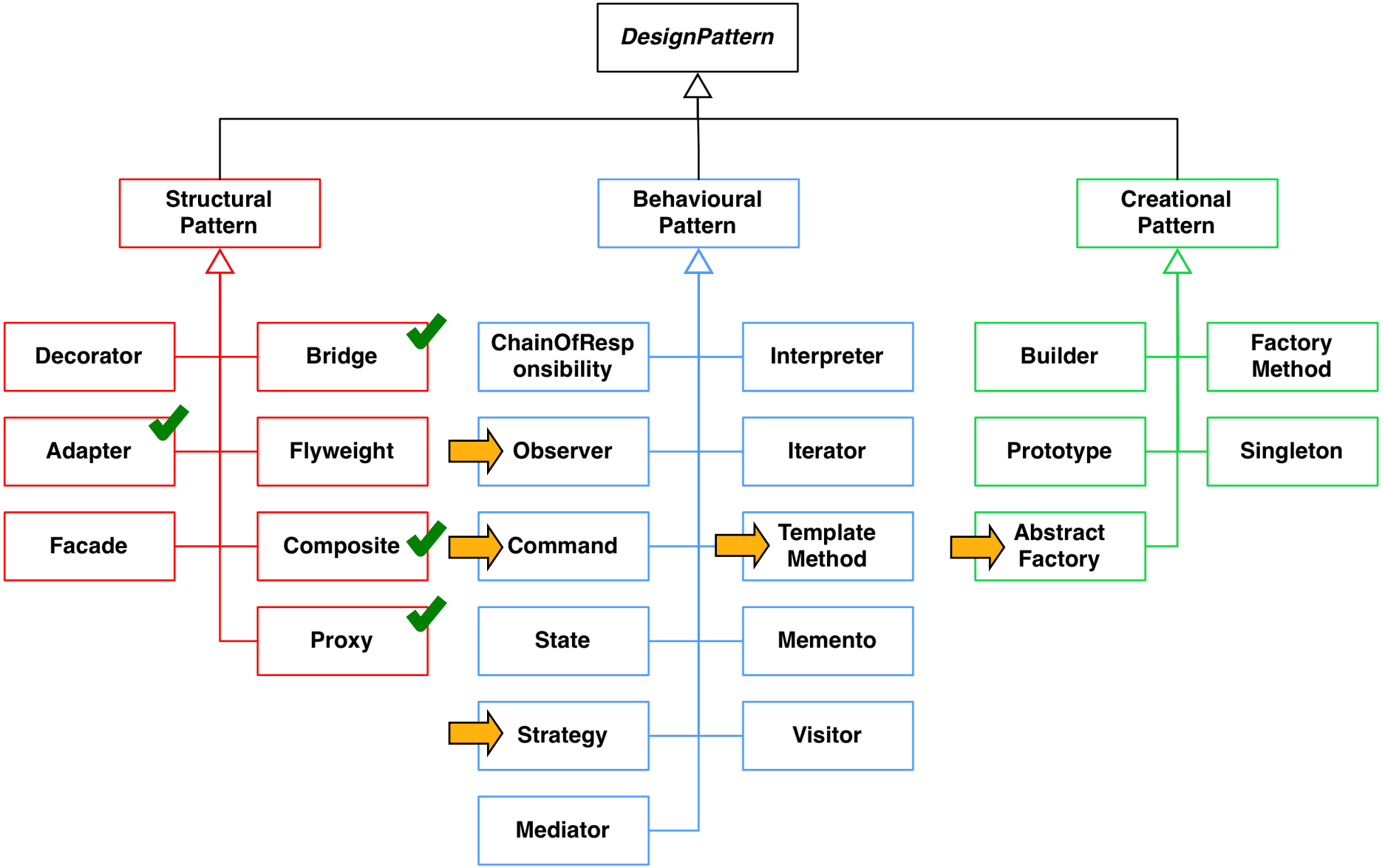
Software Patterns

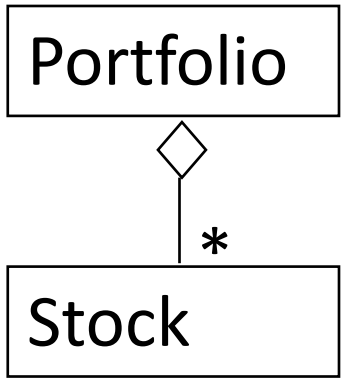
L04: Behavioral and Creational Patterns

Marlo Häring & Prof. W. Maalej (@maalejw)

