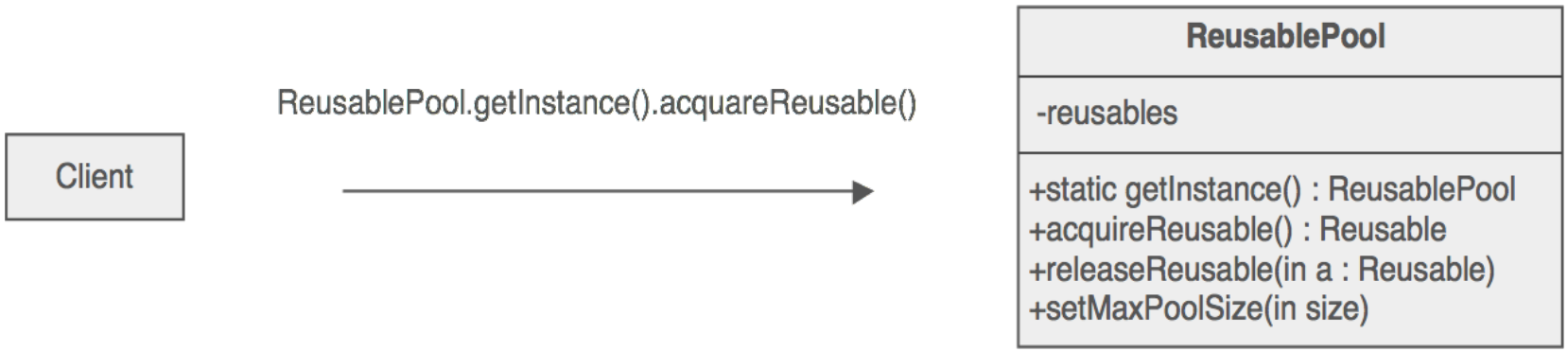
Intent

- Object pooling can offer a significant performance boost
- it is most effective in situations where:
  - the cost of initializing a class instance is high
  - the rate of instantiation of a class is high
  - the number of instantiations in use at any one time is low.

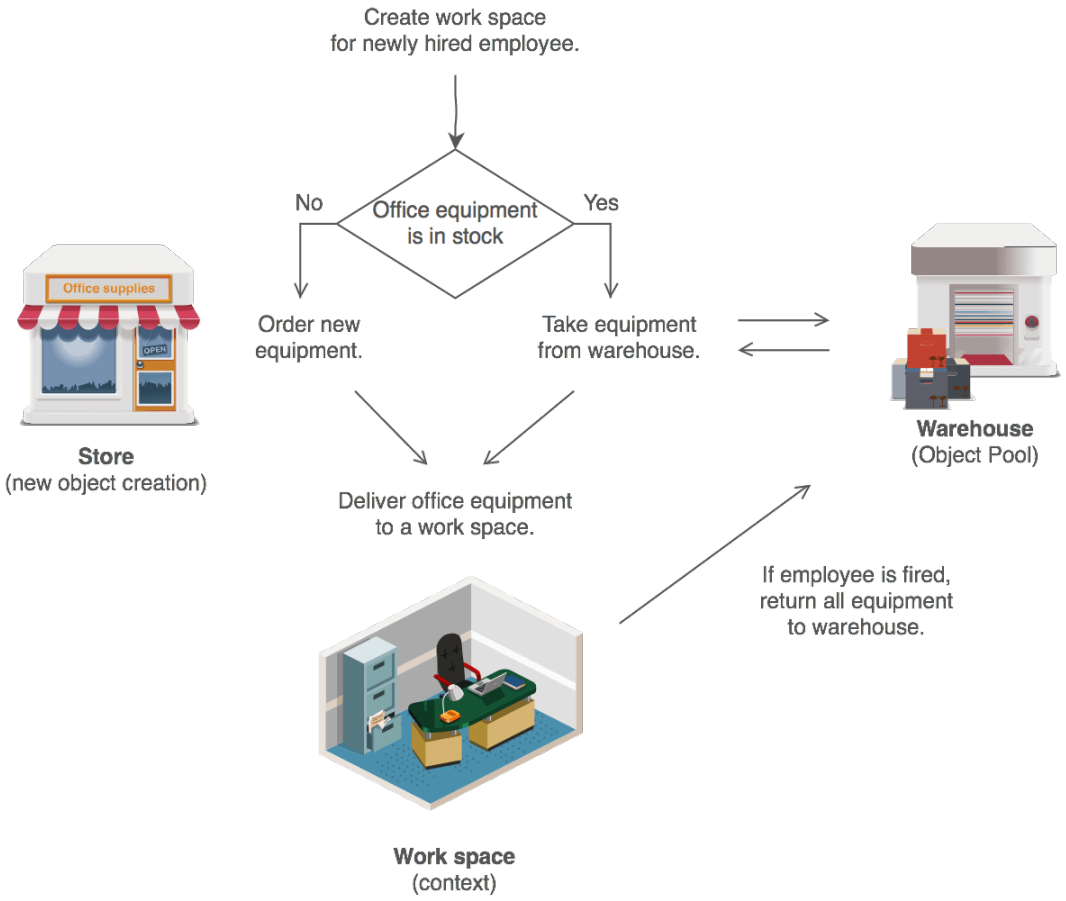
Problem

- It is desirable to keep all Reusable objects that are not currently in use in the same object pool so that they can be managed by one coherent policy.
- To achieve this, the Reusable Pool class is designed to be a singleton class.

Object Pool


Client
ReusablePool.getInstance().acquareReusable()
ReusablePool
-reusables
+static getInstance() : ReusablePool
+acquireReusable() : Reusable
+releaseReusable(in a : Reusable)
+setMaxPoolSize(in size)

Object Pool


Create work space
for newly hired employee.
Office equipment
is in stock
No
Yes
Order new
equipment.
Take equipment
from warehouse.
Office supplies
OPEN
Store
(new object creation)
Warehouse
(Object Pool)
Deliver office equipment
to a work space.
If employee is fired,
return all equipment
to warehouse.
Work space
(context)

Prototype

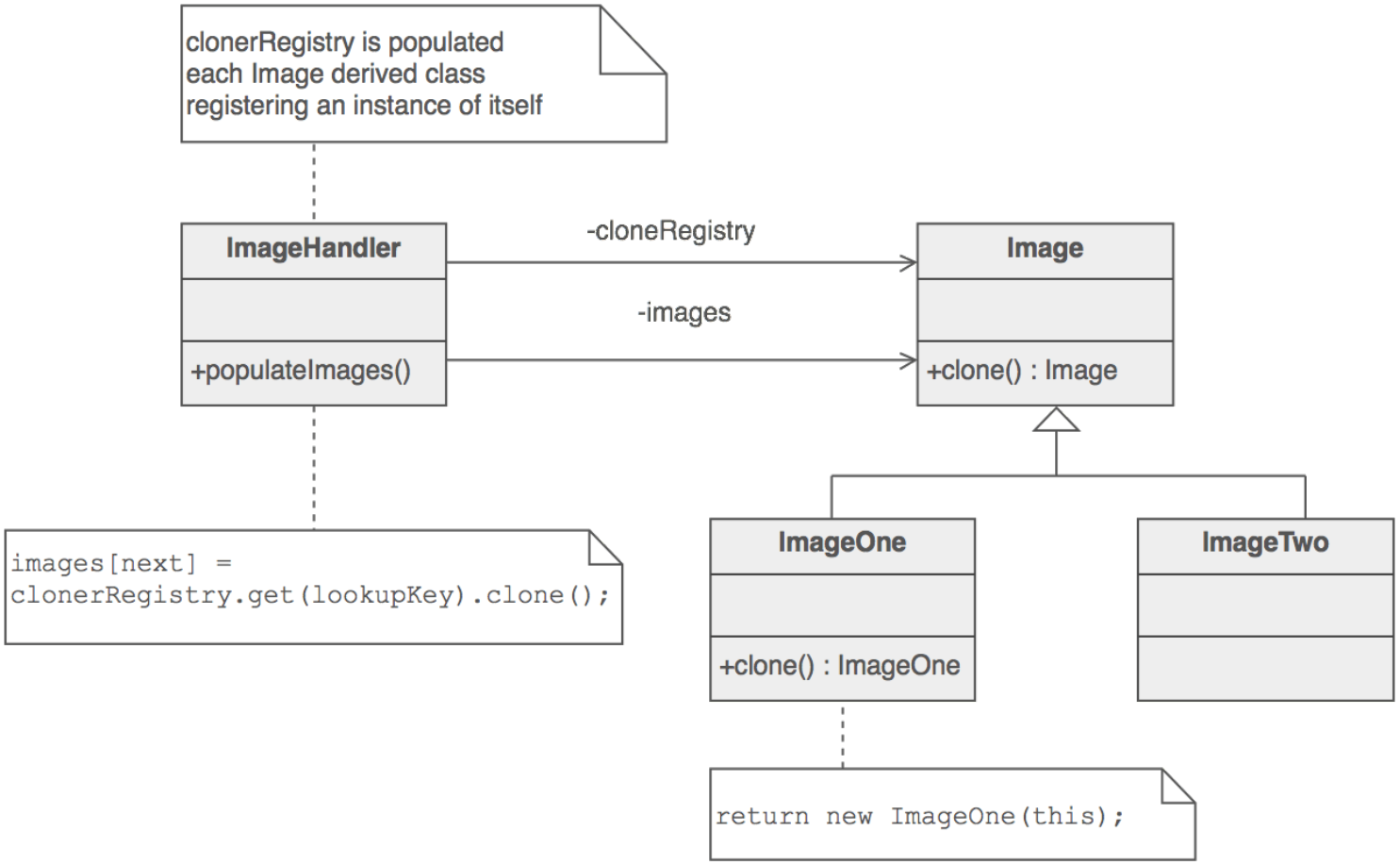
Intent

- Specify the kinds of objects to create using a prototypical instance and create new objects by copying this prototype.
- Co-opt one instance of a class for use as a breeder of all future instances.
- The new operator considered harmful.

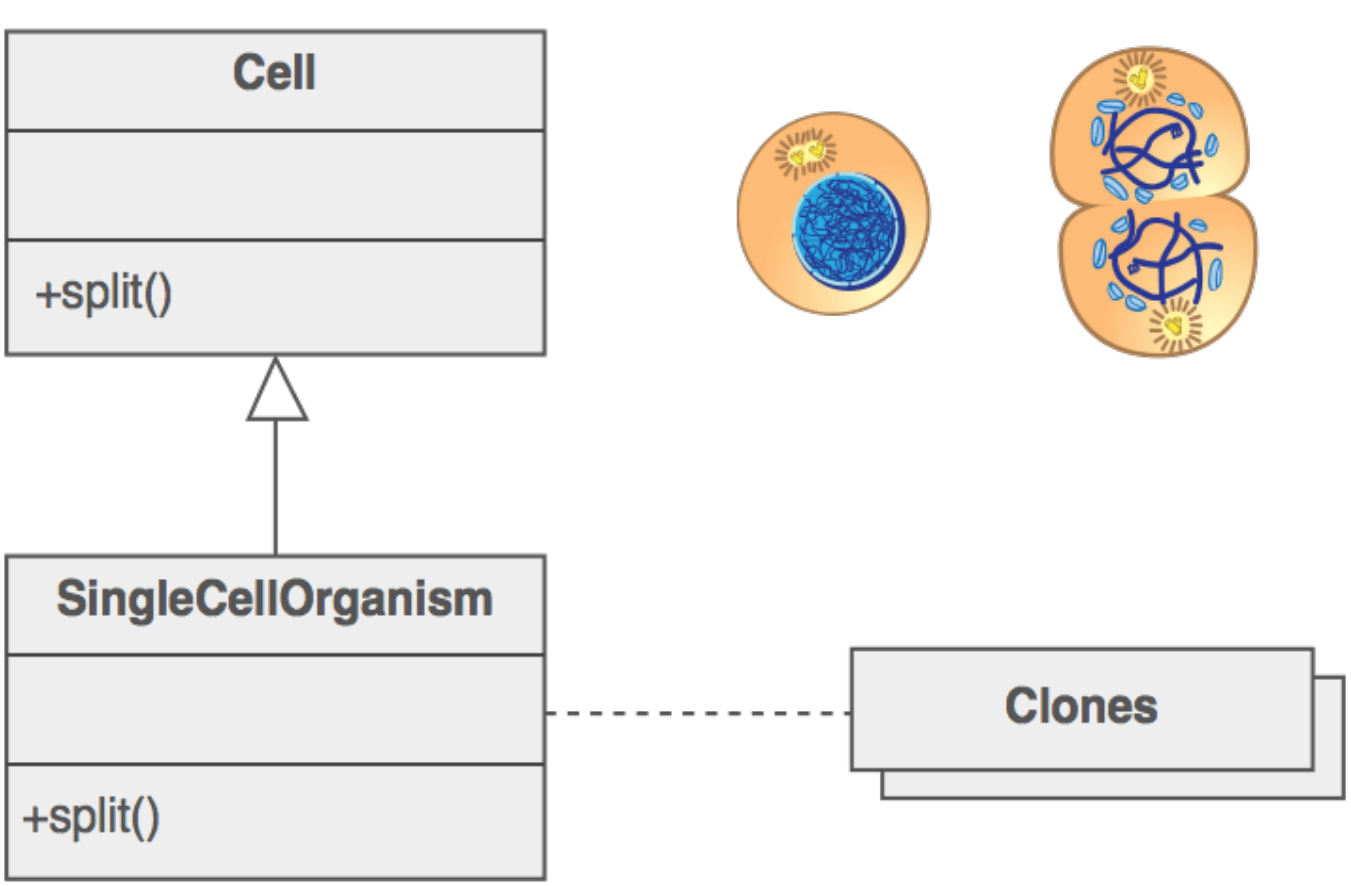
Problem

- Application "hard wires" the class of object to create in each "new" expression.

Prototype


clonerRegistry is populated
each Image derived class
registering an instance of itself
ImageHandler
+populateImages()
-cloneRegistry
-images
Image
+clone() : Image
images[next] =
clonerRegistry.get(lookupKey).clone();
ImageOne
+clone() : ImageOne
ImageTwo
return new ImageOne(this);

Prototype



Singleton*

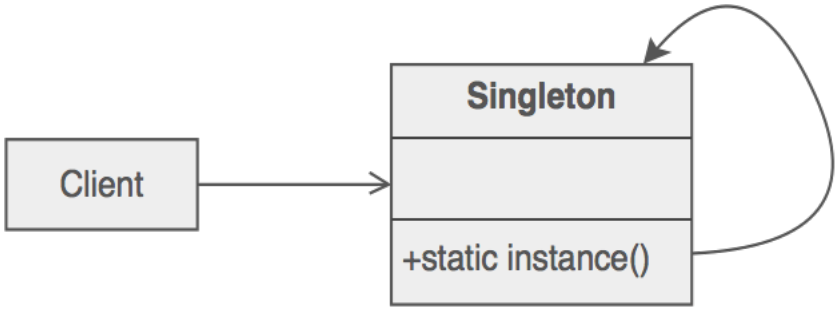
Intent

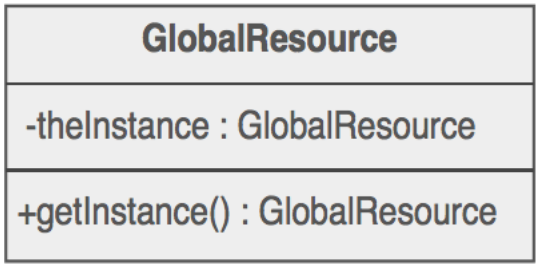
- Ensure a class has only one instance and provide a global point of access to it.
- Encapsulated "just-in-time initialization" or "initialization on first use".

Problem

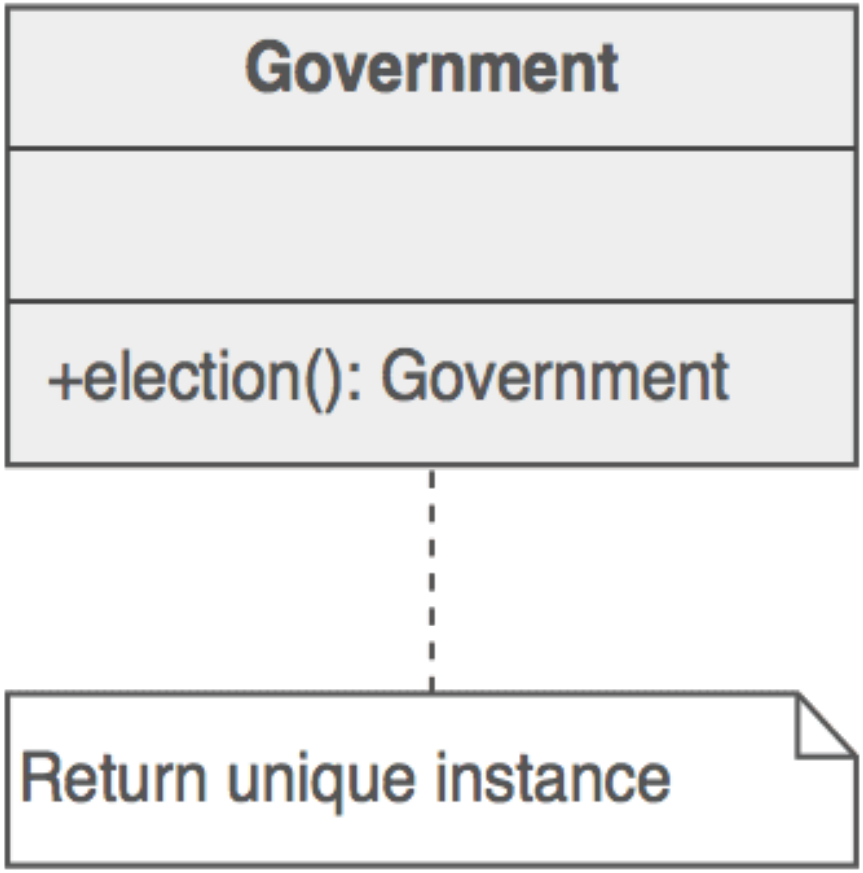
- Application needs one, and only one, instance of an object. Additionally, lazy initialization and global access are necessary.

Singleton*


Client
Singleton
+static instance()


GlobalResource
-theInstance : GlobalResource
+getInstance() : GlobalResource

Singleton



Adapter*

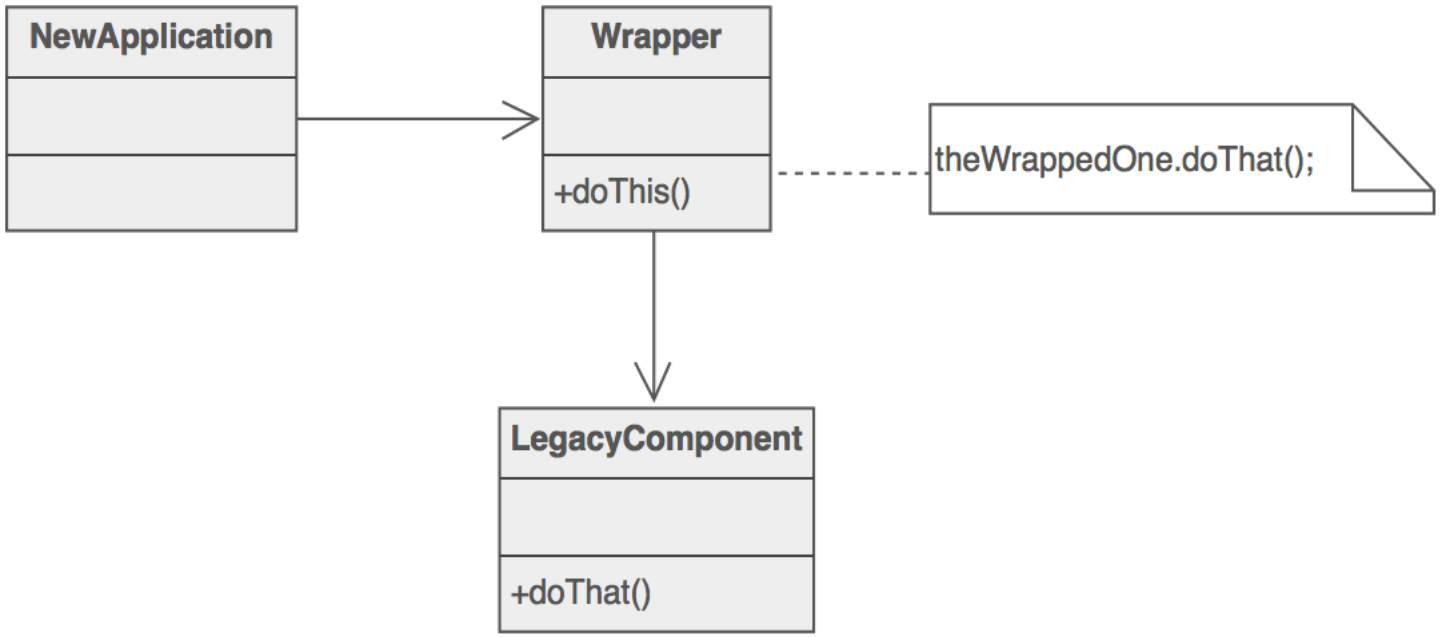
Intent

- 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

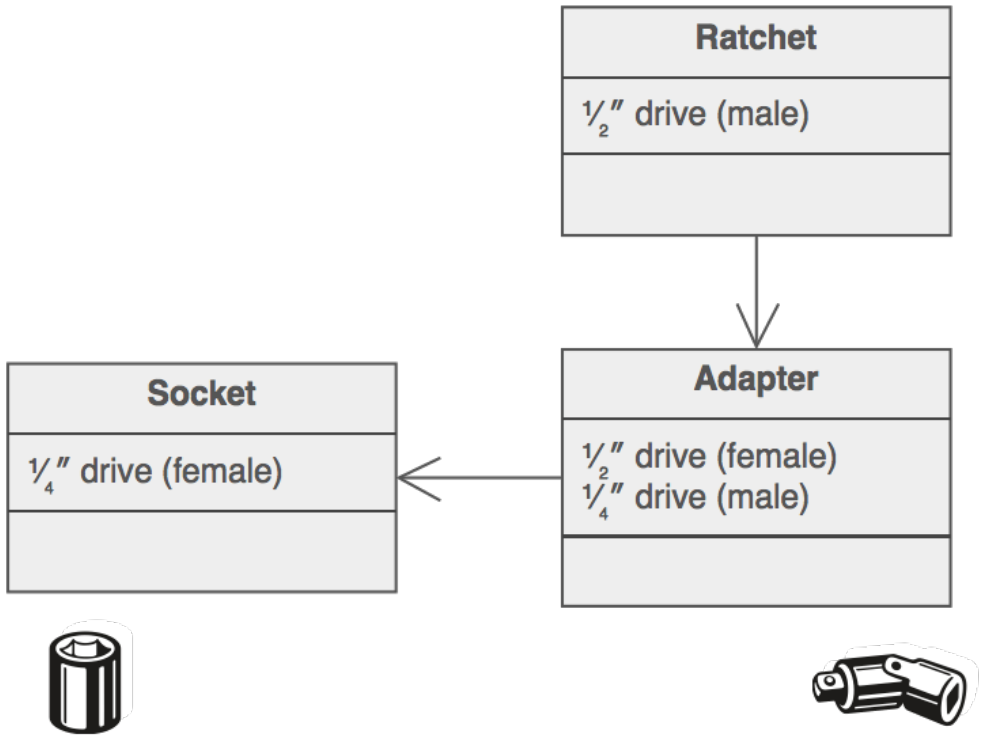
Problem

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.

Adapter*


NewApplication
Wrapper
+doThis()
theWrappedOne.doThat();
LegacyComponent
+doThat()

Adapter


Ratchet
1/2" drive (male)
Socket
1/4" drive (female)
Adapter
1/2" drive (female)
1/4" drive (male)

Bridge

Intent

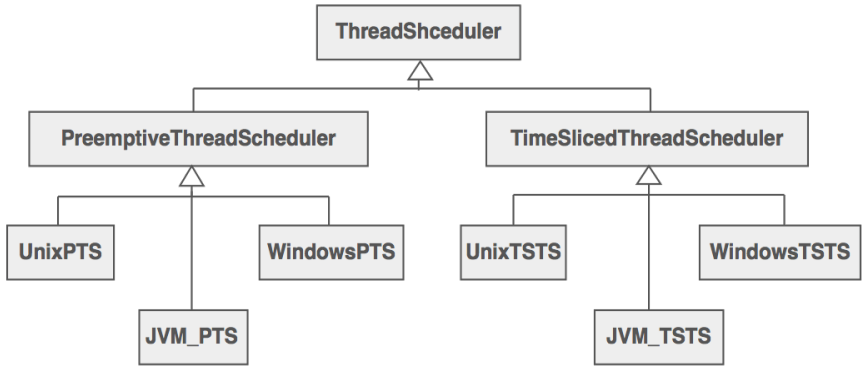
- Decouple an abstraction from its implementation so that the two can vary independently.
- Publish interface in an inheritance hierarchy, and bury implementation in its own inheritance hierarchy.
- Beyond encapsulation, to insulation

Problem

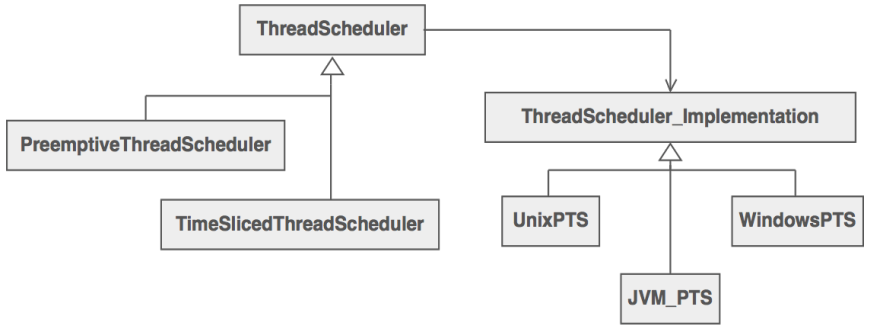
- "Hardening of the software arteries" has occurred by using subclassing of an abstract base class to provide alternative implementations. This locks in compile-time binding between interface and implementation. The abstraction and implementation cannot be independently extended or composed.

Bridge

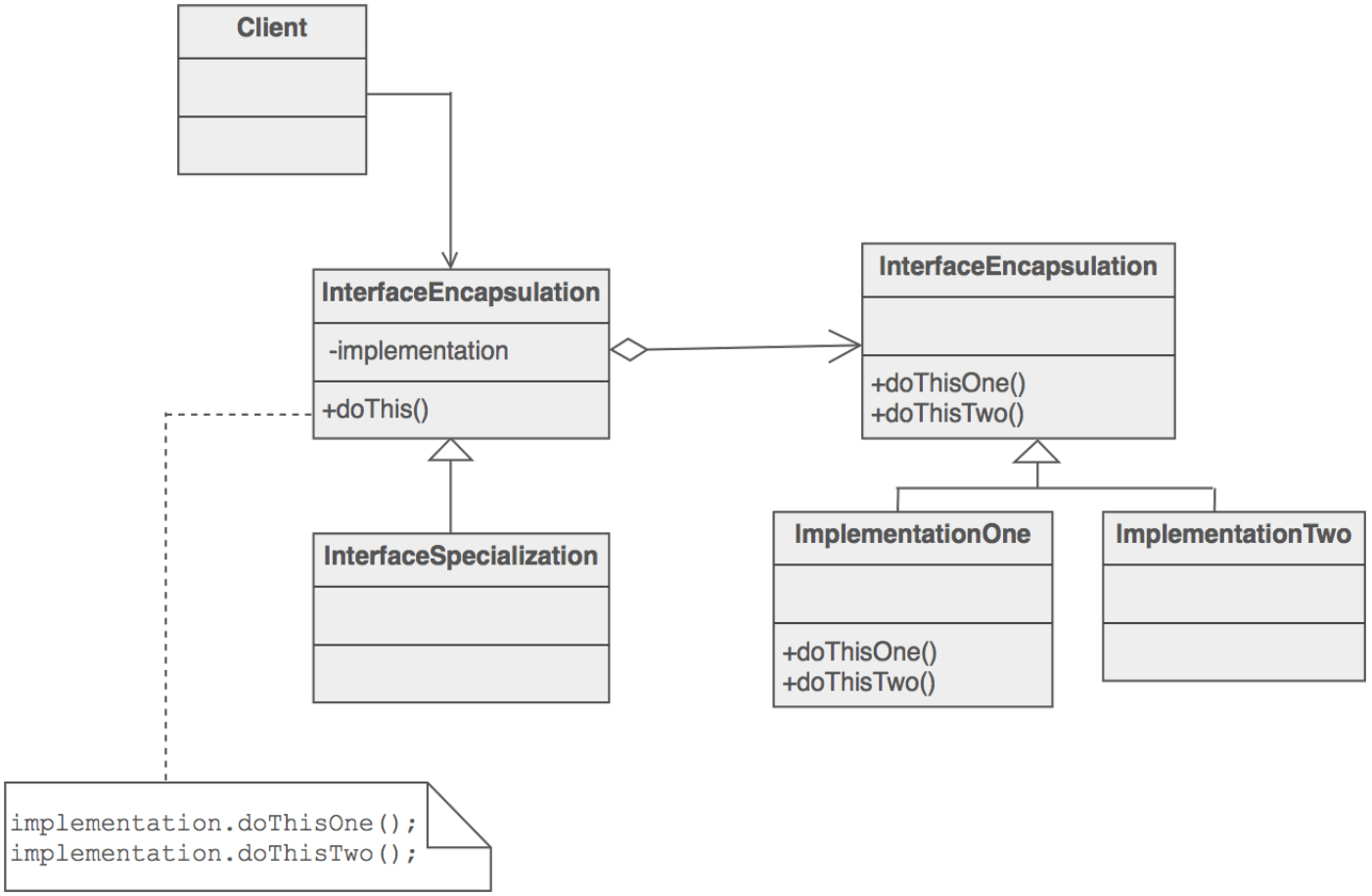
Before



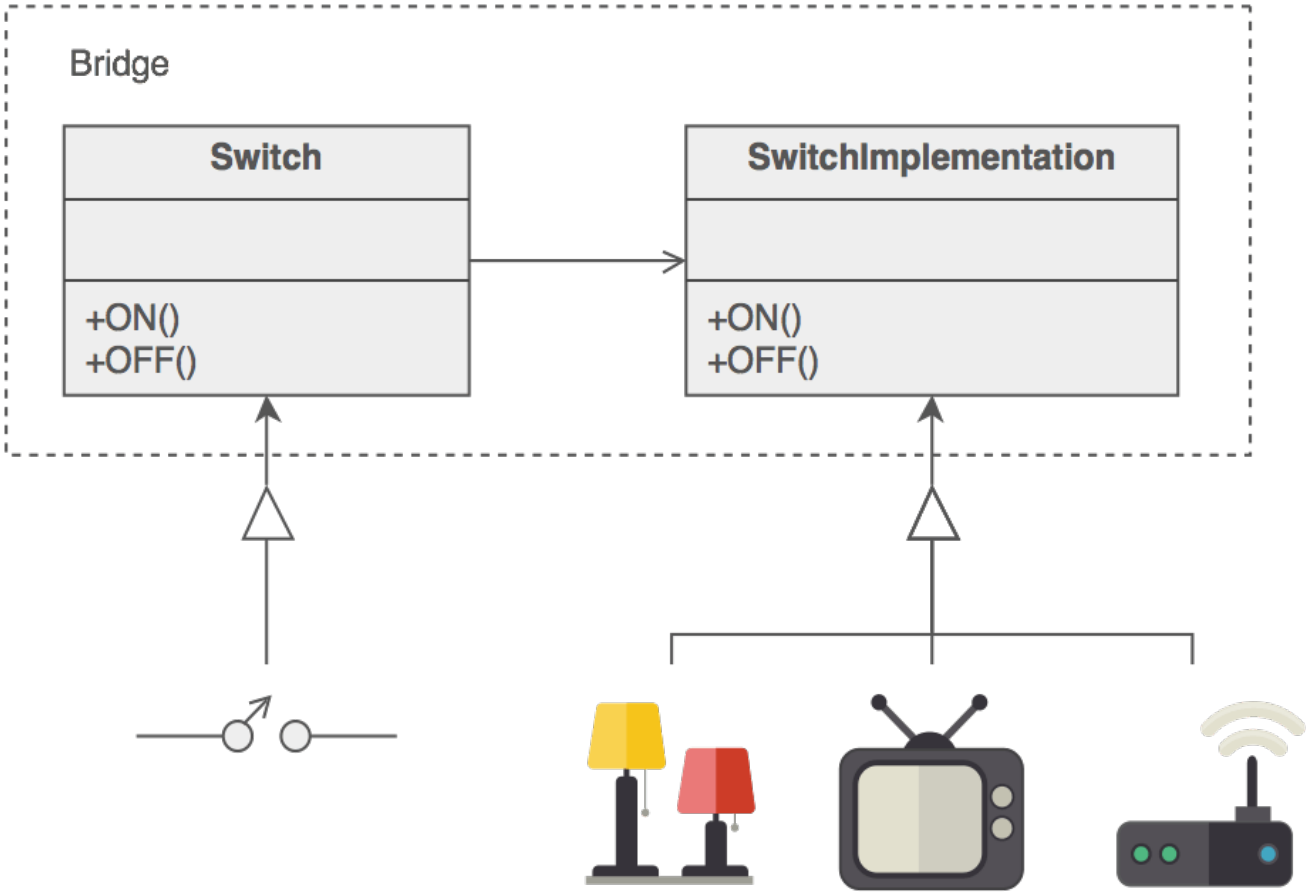
After



Bridge


Client
InterfaceEncapsulation
-implementation
+doThis()
InterfaceEncapsulation
+doThisOne()
+doThisTwo()
InterfaceSpecialization
ImplementationOne
+doThisOne()
+doThisTwo()
ImplementationTwo
implementation.doThisOne();
implementation.doThisTwo();

Bridge



Composite

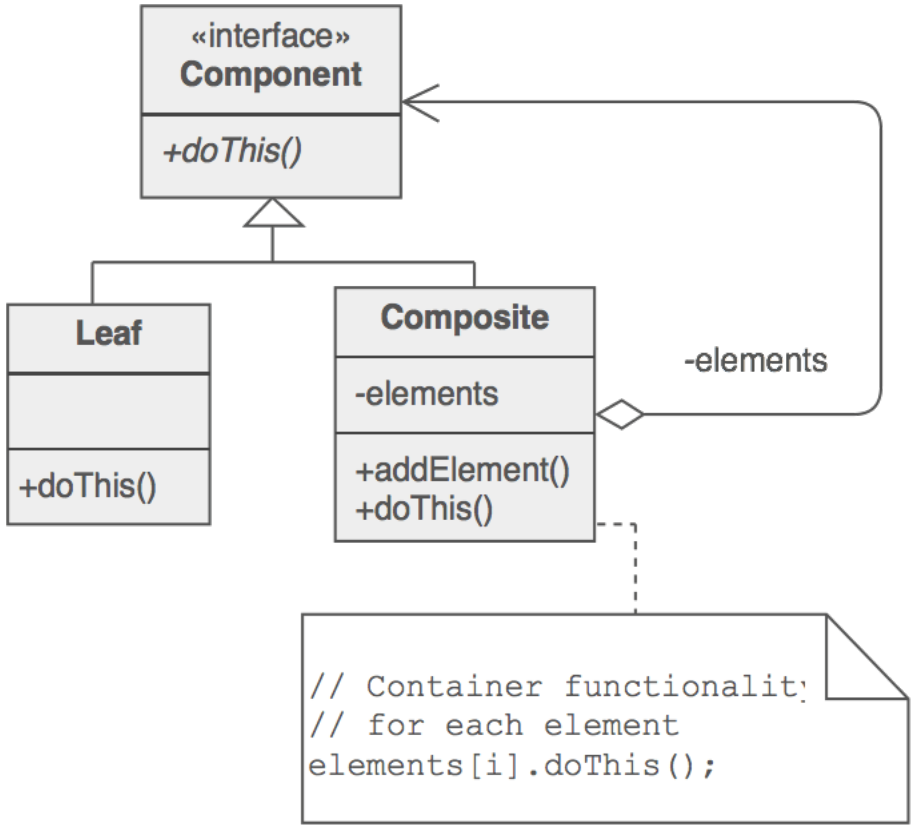
Intent

- Compose objects into tree structures. Composite lets clients treat individual objects and compositions of objects uniformly.
- Recursive composition
- "Directories contain entries, each of which could be a directory."
- 1-to-many "has a" up the "is a" hierarchy