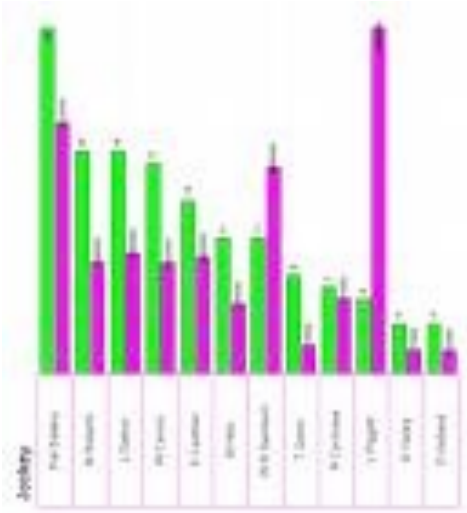
Design patterns taxonomy

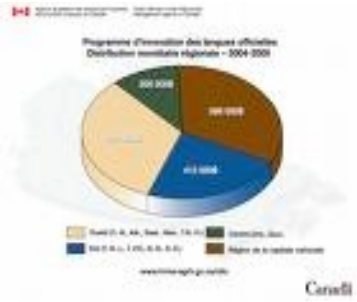


Problem: maintain consistency

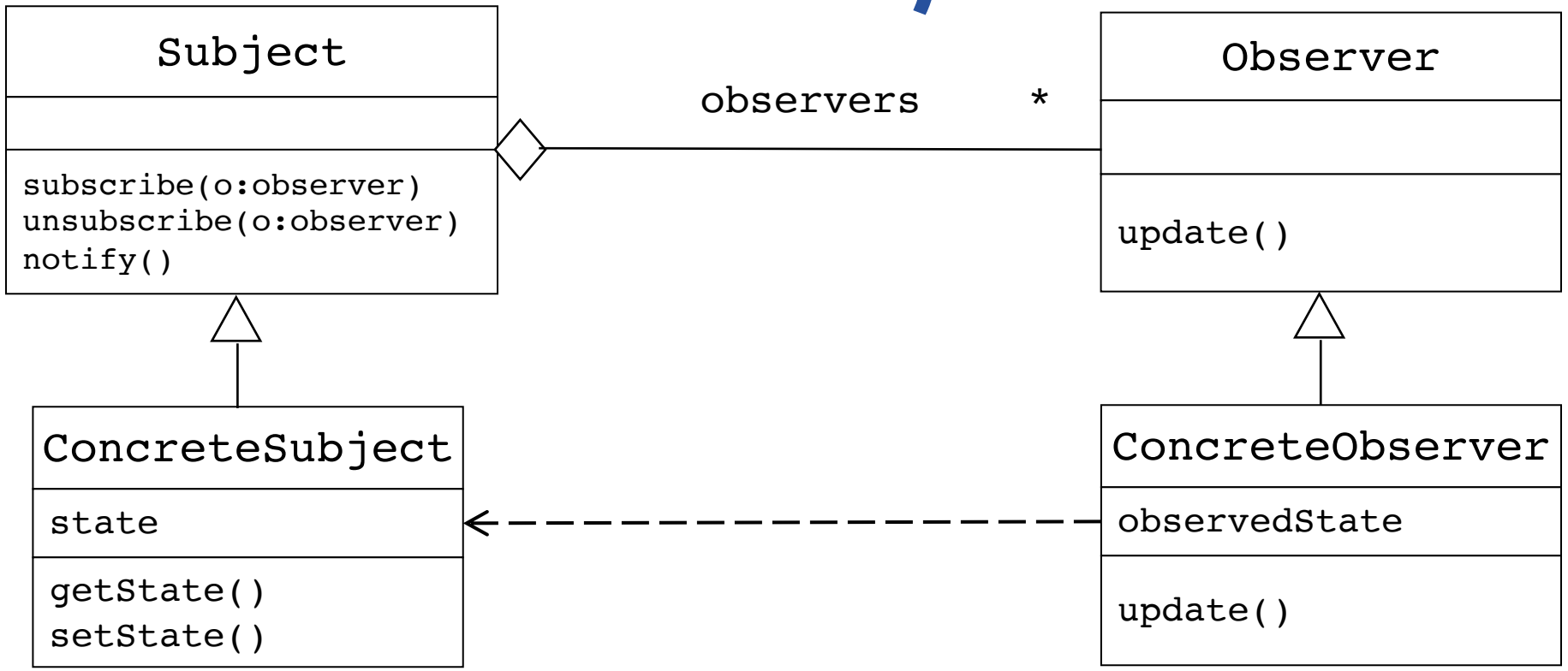


- We have an object that changes its state quite often
  - Example: A Portfolio of stocks
- We want to provide multiple views of the current state of the portfolio
  - Example: Histogram view, pie chart view, timeline view
- Requirements:
  - The system should maintain consistency across the (redundant) views, whenever the state of the observed object changes
  - It should be possible to add new views - for example an alarm - without having to recompile the observed object or the existing views





Solution: observer pattern (decouples an abstraction from its views)



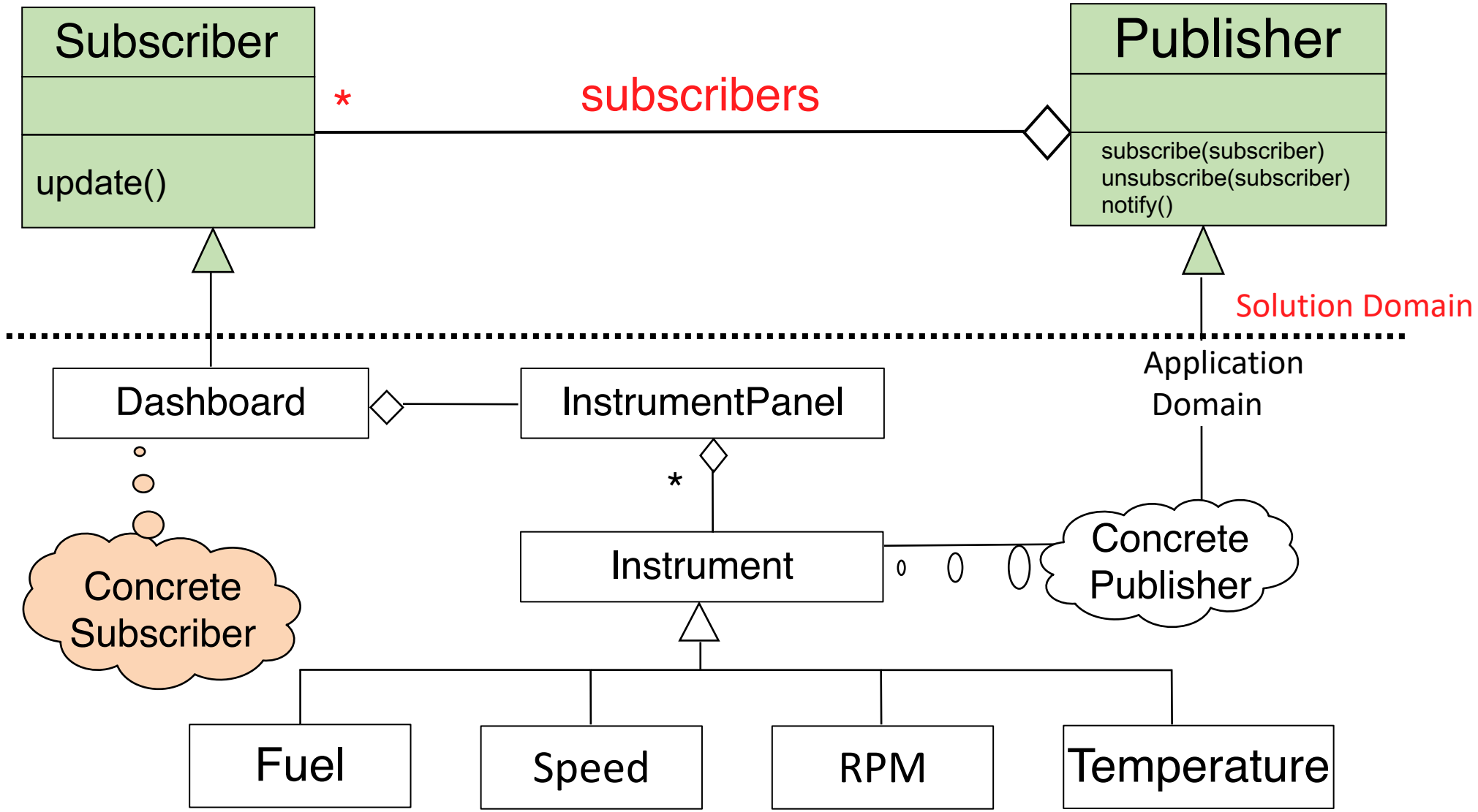
- The Subject represents the entity object
- Observers attach to the Subject by calling subscribe()
- Each Observer has a different view of the state of the entity object
  - The state is contained in the subclass ConcreteSubject
  - The state can be obtained and set by subclasses of type ConcreteObserver