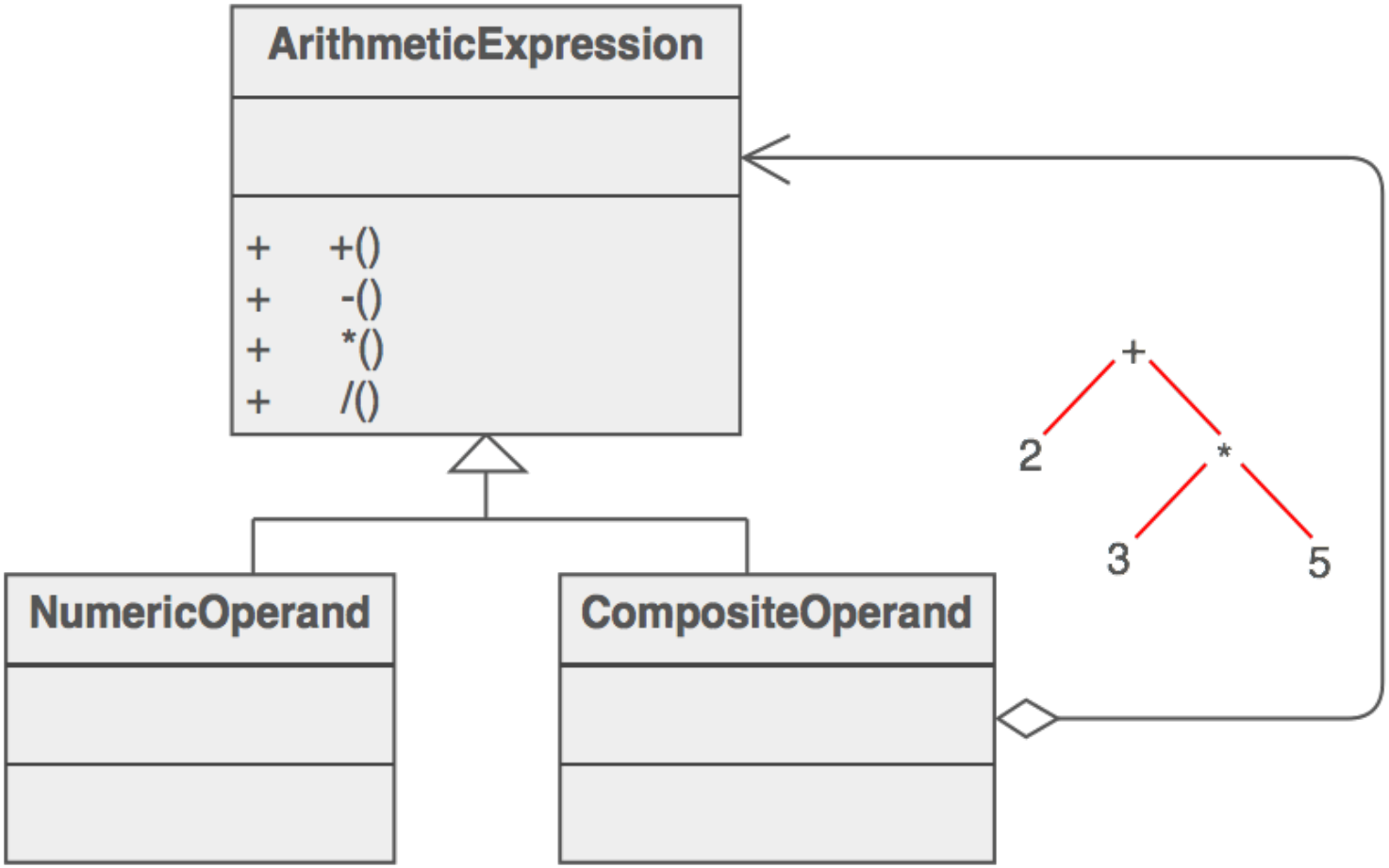
Problem

- Application needs to manipulate a hierarchical collection of "primitive" and "composite" objects. Processing of a primitive object is handled one way, and processing of a composite object is handled differently. Having to query the "type" of each object before attempting to process it is not desirable.

Composite


«interface»
Component
+doThis()
Leaf
+doThis()
Composite
-elements
+addElement()
+doThis()
-elements
// Container functionalit
// for each element
elements[i].doThis();

Composite


ArithmeticExpression
+ +()
+ -()
+ *()
+ /()
NumericOperand
CompositeOperand
+
2
*
3
5

Decorator*

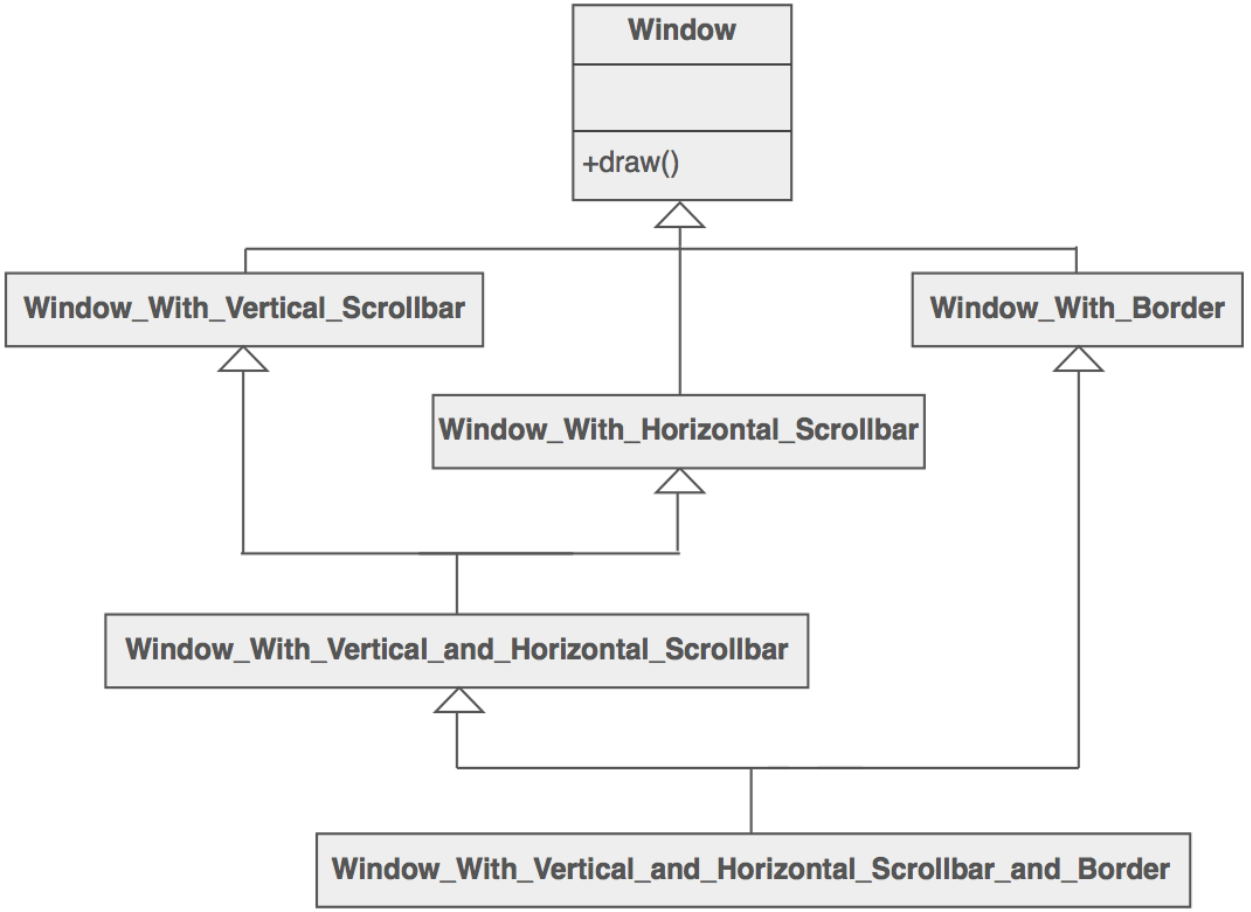
Intent

- Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
- Client-specified embellishment of a core object by recursively wrapping it.
- Wrapping a gift, putting it in a box, and wrapping the box.

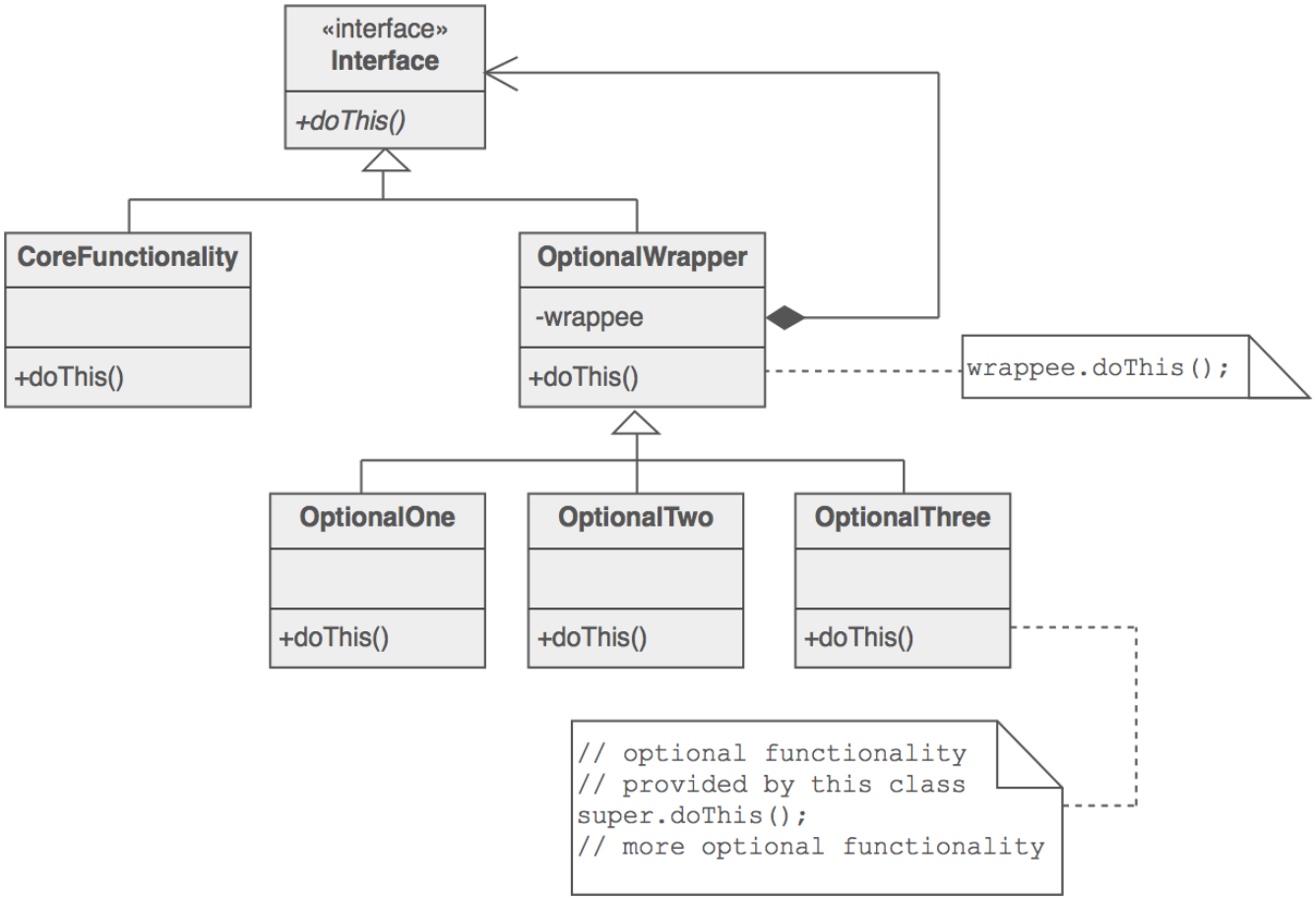
Problem

- You want to add behavior or state to individual objects at run-time. Inheritance is not feasible because it is static and applies to an entire class.

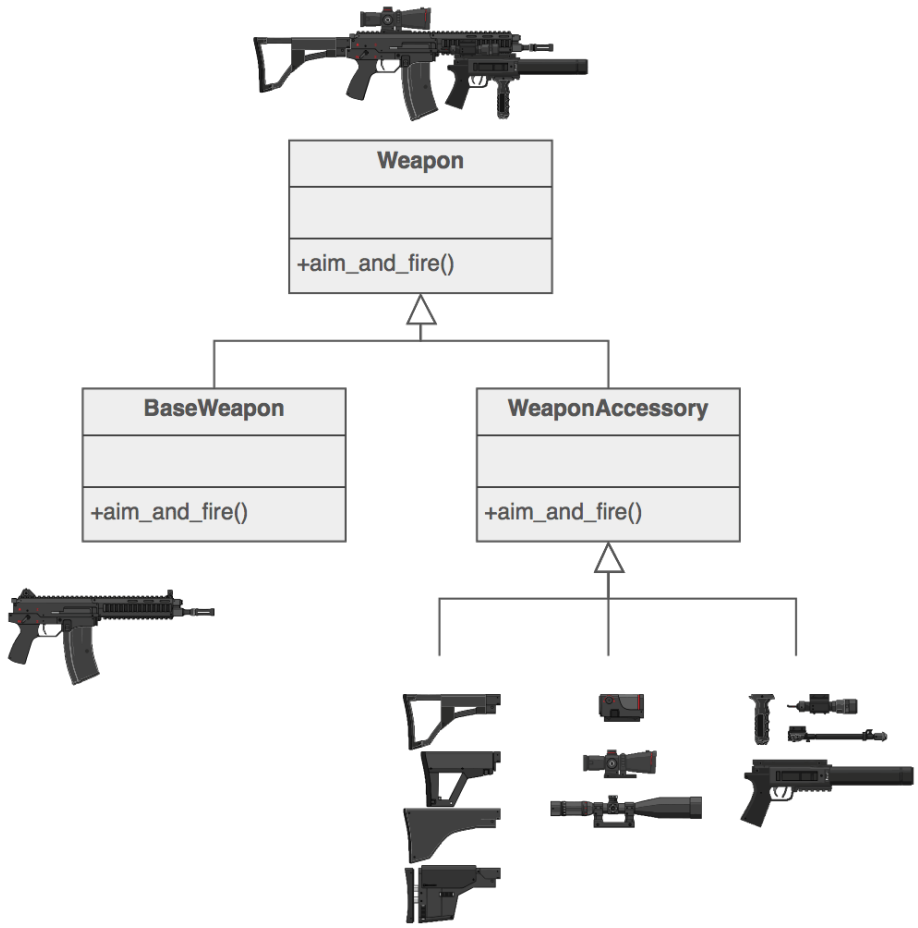
Decorator -- Problem



Decorator


«interface»
Interface
+doThis()
CoreFunctionality
+doThis()
OptionalWrapper
-wrappee
+doThis()
wrappee.doThis();
OptionalOne
+doThis()
OptionalTwo
+doThis()
OptionalThree
+doThis()
// optional functionality
// provided by this class
super.doThis();
// more optional functionality

Decorator


Weapon
+aim_and_fire()
BaseWeapon
+aim_and_fire()
WeaponAccessory
+aim_and_fire()

Facade

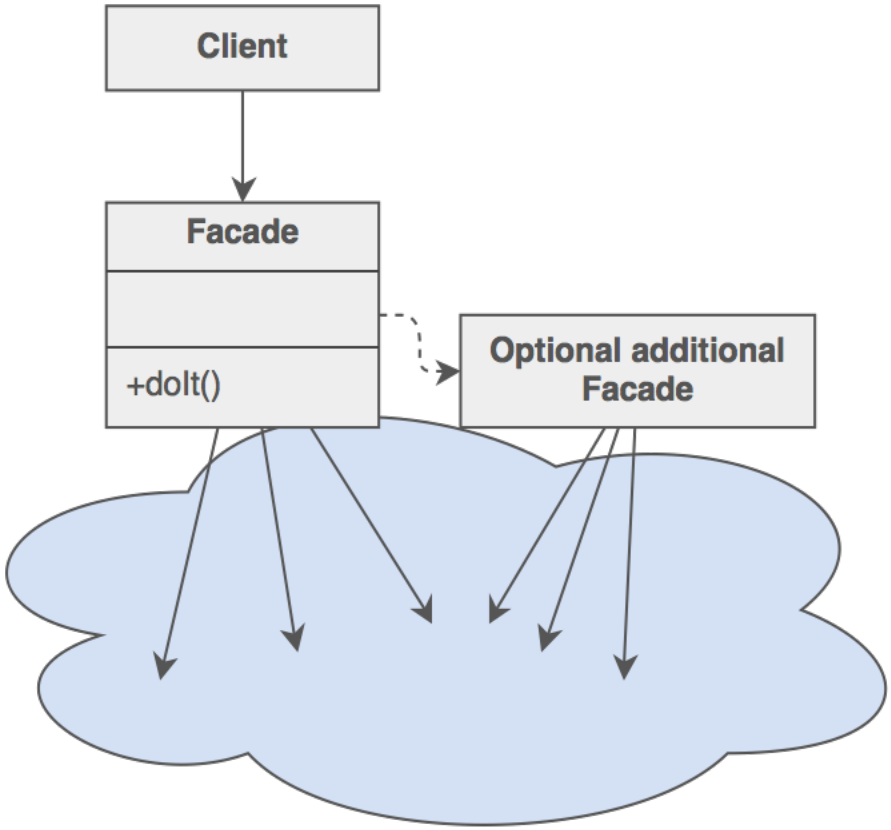
Intent

- Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
- Wrap a complicated subsystem with a simpler interface.