Observer pattern

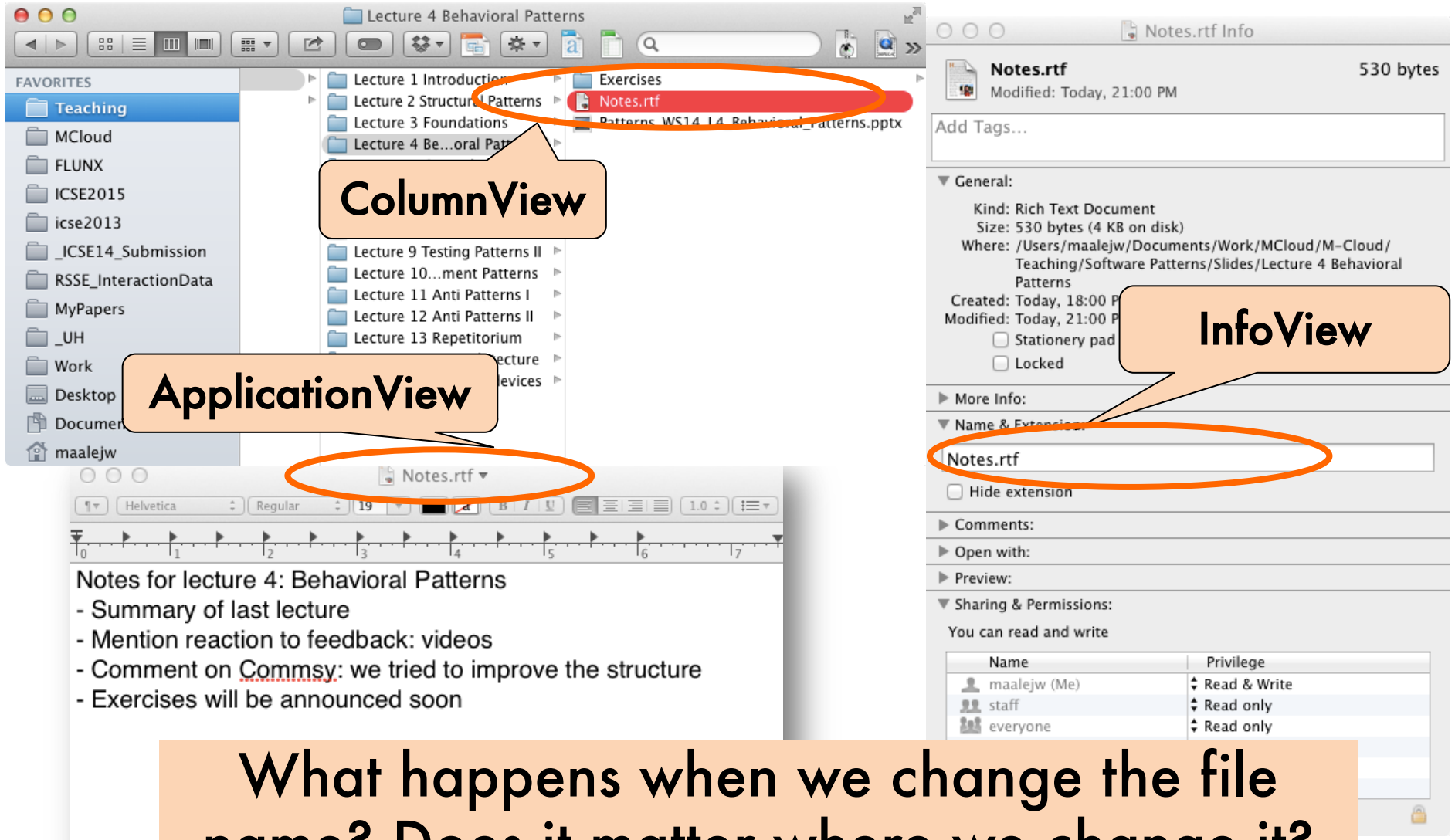
- Models a 1-to-many dependency between objects
  - Connects the state of an observed object, the subject with many observing objects, i.e. the observers
- Usage:
  - Maintaining consistency across redundant states
  - Optimizing a batch of changes to maintain consistency
- Three variants for maintaining the consistency:
  - Push Notification: Every time the state of the subject changes, all the observers are notified of the change
    - Push-Update: The subject sends the state that has been changed to the observers
    - Pull-Update: The object must request the desired data after receiving notification
  - Pull Information: An observer inquires about the state of the subject
- Also called Publish and Subscribe

Observer pattern for a car dashboard

Application domain vs solution domain objects

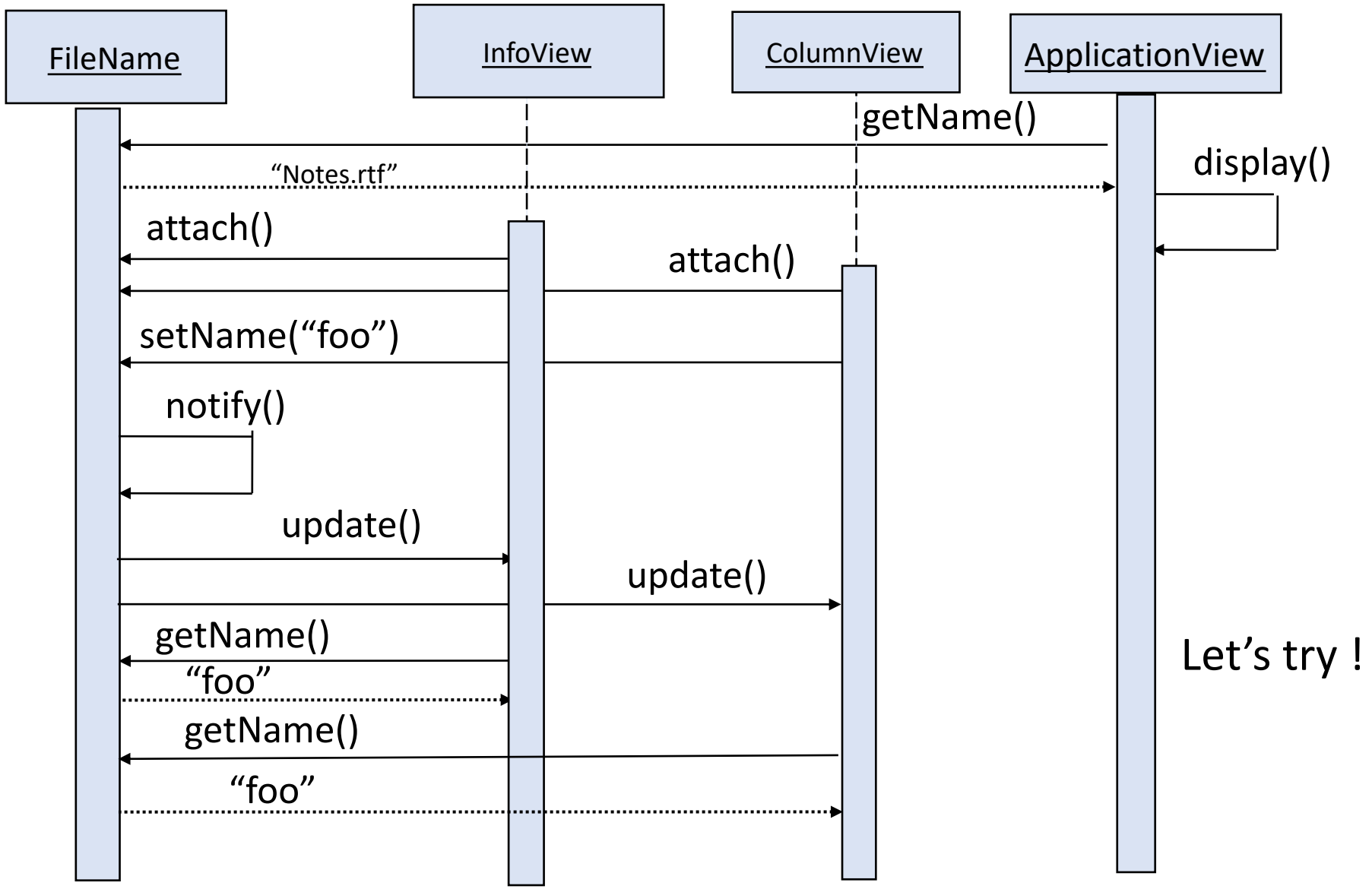


Multiple views of a file

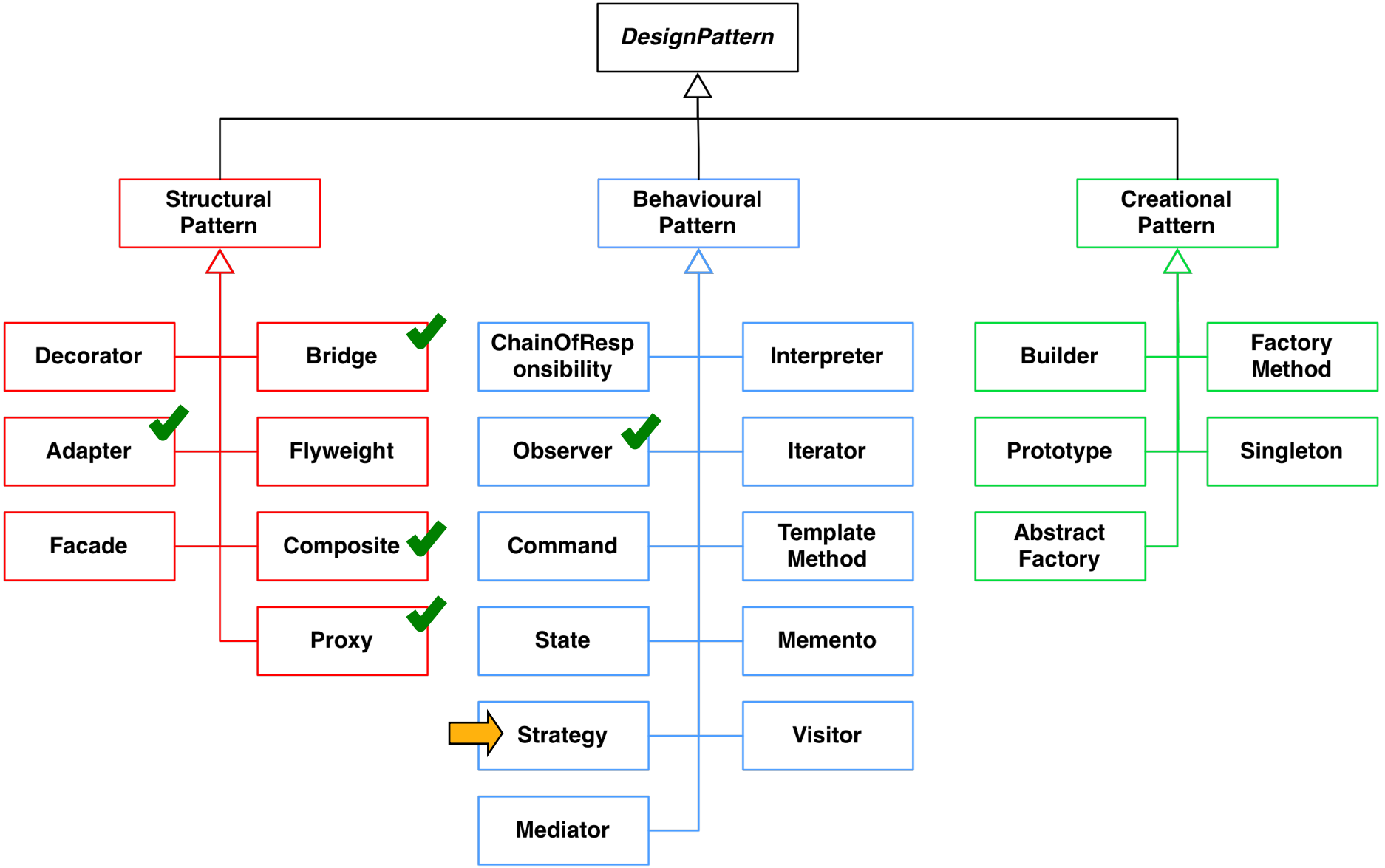


What happens when we change the file name? Does it matter where we change it?

Sequence diagram: change filename to "foo"