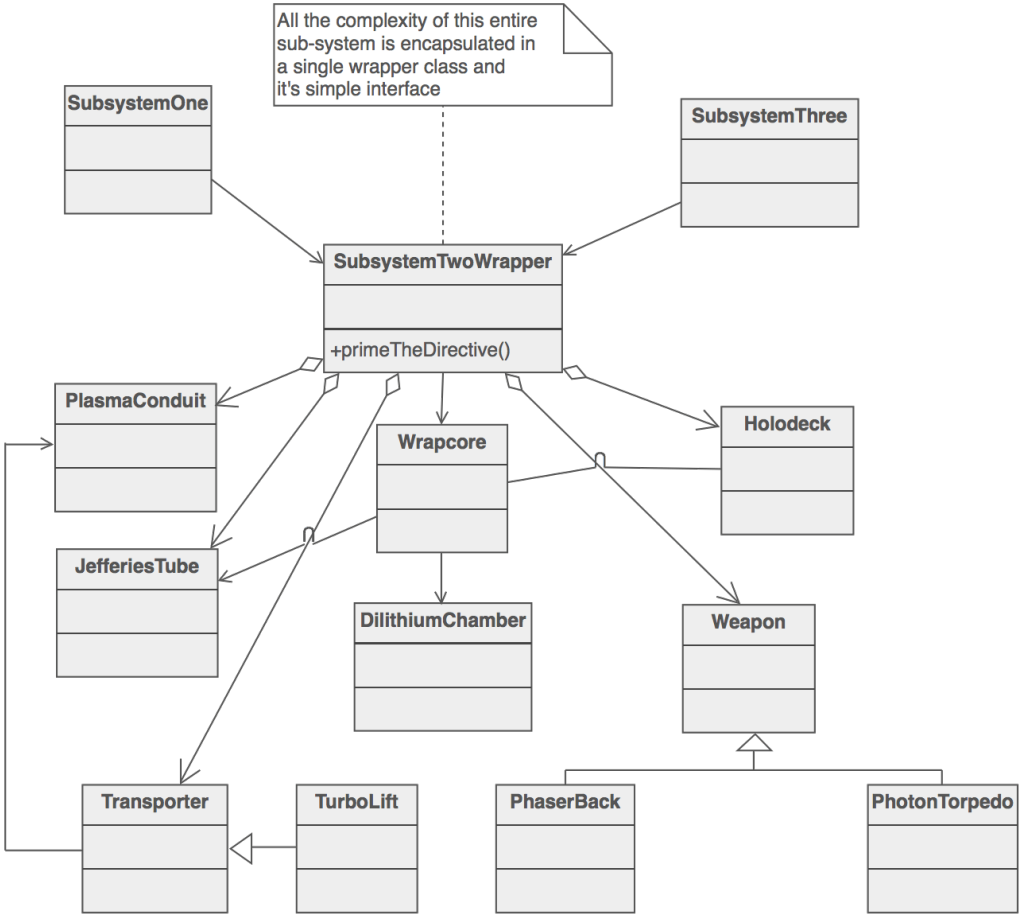
Problem

- A segment of the client community needs a simplified interface to the overall functionality of a complex subsystem.

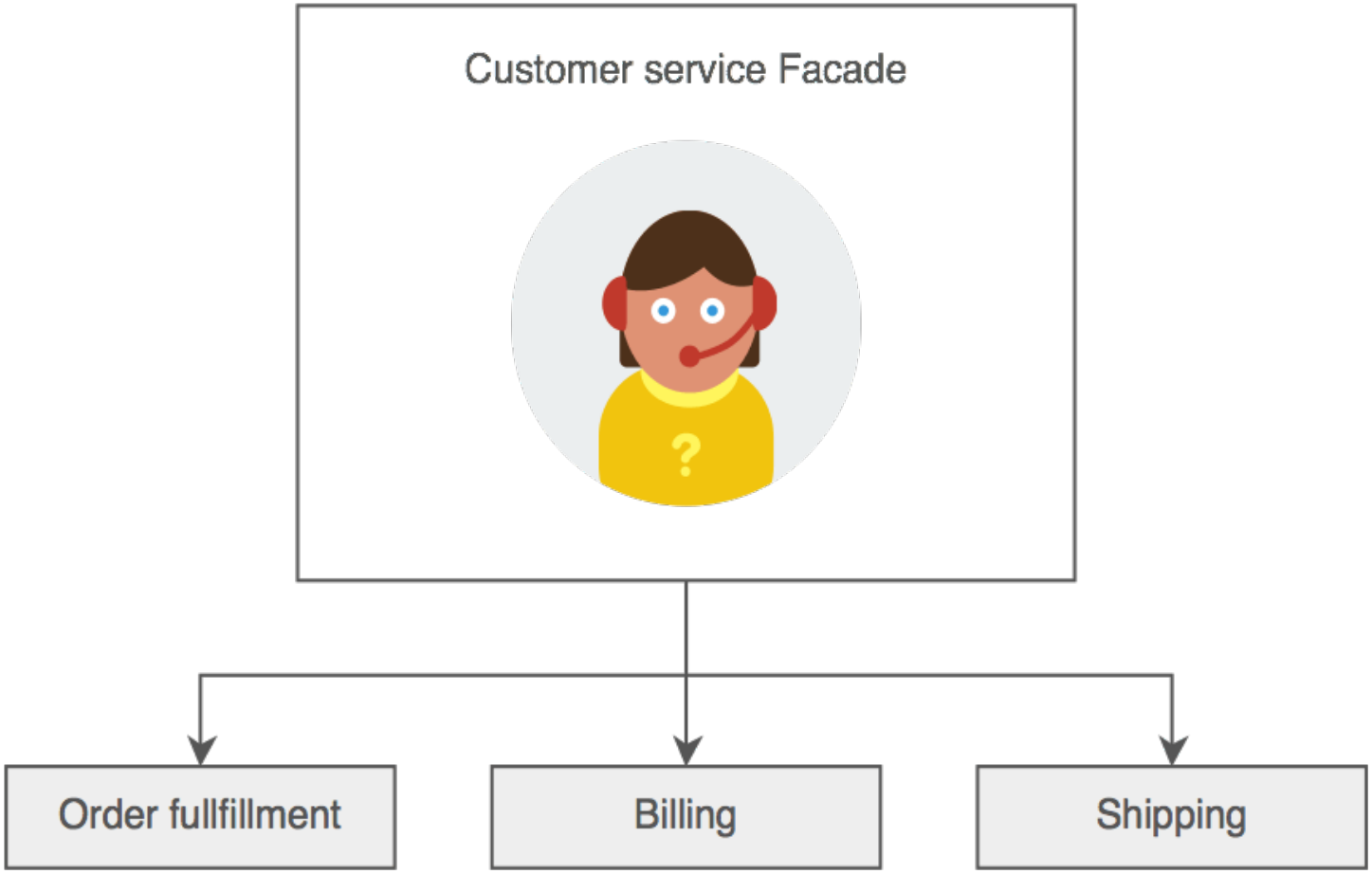
Facade



Facade


All the complexity of this entire sub-system is encapsulated in a single wrapper class and it's simple interface
SubsystemOne
SubsystemThree
SubsystemTwoWrapper
+primeTheDirective()
PlasmaConduit
Wrapcore
Holodeck
JefferiesTube
DilithiumChamber
Weapon
Transporter
TurboLift
PhaserBack
PhotonTorpedo

Facade


Customer service Facade
Order fullfillment
Billing
Shipping

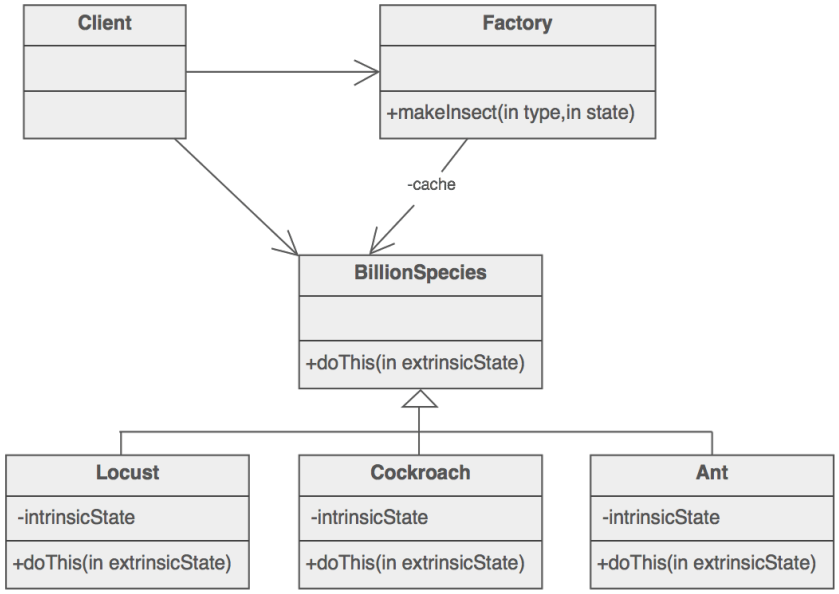
Flyweight

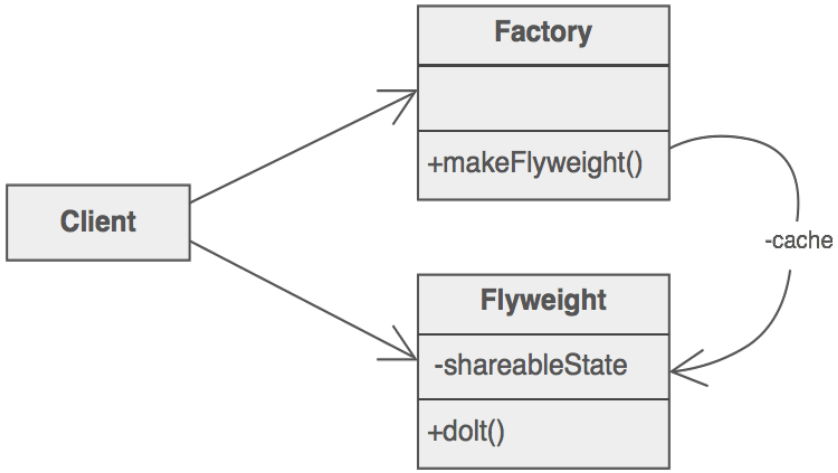
Intent

- Use sharing to support large numbers of fine-grained objects efficiently.
- The Motif GUI strategy of replacing heavy-weight widgets with light-weight gadgets.

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.


Client
Factory
+makeInsect(in type,in state)
-cache
BillionSpecies
+doThis(in extrinsicState)
Locust
-intrinsicState
+doThis(in extrinsicState)
Cockroach
-intrinsicState
+doThis(in extrinsicState)
Ant
-intrinsicState
+doThis(in extrinsicState)


Factory
+makeFlyweight()
Client
-cache
Flyweight
-shareableState
+dolt()

Flyweight

Flyweight

Browser loads images just once and then reuses them from pool:

Private Class Data

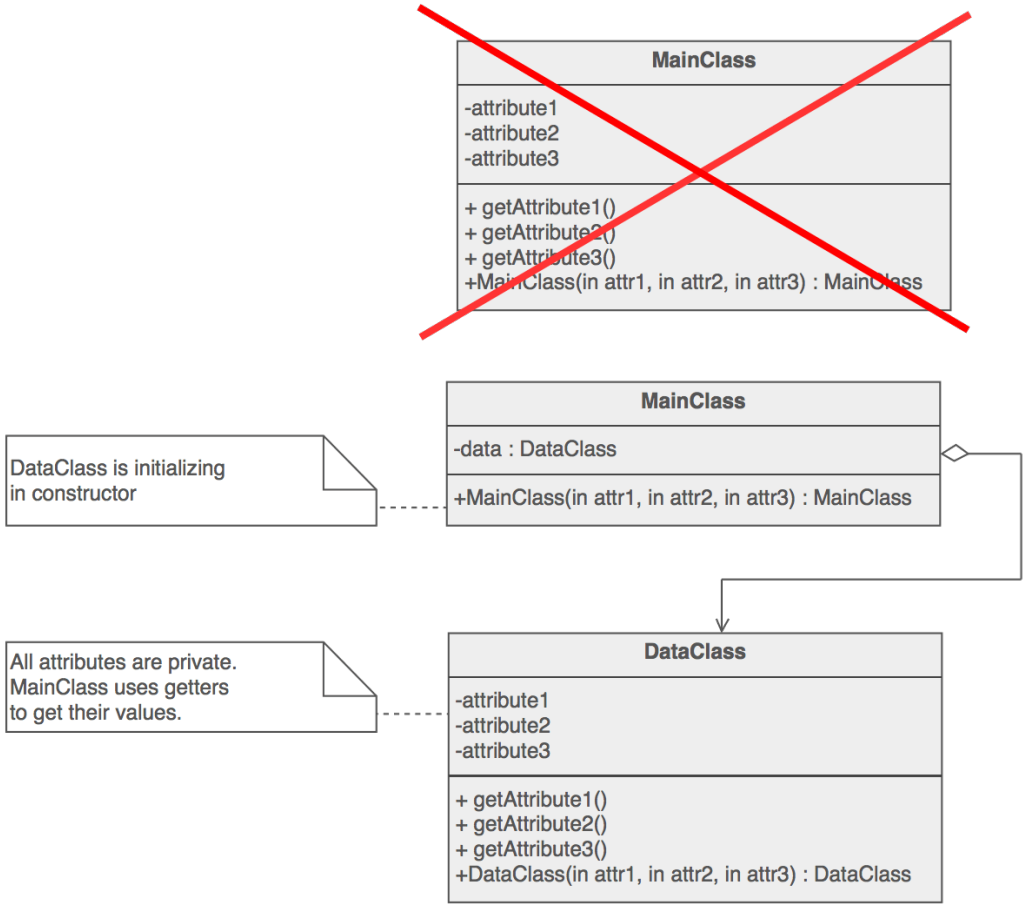
Intent

- Control write access to class attributes
- Separate data from methods that use it
- Encapsulate class data initialization
- Providing new type of final - *final after constructor*

Problem

- The motivation for this design pattern comes from the design goal of protecting class state by minimizing the visibility of its attributes (data)

Private Class Data



Proxy Design Pattern

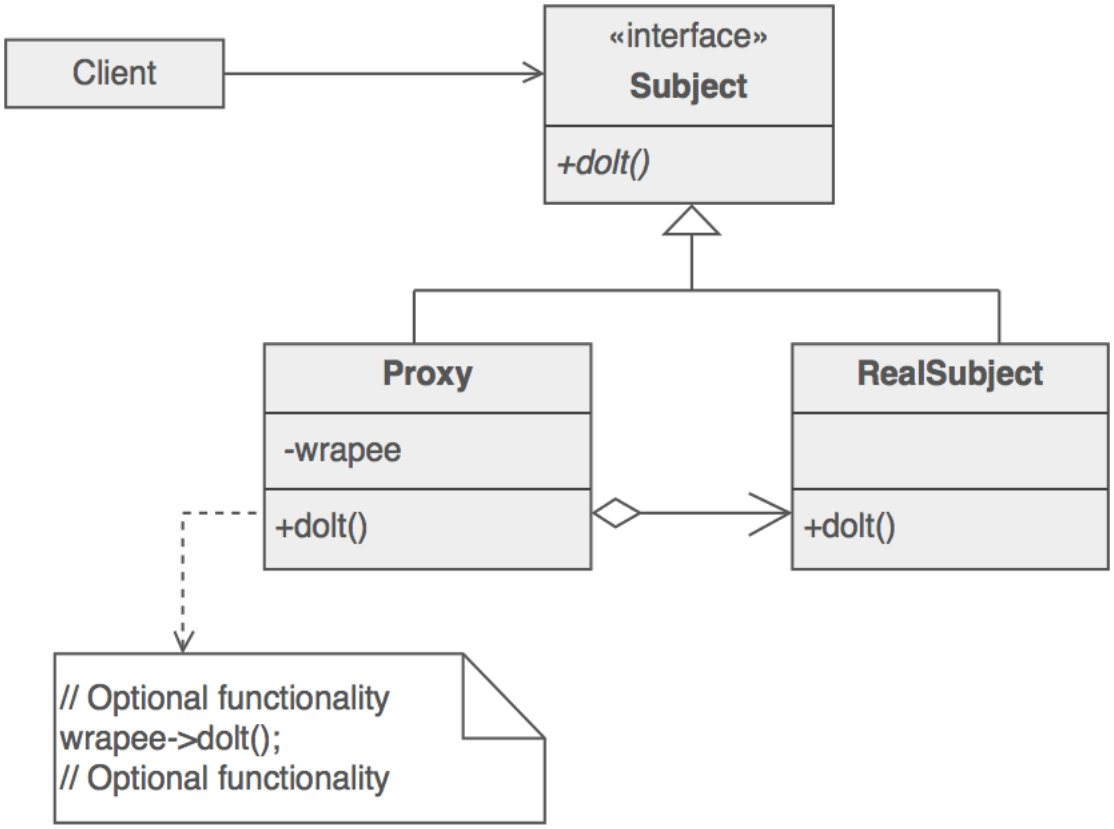
Intent

- 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.