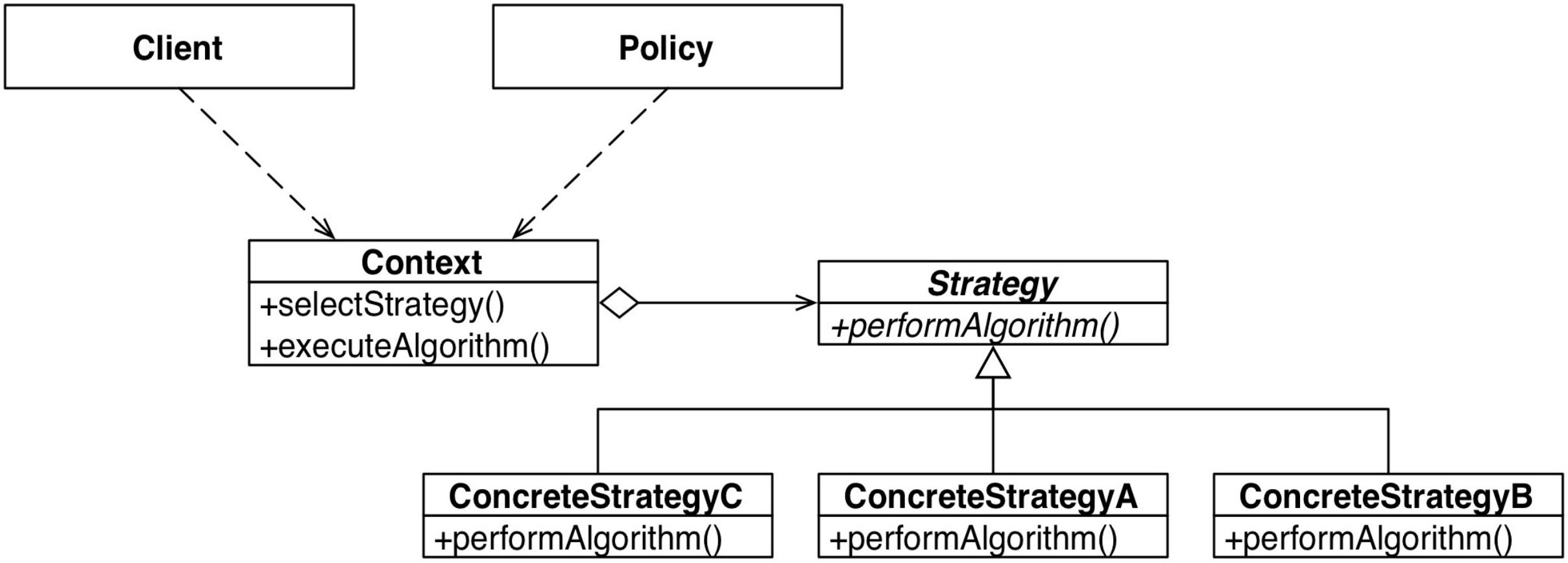
Design patterns taxonomy



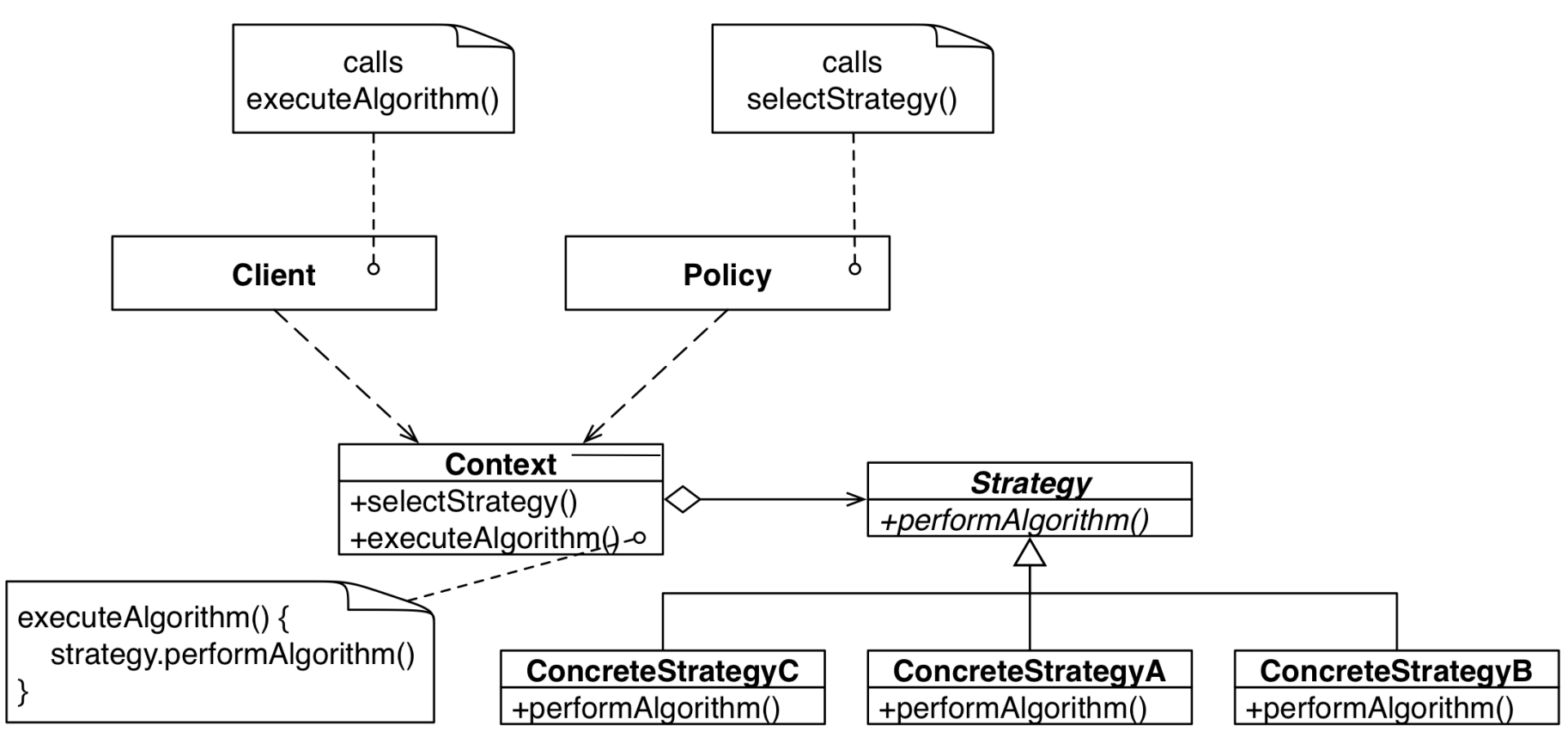
Problem: algorithm selection at runtime

- Different algorithms exists for a specific task
  - We want to switch between these algorithms at run time
- Examples of tasks:
  - Different ways to sort a list (bubblesort, mergesort, quicksort)
  - Different collision strategies for objects in video games
  - Different ways to parse a set of tokens into an abstract syntax tree
- If we need a new algorithm, we want to add it without disturbing the application or the other algorithms

Solution: strategy pattern



Strategy pattern

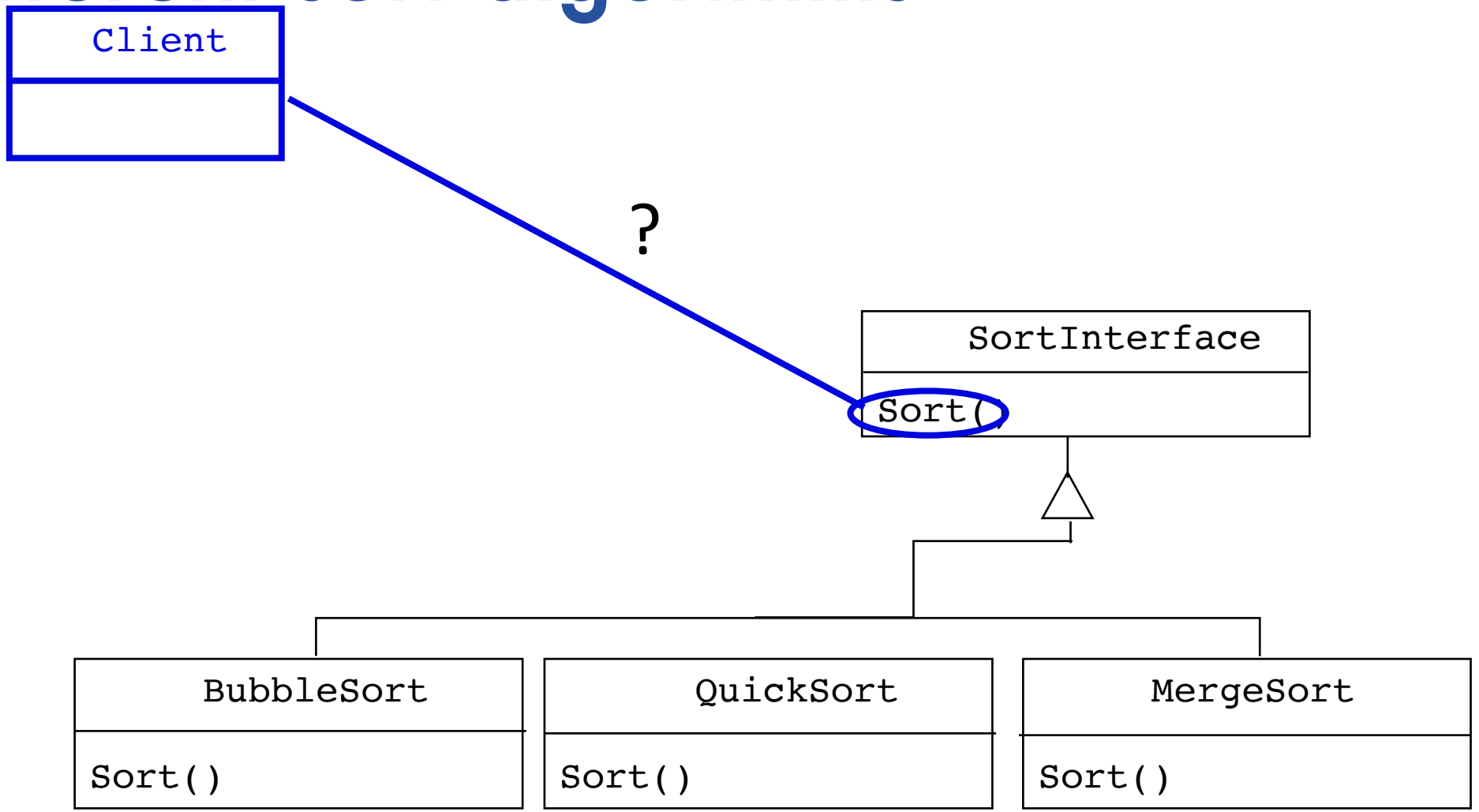


The Policy selects the Strategy in Context. This determines which ConcreteStrategy is executed, when the Client calls executeAlgorithm()

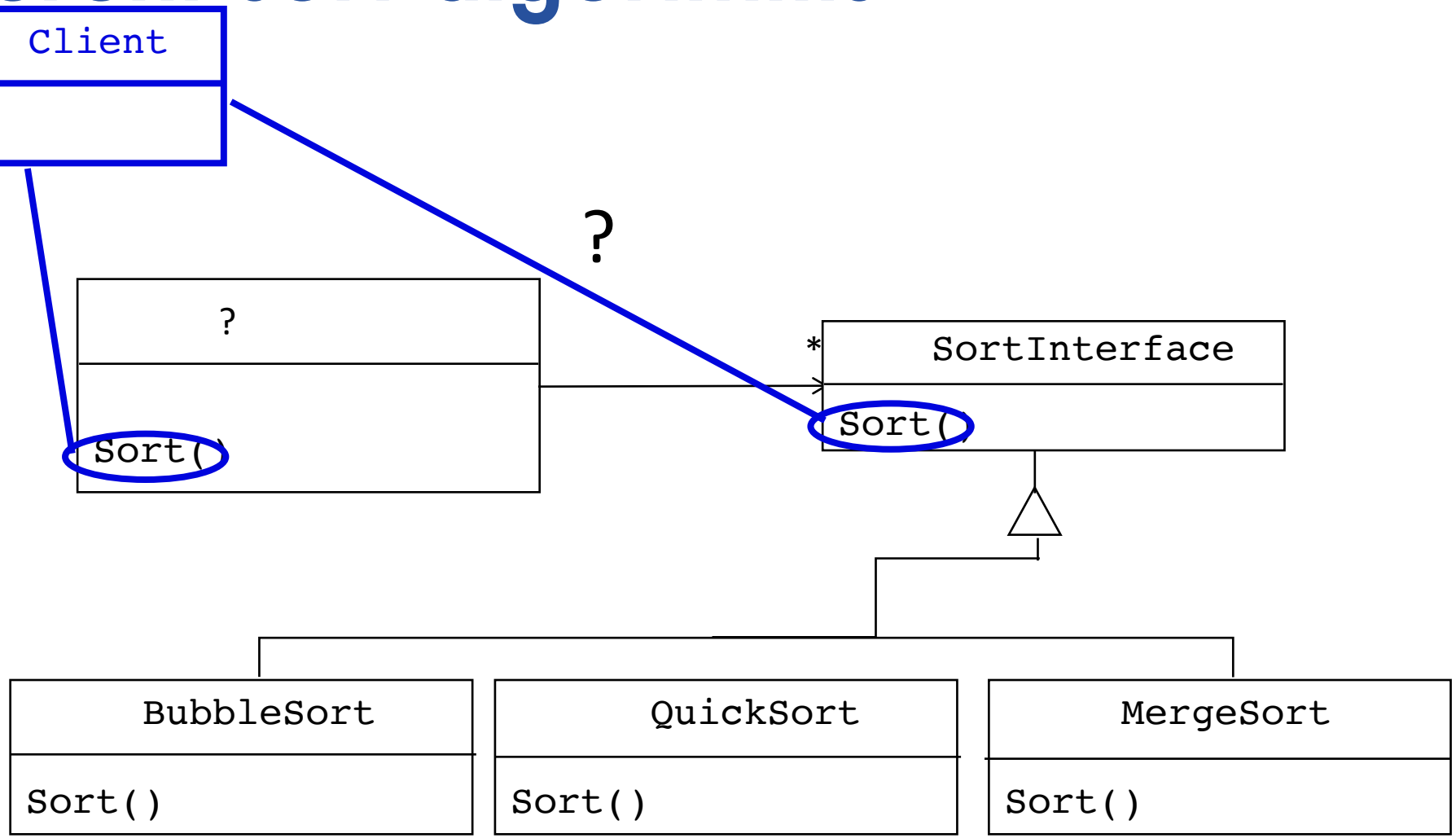
Applicability of strategy pattern

- Many related classes differ only in their behavior
- Different *variants of an algorithm* are needed that trade-off space against time
- A specific implementation needs to be selected based on the current context *at runtime*.

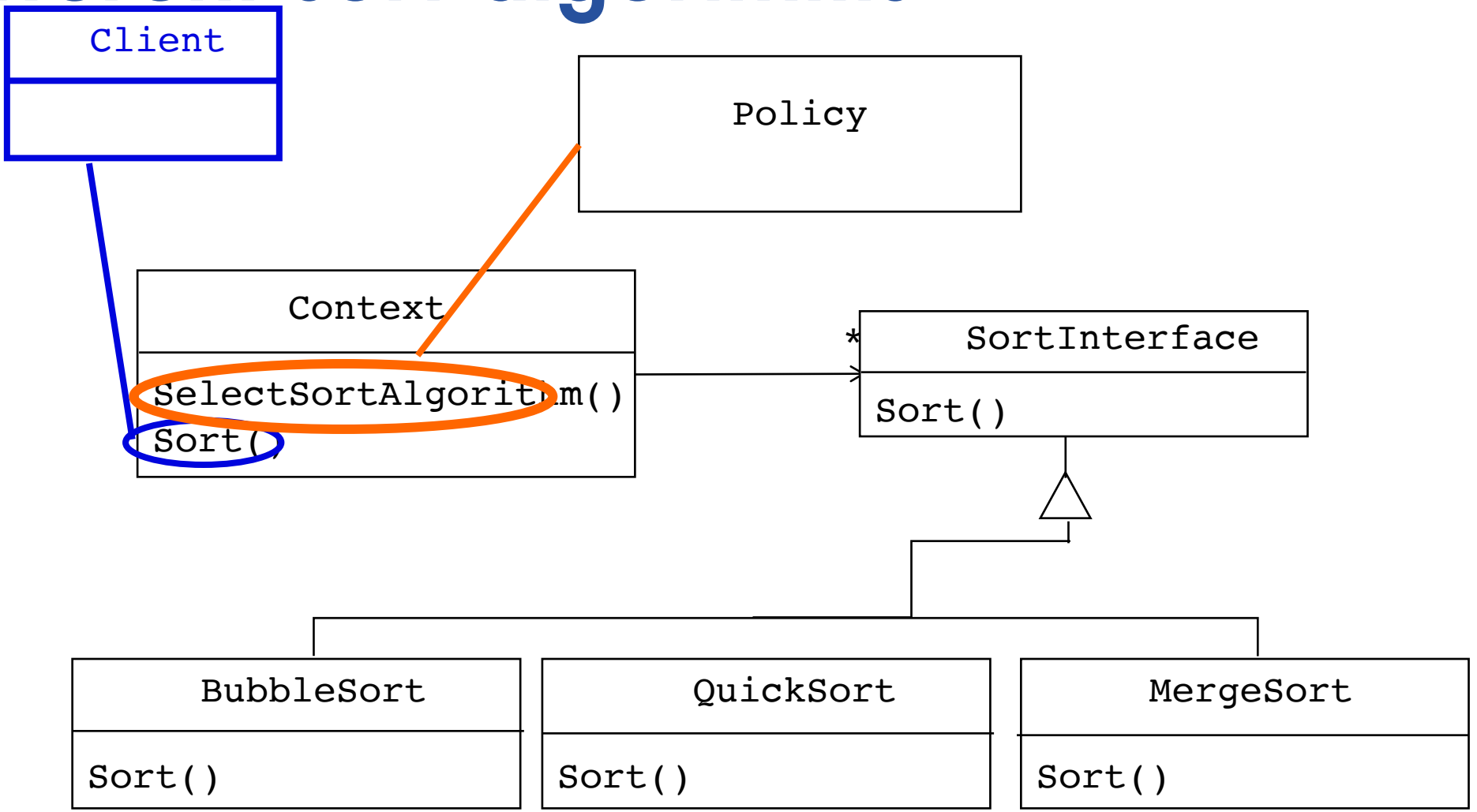
Using a strategy pattern to switch between 3 different sort algorithms