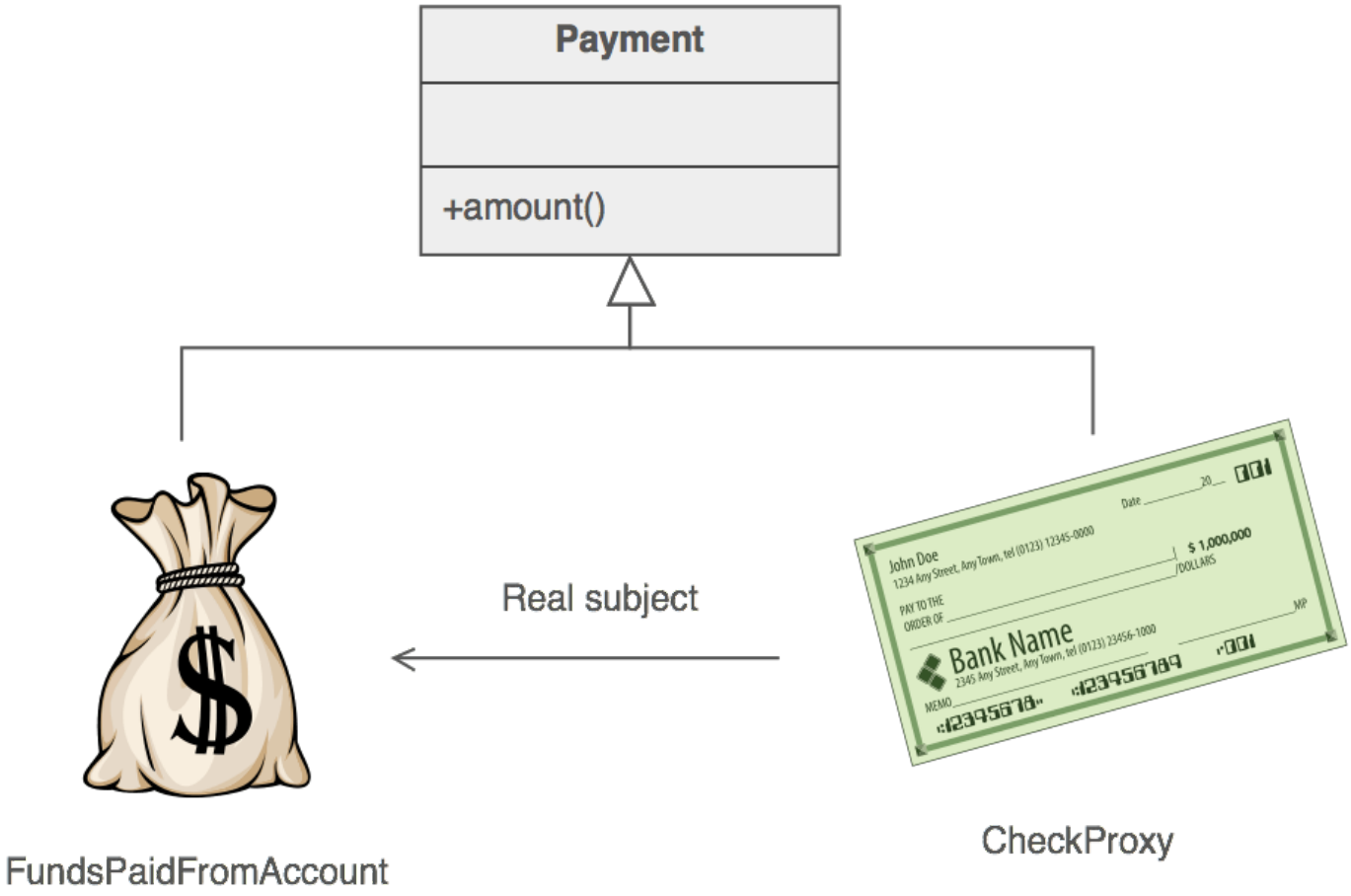
Problem

- 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.

Proxy Design Pattern



Proxy Design Pattern



Behavioral patterns

Design Patterns that identify common communication patterns between objects and realize these patterns.

These patterns increase flexibility in carrying out this communication.

Command*

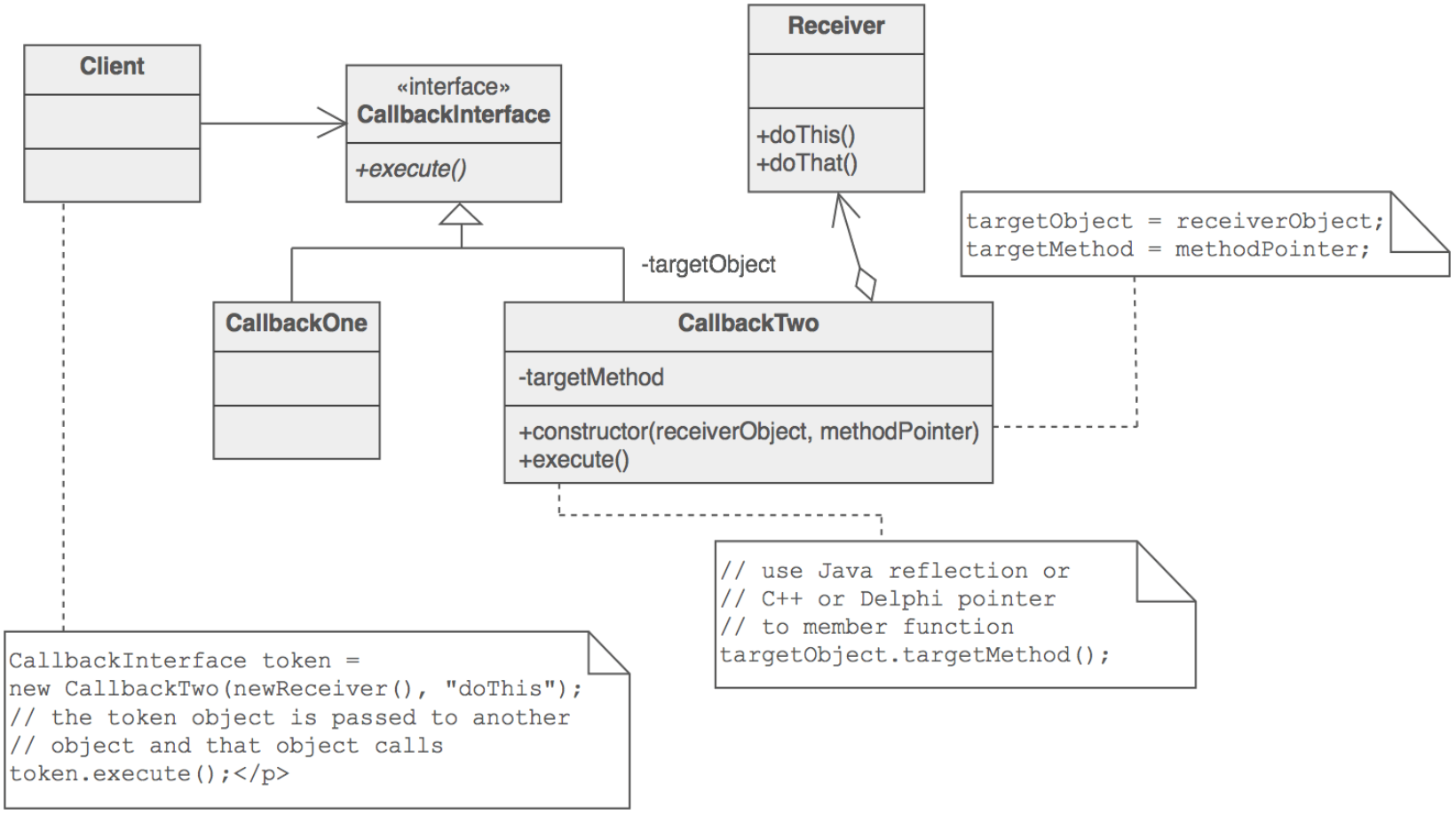
Intent

- Encapsulate a request as an object, thereby letting you parametrize clients with different requests, queue or log requests, and support undoable operations.
- Promote "invocation of a method on an object" to full object status
- An object-oriented callback

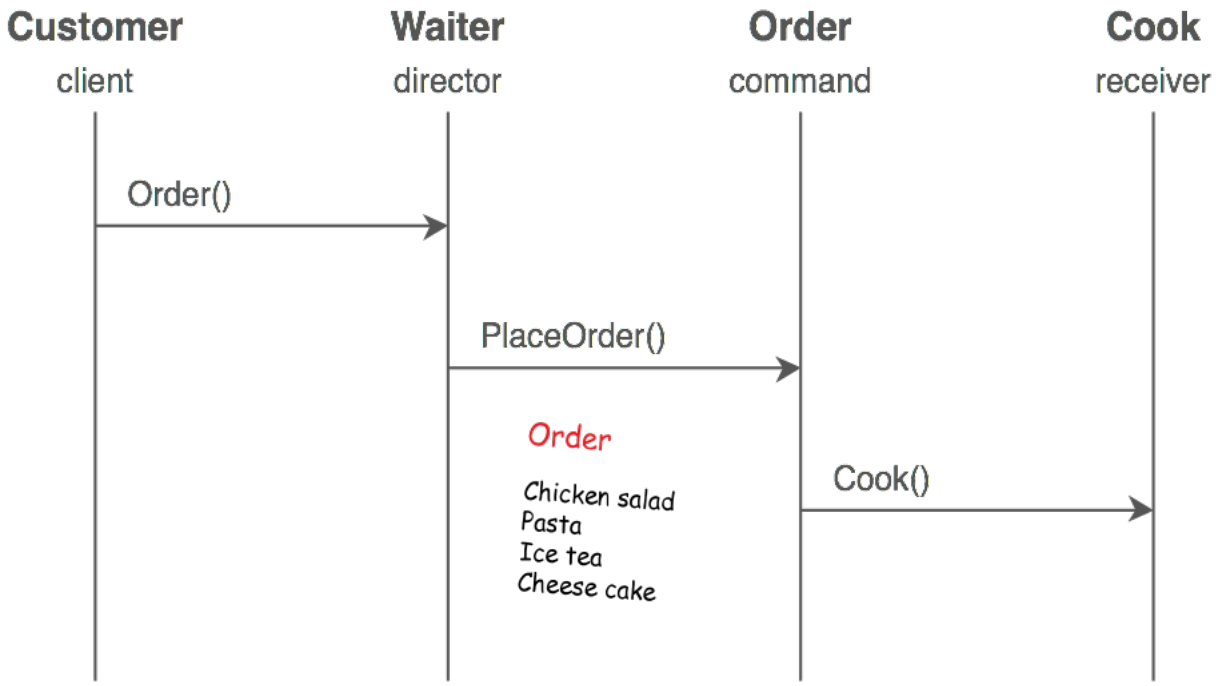
Problem

- Need to issue requests to objects without knowing anything about the operation being requested or the receiver of the request.

Command*


Client
«interface»
CallbackInterface
+execute()
Receiver
+doThis()
+doThat()
-targetObject
CallbackOne
CallbackTwo
-targetMethod
+constructor(receiverObject, methodPointer)
+execute()
targetObject = receiverObject;
targetMethod = methodPointer;
// use Java reflection or
// C++ or Delphi pointer
// to member function
targetObject.targetMethod();
CallbackInterface token =
new CallbackTwo(newReceiver(), "doThis");
// the token object is passed to another
// object and that object calls
token.execute();</p>

Command*


Customer
client
Waiter
director
Order
command
Cook
receiver
Order()
PlaceOrder()
Order
Chicken salad
Pasta
Ice tea
Cheese cake
Cook()

Command*

1. Define a Command interface with a method signature like execute().
2. Create one or more derived classes that encapsulate some subset of the following: a "receiver" object, the method to invoke, the arguments to pass.
3. Instantiate a Command object for each deferred execution request.
4. Pass the Command object from the creator (aka sender) to the invoker (aka receiver).
5. The invoker decides when to execute().

Iterator*

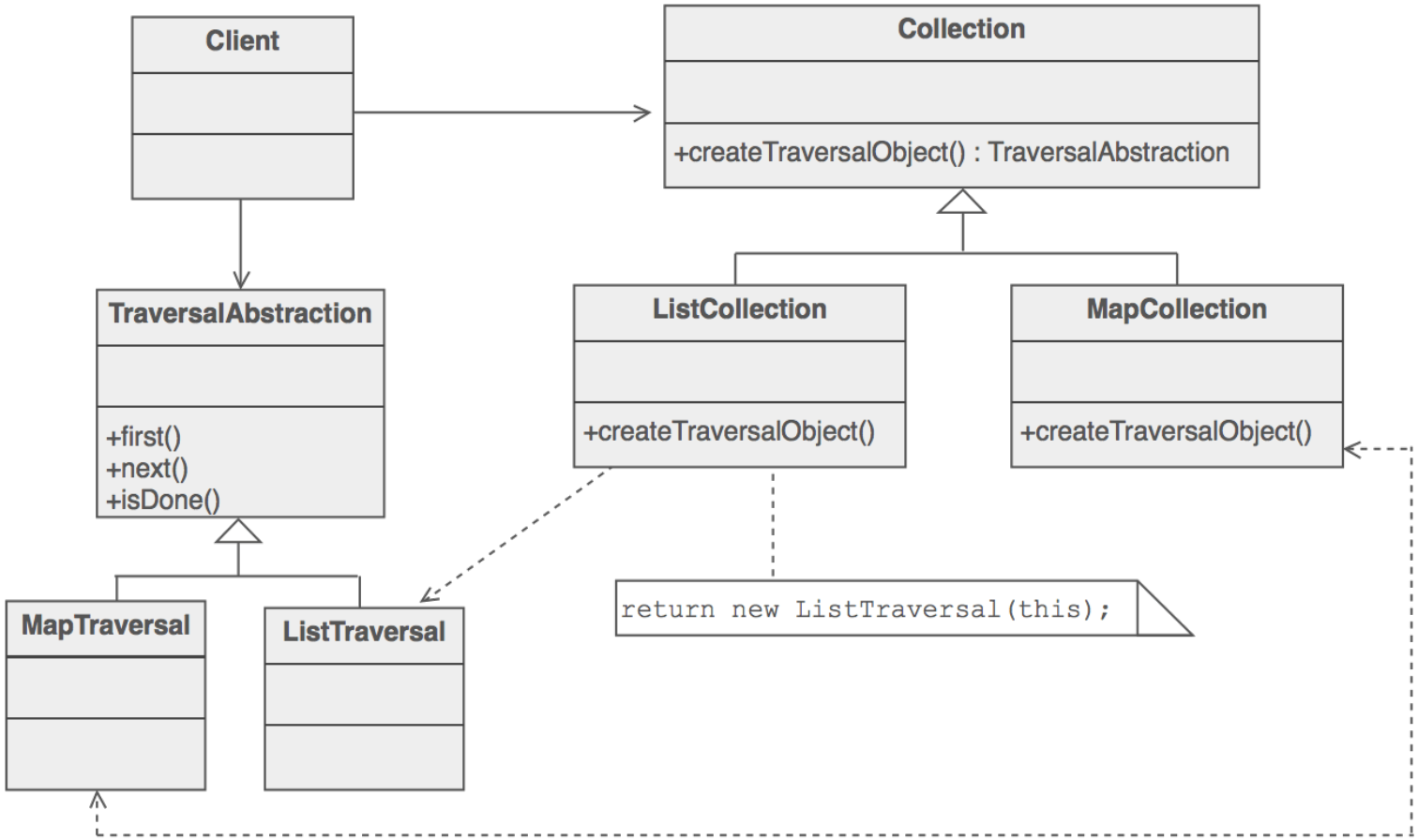
Intent

- Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
- The C++ and Java standard library abstraction that makes it possible to decouple collection classes and algorithms.
- Promote to "full object status" the traversal of a collection.
- Polymorphic traversal

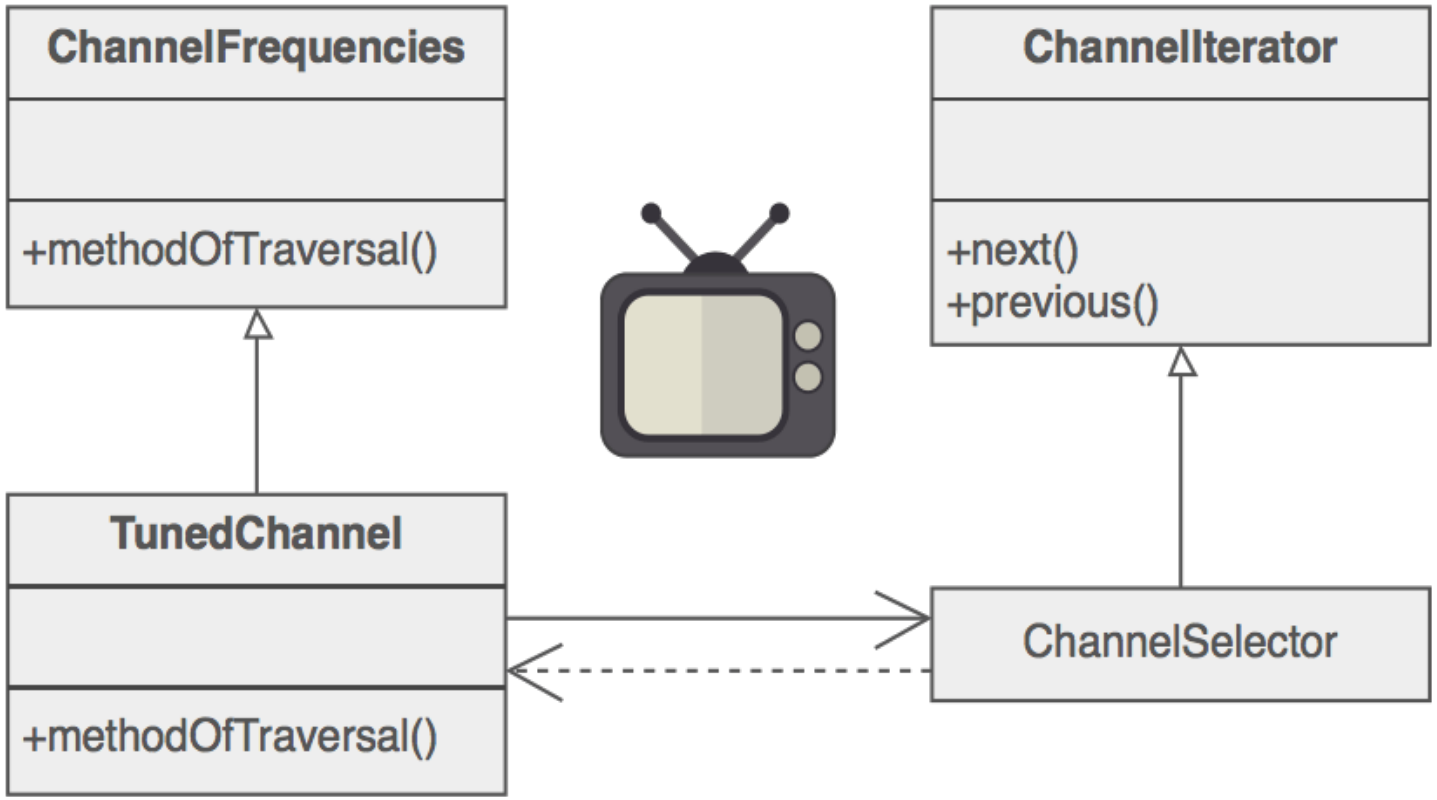
Problem

- Need to "abstract" the traversal of wildly different data structures so that algorithms can be defined that are capable of interfacing with each transparently.