Using a strategy pattern to switch between 3 different sort algorithms



Using a strategy pattern to switch between 3 different sort algorithms



Rapid Development Time is important

Fast Execution Time is important

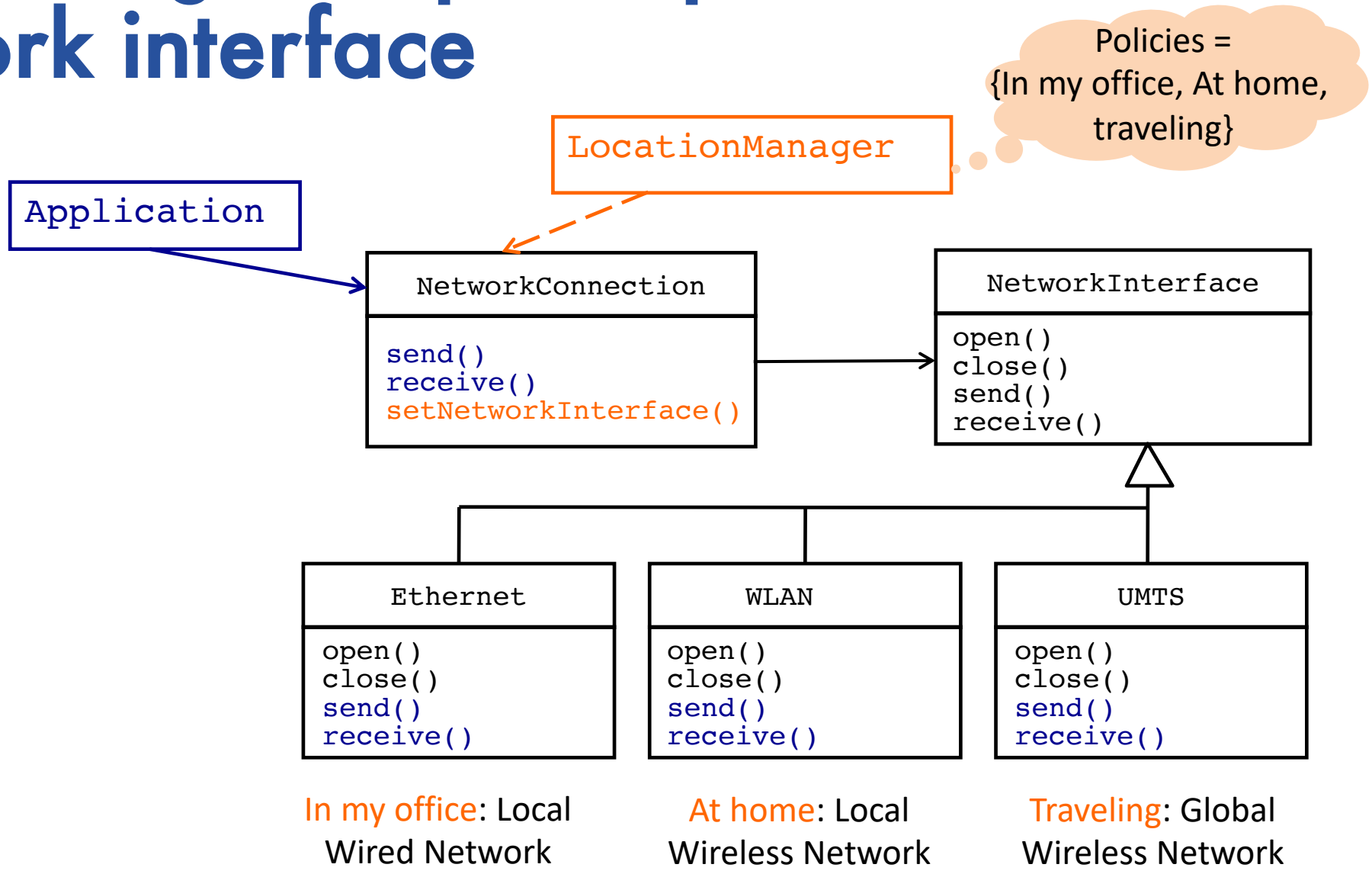
Efficient Use of Memory is important

Policy class

```
public class Policy {
        private Database db;
        private boolean timeIsImportant = true;
        private boolean spaceIsImportant = false;

        public Policy (Database db) { this.db = db; }
        ...
        // check status of database and adapt sort algorithm
        public void checkPolicy() {
                checkDB();   // check status of database
                if (timeIsImportant && !spaceIsImportant)
                        db.selectSortAlgorithm(new MergeSort());
                else if (timeIsImportant && spaceIsImportant)
                        db.selectSortAlgorithm(new QuickSort());
        }
}
```

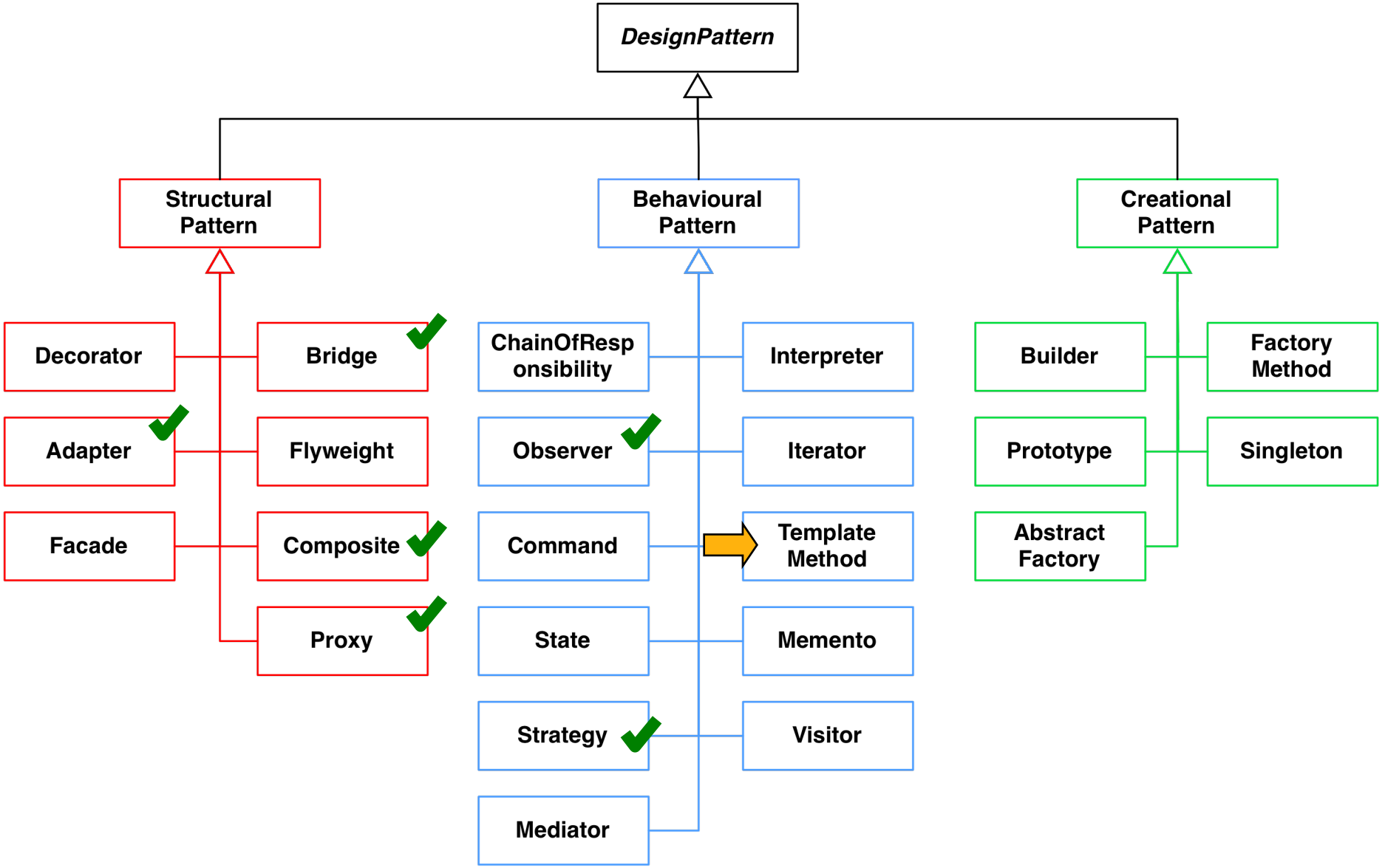
Supporting multiple implementations of a network interface