Iterator*


Client
Collection
+createTraversalObject() : TraversalAbstraction
TraversalAbstraction
+first()
+next()
+isDone()
ListCollection
+createTraversalObject()
MapCollection
+createTraversalObject()
MapTraversal
ListTraversal
return new ListTraversal(this);

Iterator*


ChannelFrequencies
+methodOfTraversal()
ChannelIterator
+next()
+previous()
TunedChannel
+methodOfTraversal()
ChannelSelector

Memento*

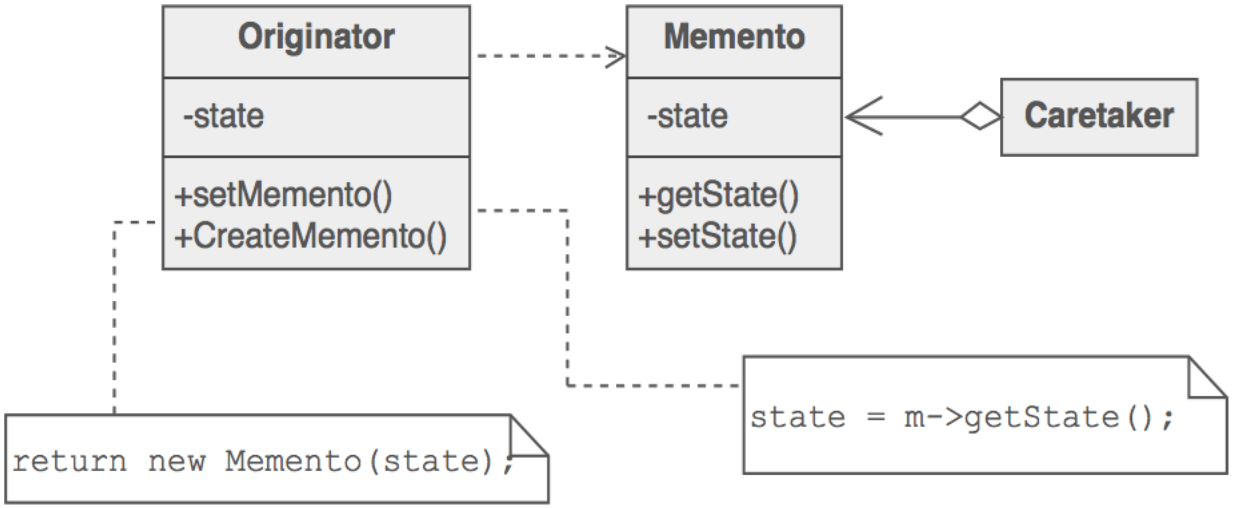
Intent

- Without violating encapsulation, capture and externalize an object's internal state so that the object can be returned to this state later.
- A magic cookie that encapsulates a "check point" capability.
- Promote undo or rollback to full object status.

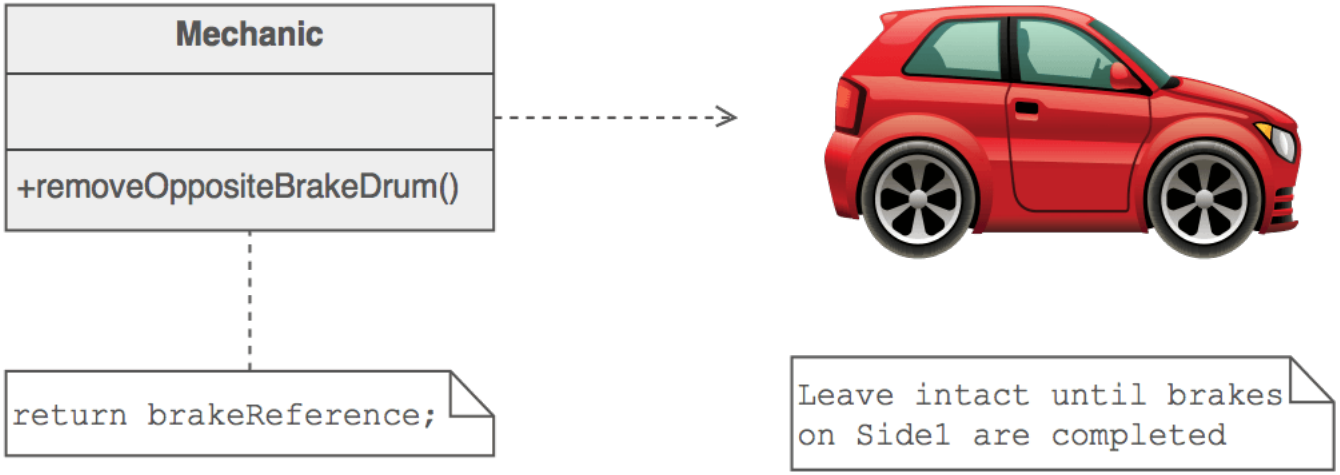
Problem

- Need to restore an object back to its previous state (e.g. "undo" or "rollback" operations).

Memento*


Originator
-state
+setMemento()
+CreateMemento()
Memento
-state
+getState()
+setState()
Caretaker
return new Memento(state);
state = m->getState();

Memento*


Mechanic
+removeOppositeBrakeDrum()
return brakeReference;
Leave intact until brakes
on Side1 are completed

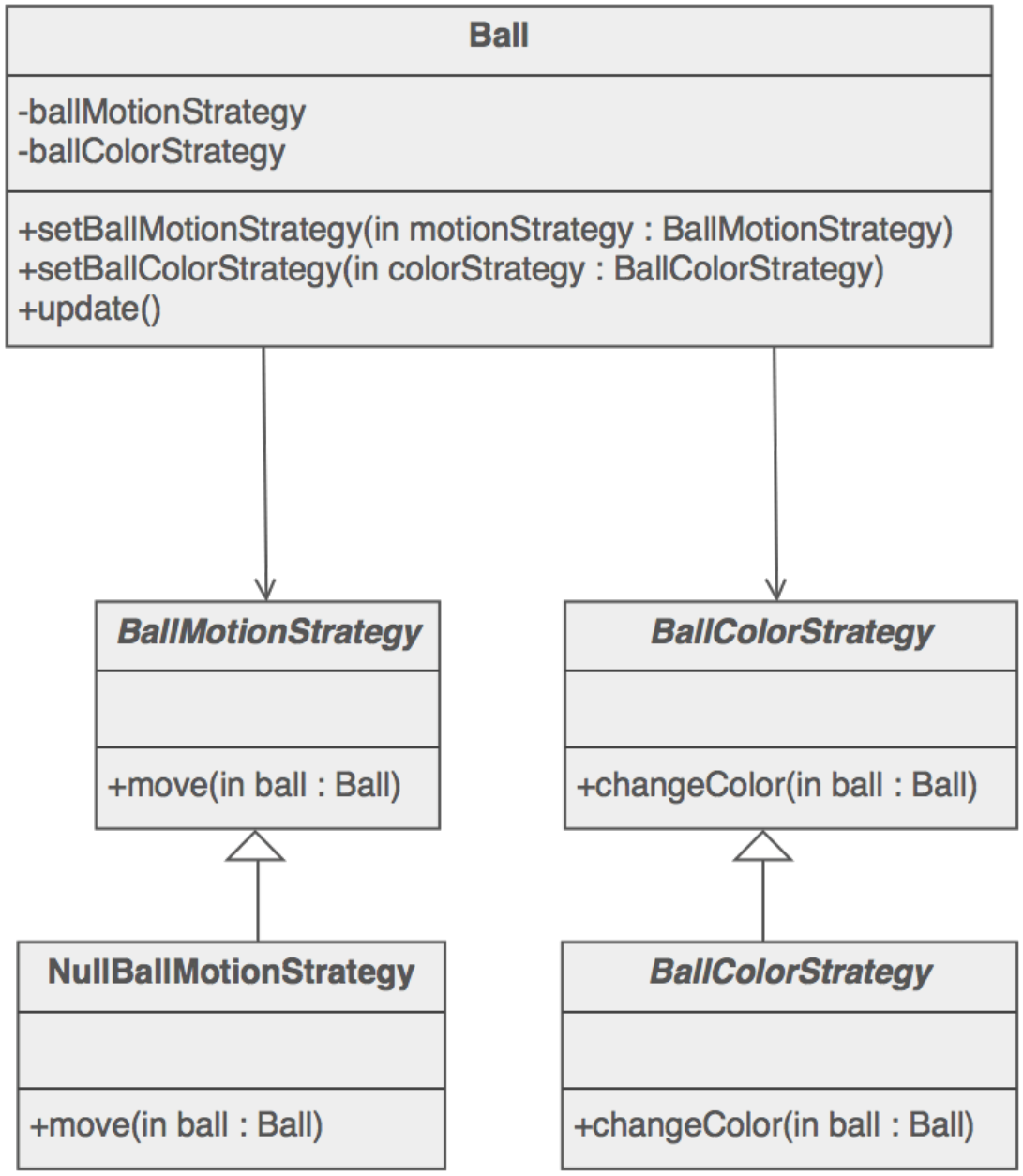
Null Object

Intent

- The intent of a Null Object is to encapsulate the absence of an object by providing a substitutable alternative that offers suitable default do nothing behavior. In short, a design where "nothing will come of nothing”

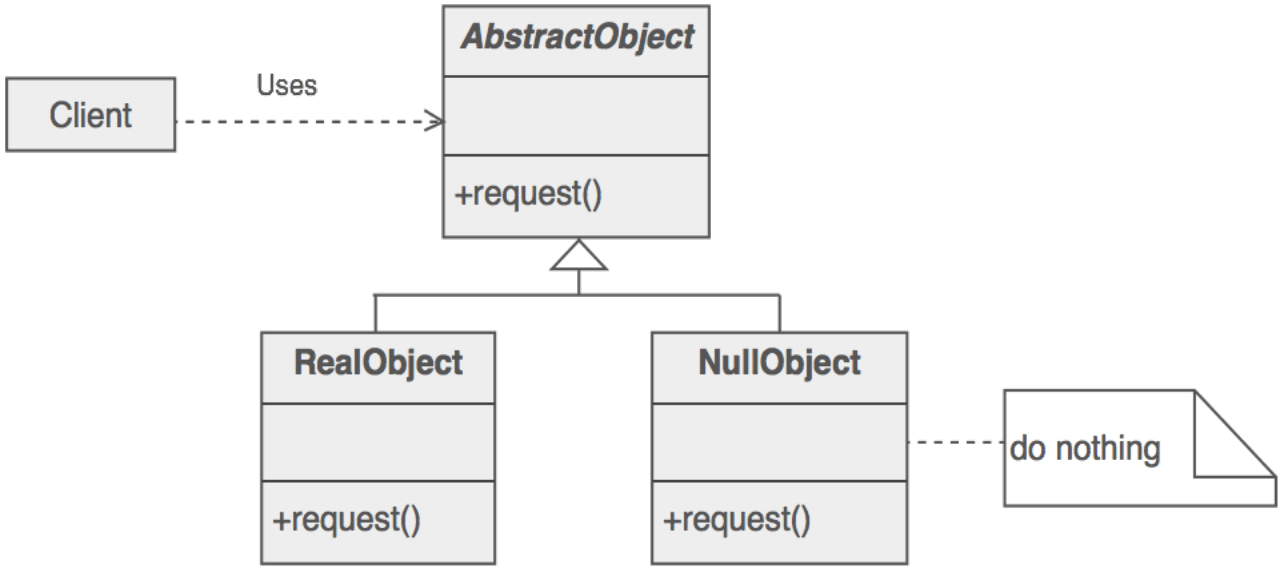
- Use the Null Object pattern when
  - an object requires a collaborator. The Null Object pattern does not introduce this collaboration--it makes use of a collaboration that already exists
  - some collaborator instances should do nothing
  - you want to abstract the handling of null away from the client

Null Object



The key to the Null Object pattern is an abstract class that defines the interface for all objects of this type. The Null Object is implemented as a subclass of this abstract class.

Null Object


Client
Uses
AbstractObject
+request()
RealObject
+request()
NullObject
+request()
do nothing

Observer*

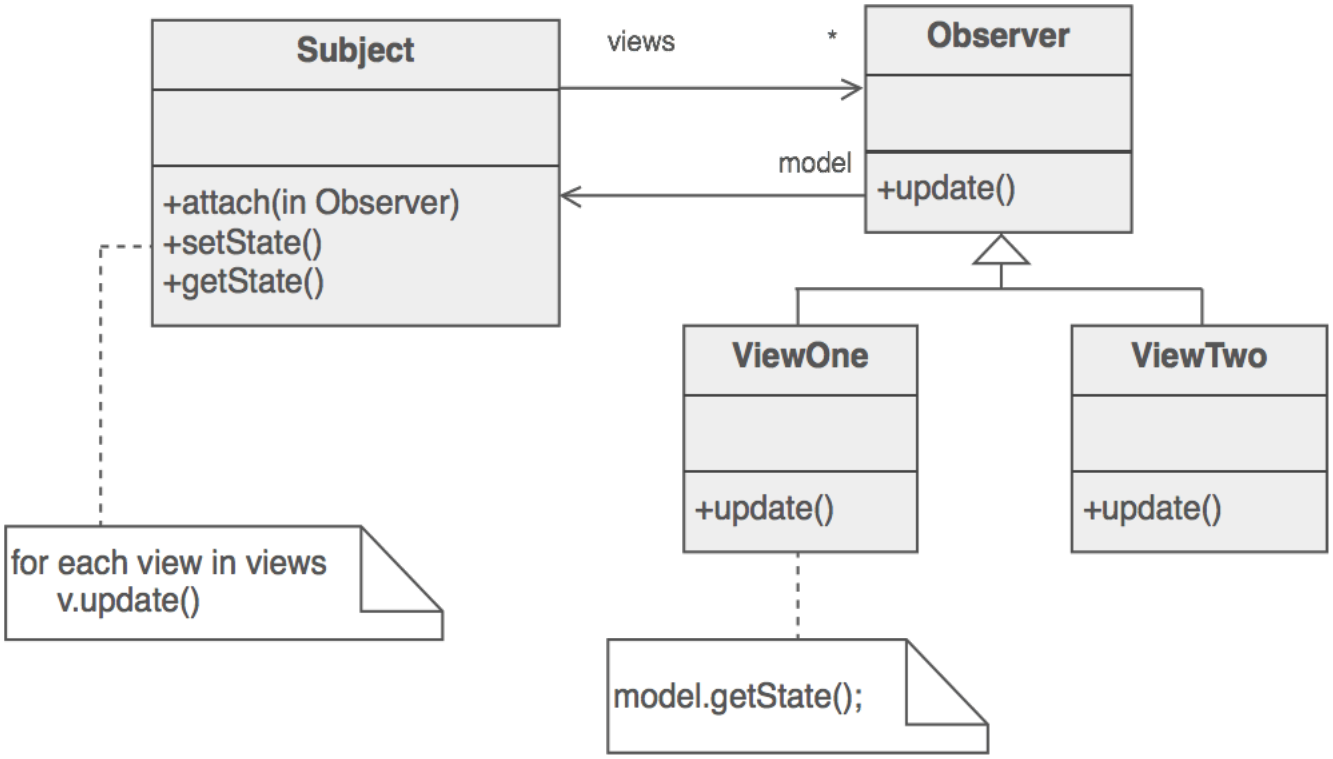
Intent

- Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- Encapsulate the core (or common or engine) components in a Subject abstraction, and the variable (or optional or user interface) components in an Observer hierarchy.
- The "View" part of Model-View-Controller.

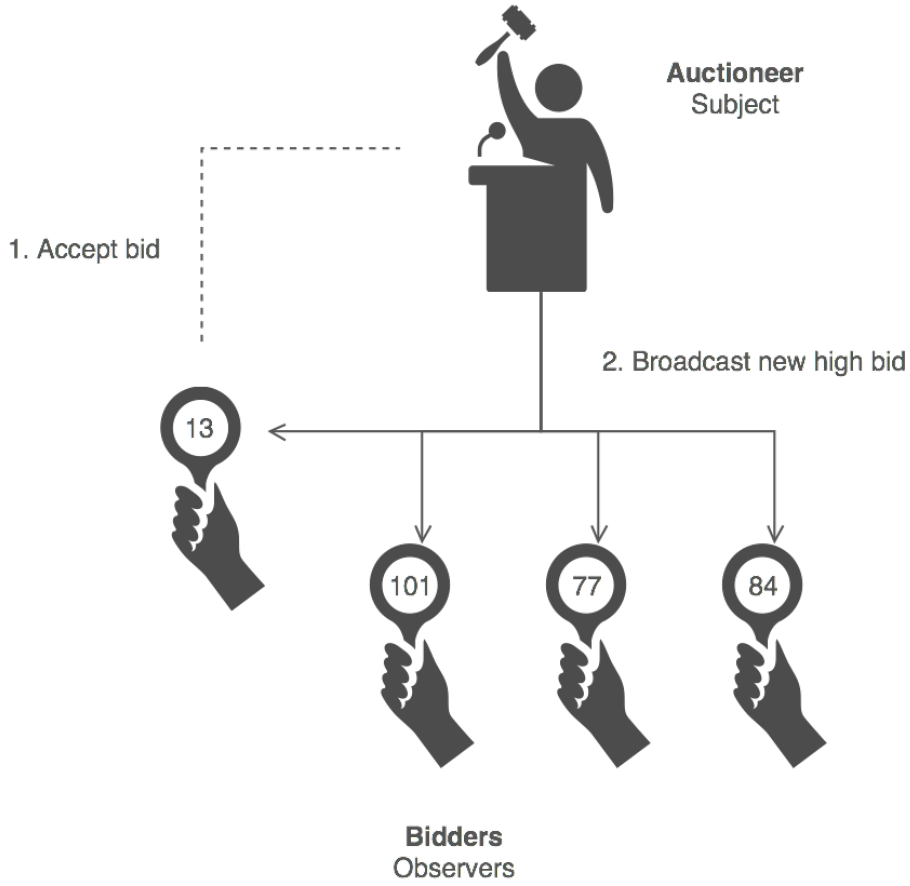
Problem

- A large monolithic design does not scale well as new graphing or monitoring requirements are levied.

Observer*


Subject
views
*
Observer
model
+attach(in Observer)
+setState()
+getState()
+update()
ViewOne
ViewTwo
+update()
+update()
for each view in views
v.update()
model.getState();

Observer*


Auctioneer
Subject
1. Accept bid
2. Broadcast new high bid
13
101
77
84
Bidders
Observers

Observer*

Check list

1. Differentiate between the core (or independent) functionality and the optional (or dependent) functionality.
2. Model the independent functionality with a "subject" abstraction.
3. Model the dependent functionality with an "observer" hierarchy.
4. The Subject is coupled only to the Observer base class.
5. The client configures the number and type of Observers.
6. Observers register themselves with the Subject.
7. The Subject broadcasts events to all registered Observers.
8. The Subject may "push" information at the Observers, or, the Observers may "pull" the information they need from the Subject.

State

Intent

- Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
- An object-oriented state machine
- wrapper + polymorphic wrappee + collaboration

Problem

- A monolithic object's behavior is a function of its state, and it must change its behavior at run-time depending on that state. Or, an application is characterized by large and numerous case statements that vector flow of control based on the state of the application.

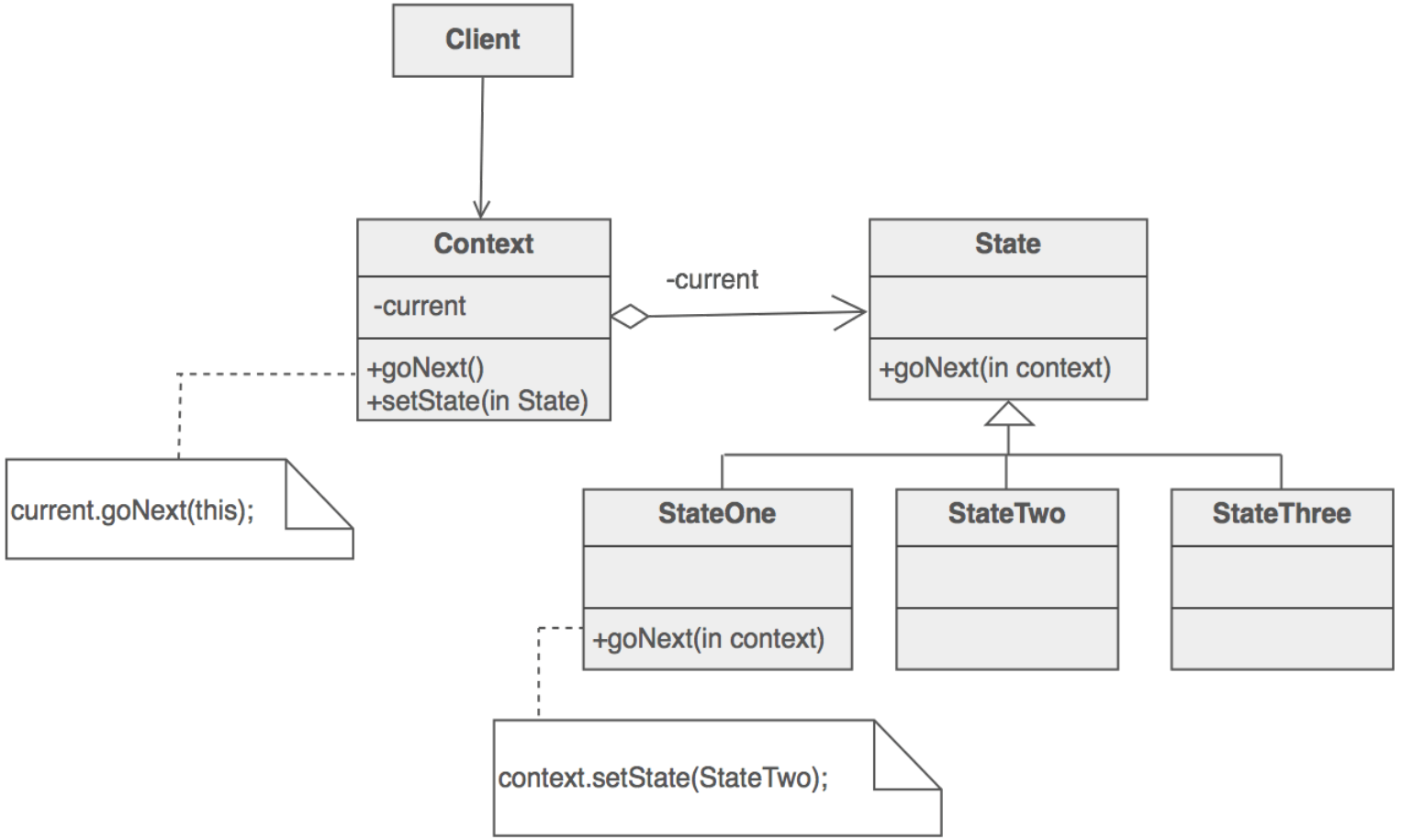
Models a **state machine** of sorts

State

The State pattern is a solution to the problem of how to make behavior depend on state.

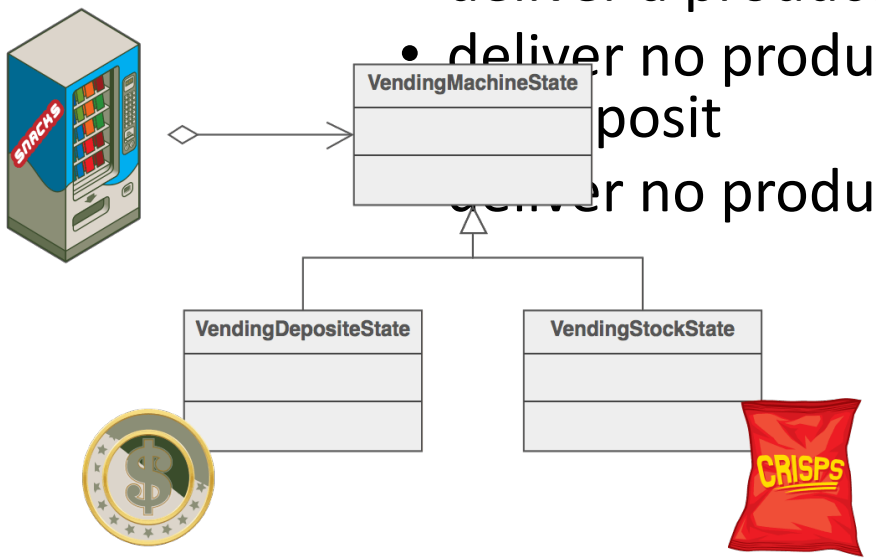
- Define a "context" class to present a single interface to the outside world.
- Define a State abstract base class.
- Represent the different "states" of the state machine as derived classes of the State base class.
- Define state-specific behavior in the appropriate State derived classes.
- Maintain a pointer to the current "state" in the "context" class.
- To change the state of the state machine, change the current "state" pointer.

State


Client
Context
-current
+goNext()
+setState(in State)
-current
State
+goNext(in context)
current.goNext(this);
StateOne
+goNext(in context)
StateTwo
StateThree
context.setState(StateTwo);

State

- Vending machines have states based on the inventory, amount of currency deposited, the ability to make change, the item selected, etc.
- When currency is deposited and a selection is made, a vending machine will either:
  - deliver a product and no change
  - deliver a product and change
  - deliver no product due to insufficient currency deposit
  - deliver no product due to inventory depletion.



Strategy*

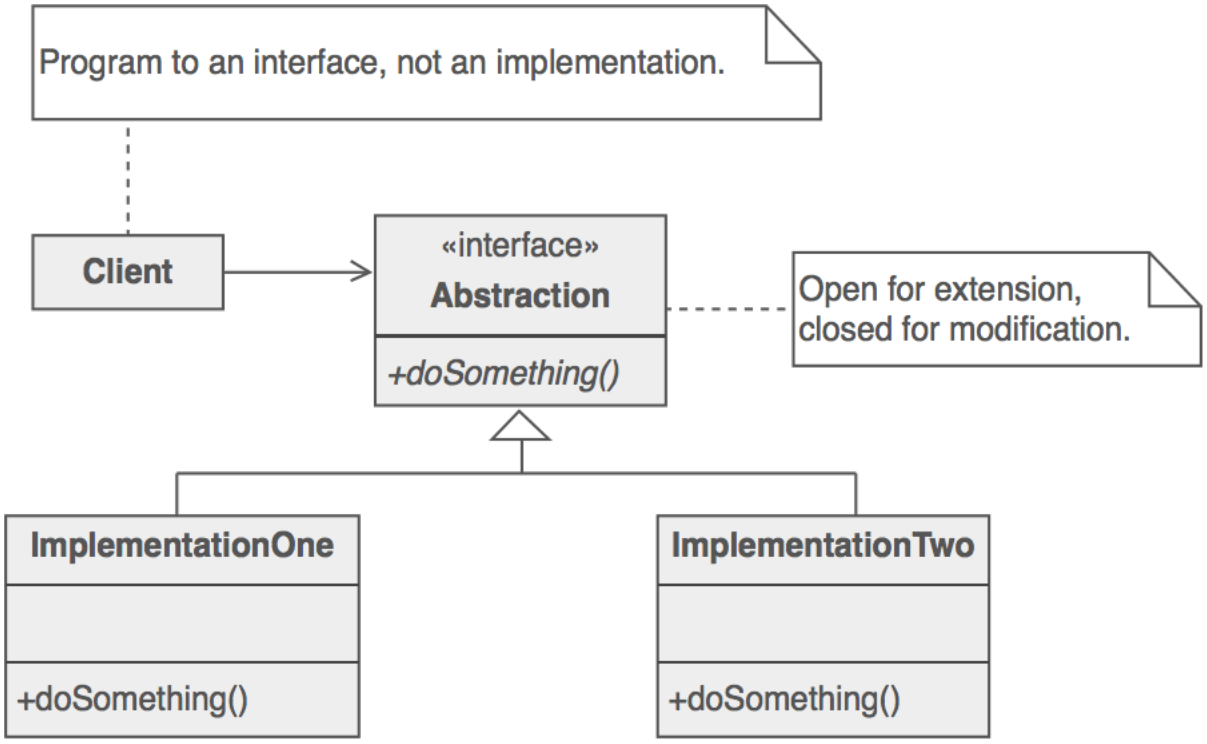
Intent

- Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it.
- Capture the abstraction in an interface, bury implementation details in derived classes.

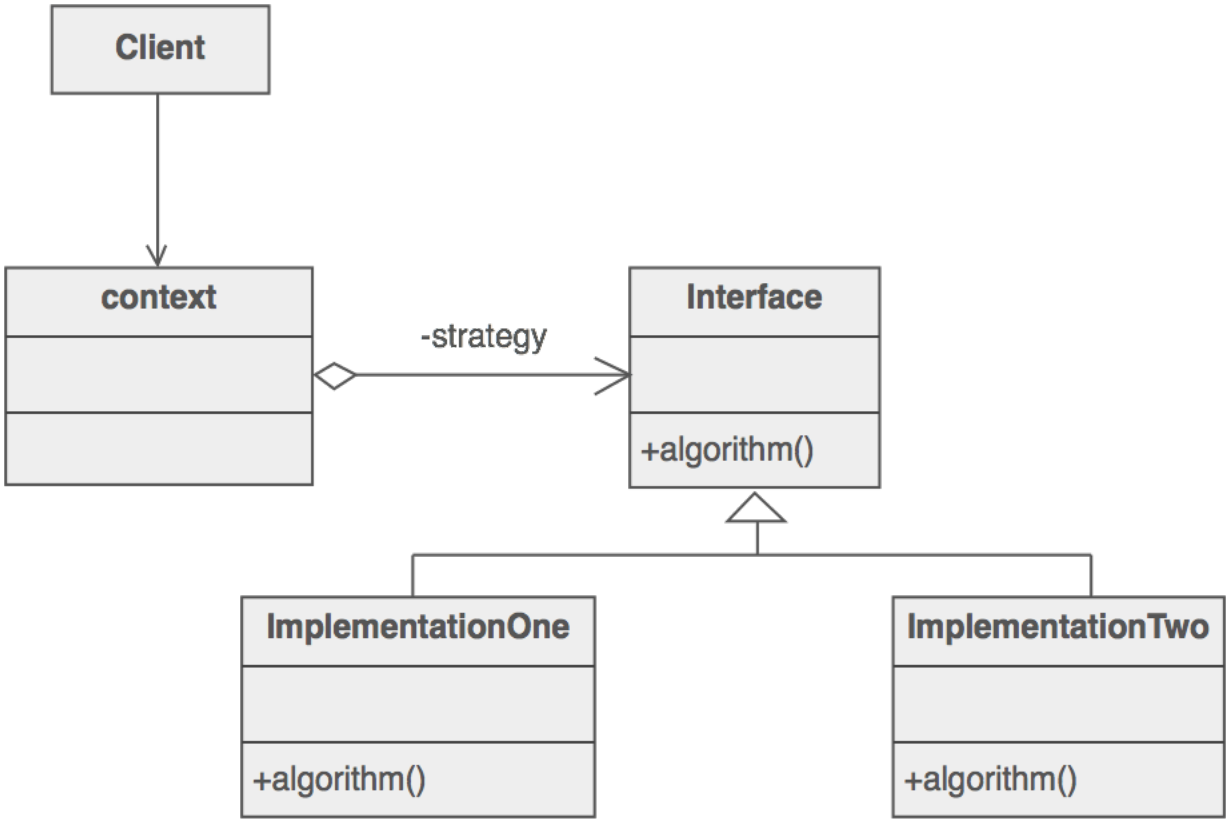
Problem

- One of the dominant strategies of object-oriented design is the "open-closed principle".

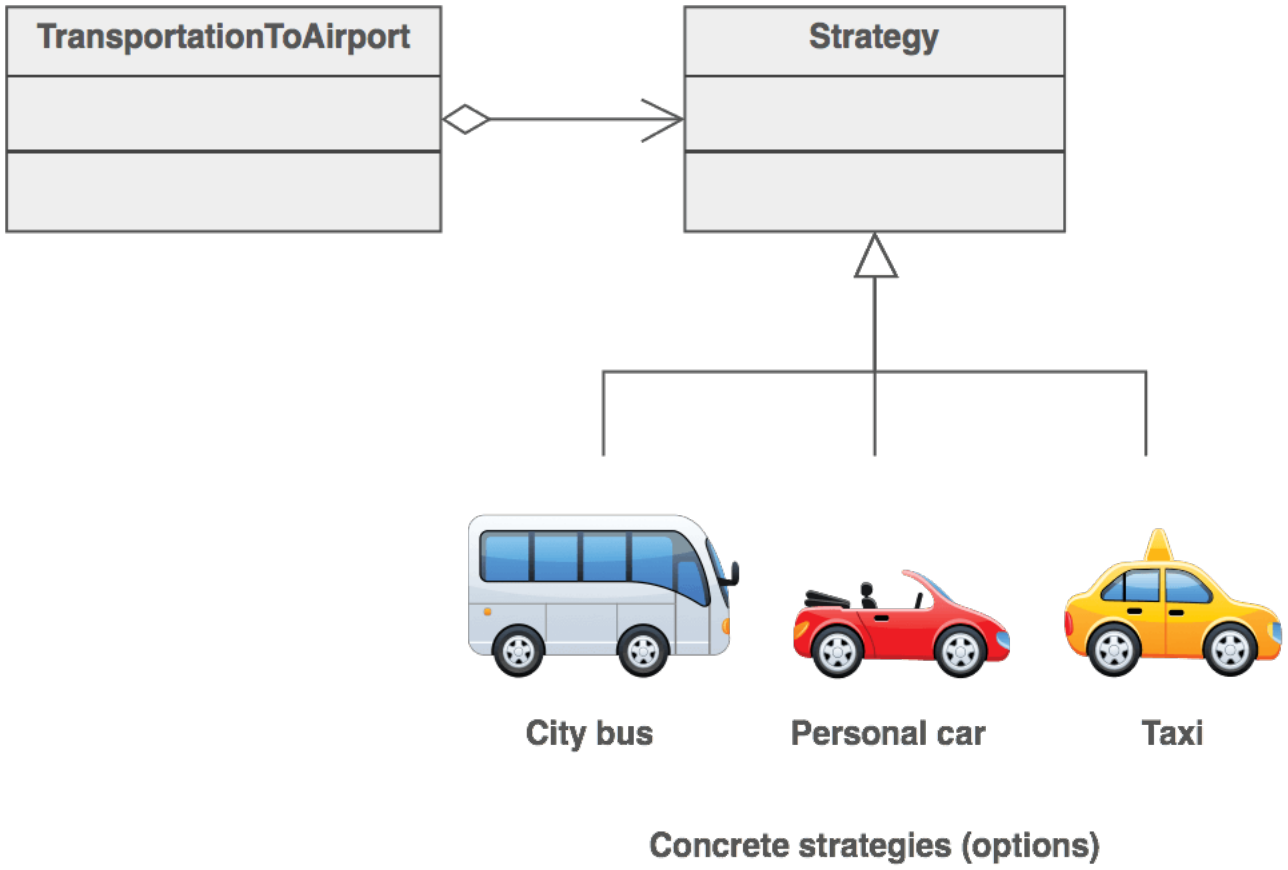
Strategy*


Program to an interface, not an implementation.
Client
«interface»
Abstraction
+doSomething()
Open for extension,
closed for modification.
ImplementationOne
+doSomething()
ImplementationTwo
+doSomething()

Strategy*


Client
context
-strategy
Interface
+algorithm()
ImplementationOne
+algorithm()
ImplementationTwo
+algorithm()

Strategy*


TransportationToAirport
Strategy
City bus
Personal car
Taxi
Concrete strategies (options)

Strategy*

Check list

1. Identify an algorithm (i.e. a behavior) that the client would prefer to access through a "flex point".
2. Specify the signature for that algorithm in an interface.
3. Bury the alternative implementation details in derived classes.
4. Clients of the algorithm couple themselves to the interface.

Template Method*

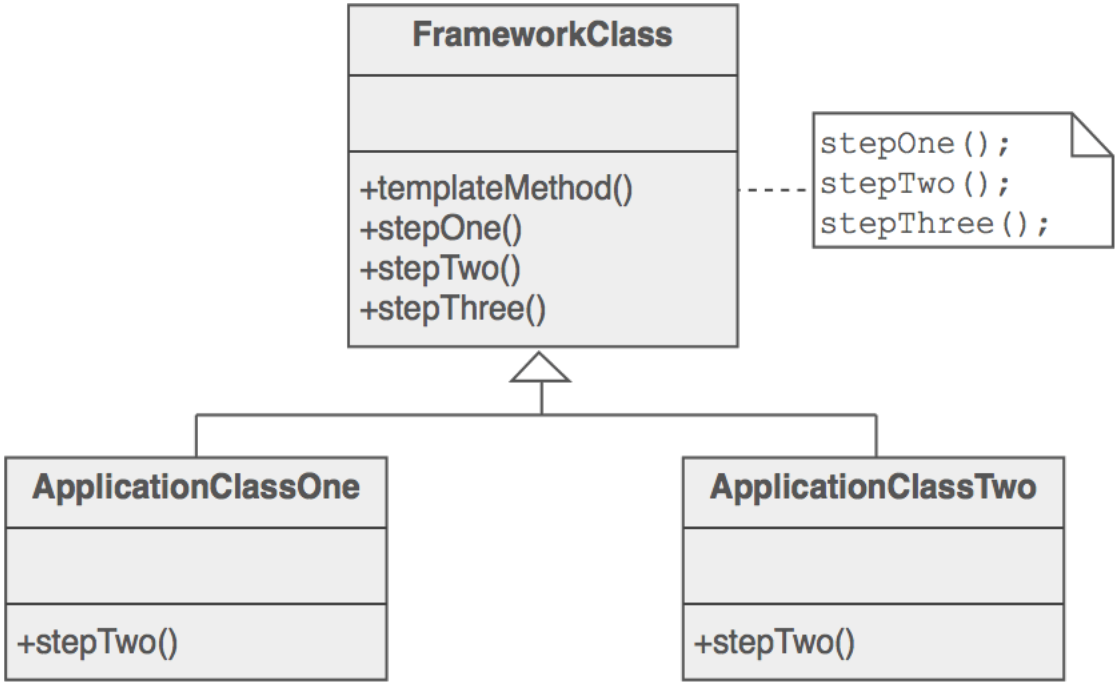
Intent

- Define the skeleton of an algorithm in an operation, deferring some steps to client subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
- Base class declares algorithm 'placeholders', and derived classes implement the placeholders.

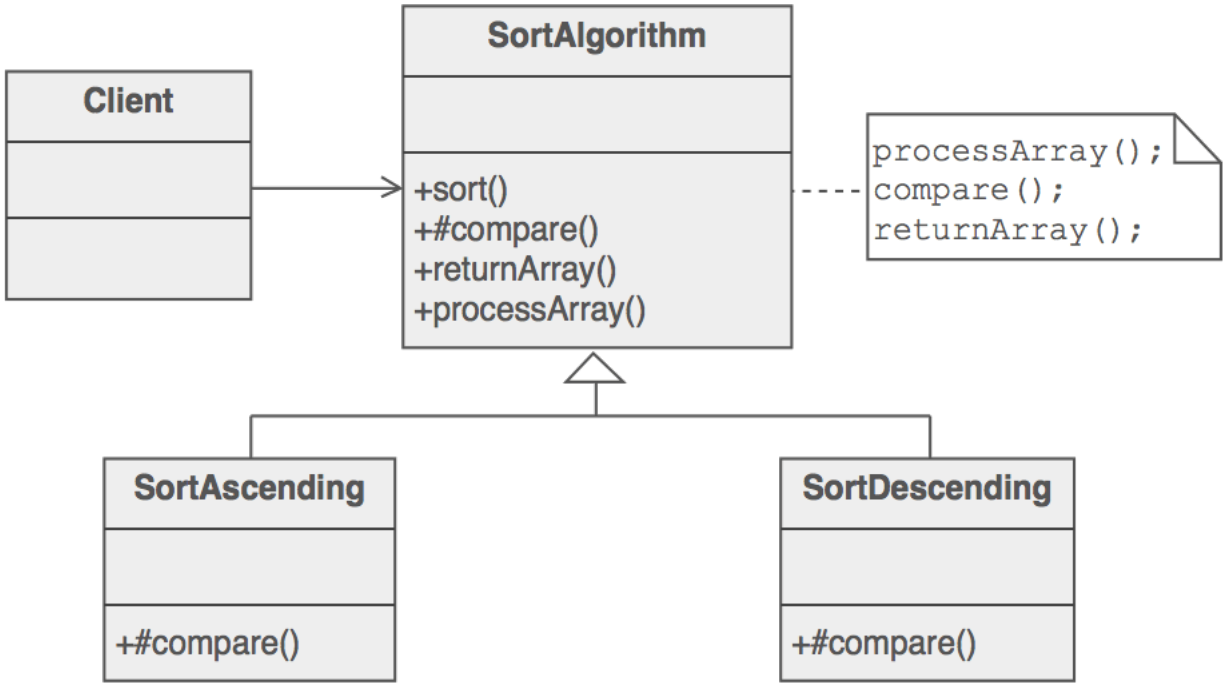
Problem

- Two different components have significant similarities, but demonstrate no reuse of common interface or implementation. If a change common to both components becomes necessary, duplicate effort must be expended.

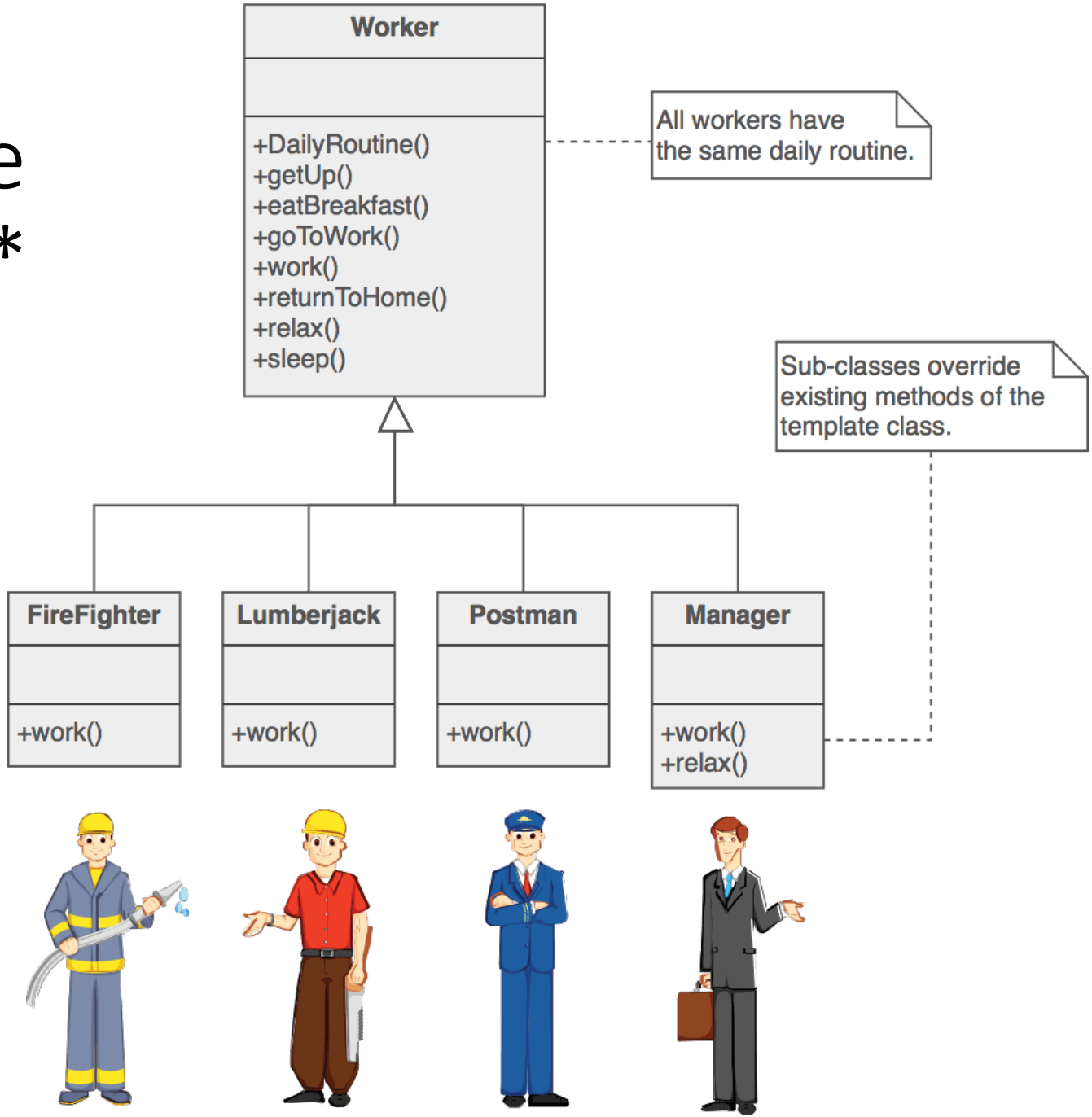
Template Method*



Template Method*



Template Method*



Visitor*

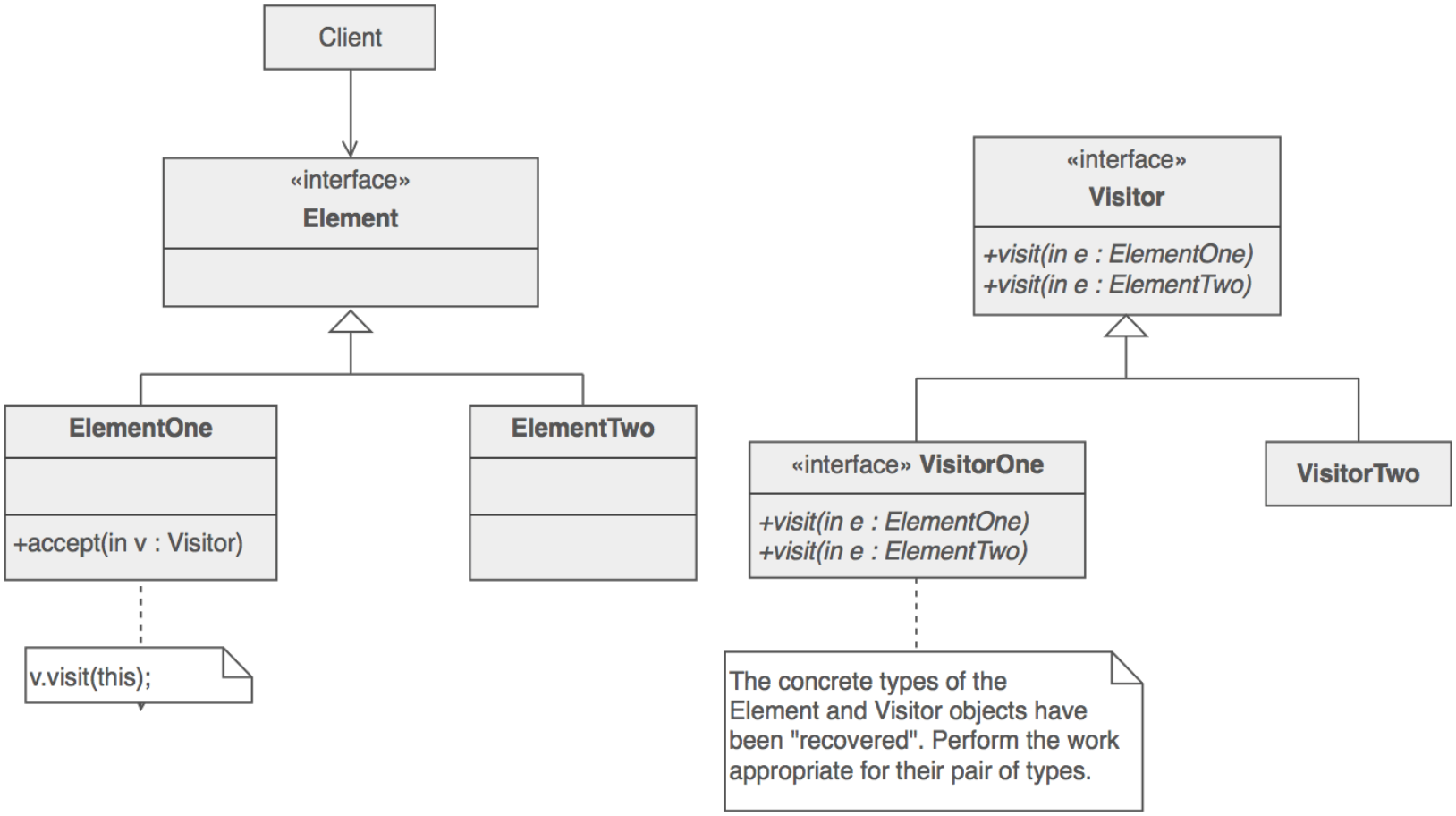
Intent

- Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.
- The classic technique for recovering lost type information.
- Do the right thing based on the type of two objects.
- Double dispatch

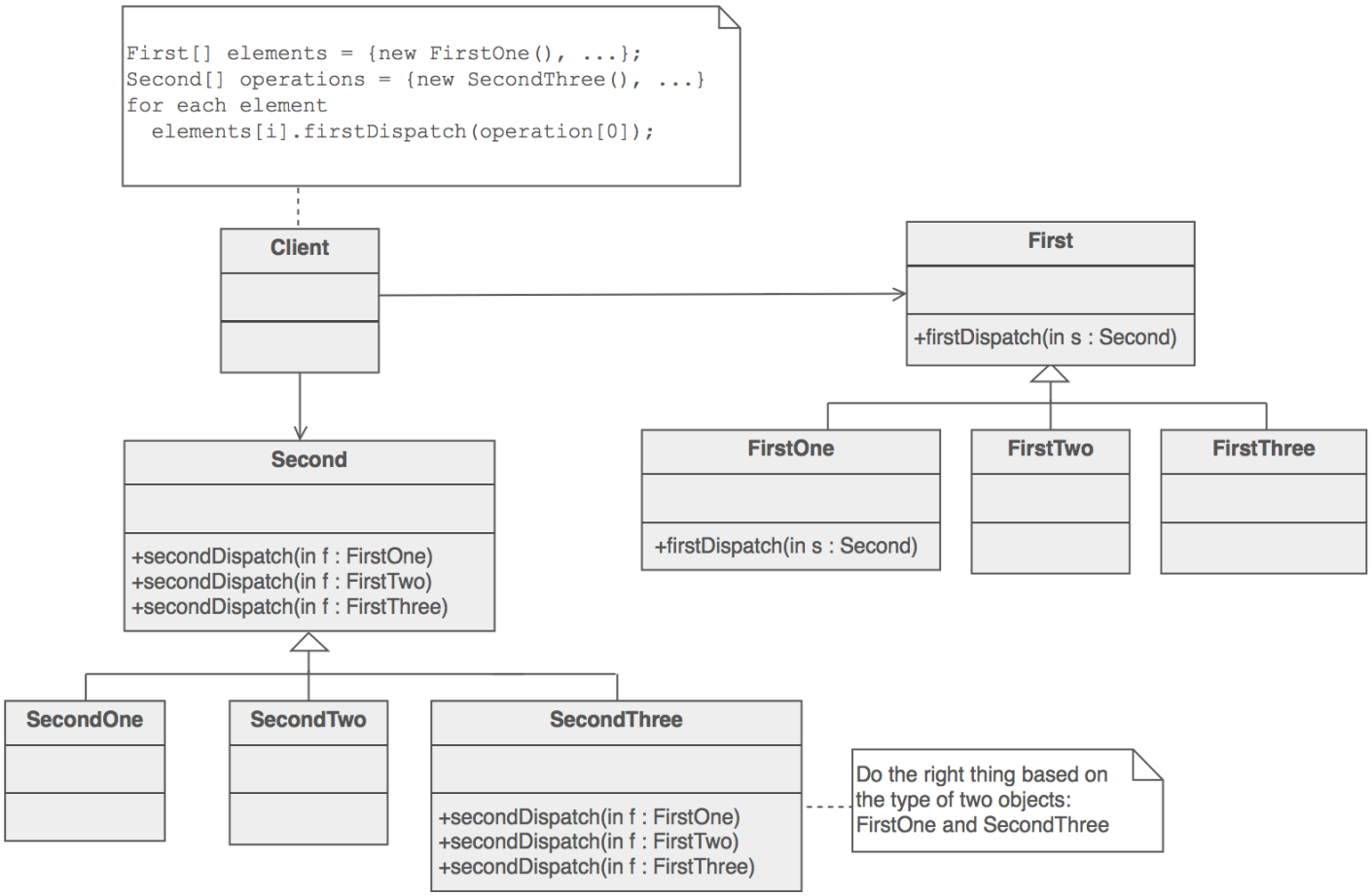
Problem

- Many distinct and unrelated operations need to be performed on node objects in a heterogeneous aggregate structure.
- You want to avoid "polluting" the node classes with these operations
- You don't want to have to query the type of each node and cast the pointer to the correct type before performing the desired operation.

Visitor* (Structure)



Visitor* (Structure)



IMPORTANT DESIGN PATTERNS

Abstract Factory (creation) **

Builder (chained creation)

Singleton (single resource) **

Adapter (making other things work) **

Command (Able to defer dispatch)

Iterator (Traverse through items)

Memento (Preserving State – ties with “Command”)

Observer (notified by state changes, View) **

State (modify behavior, model state machine)

Strategy (family of algorithms; runs one) **

Visitor (add new functionality without changing) **