Strategy vs. Bridge

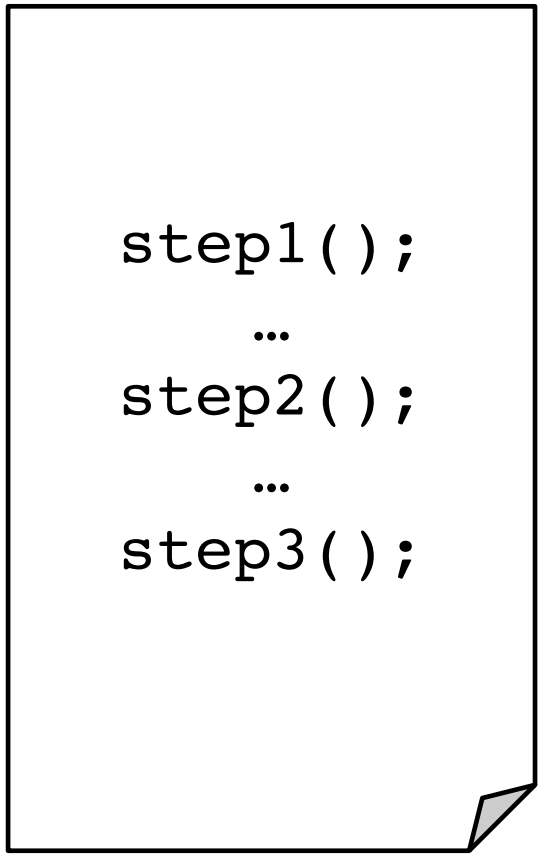
- Behavioral pattern vs. structural pattern
- Bridge: design decision
  - Depending on client an implementation is chosen at start up
- Strategy: interchangeable algorithms
  - Depending on the policy, a specific algorithm is chosen at runtime
  - These algorithms vary independently from clients using it

Design patterns taxonomy



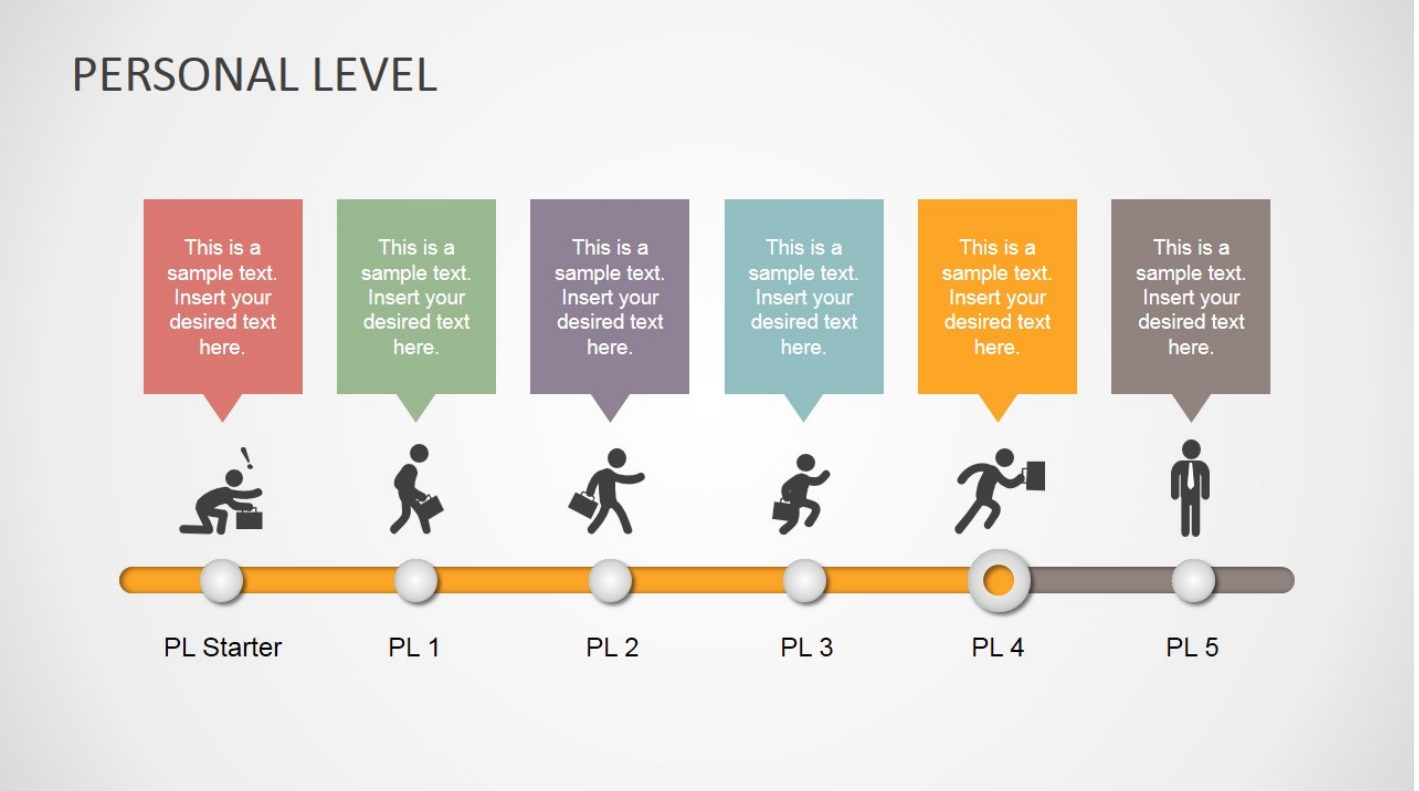
Problem: common behavior with some differences

- Several subclasses share the same workflow/algorithm but differ on the specifics
  - Common steps should not be duplicated in the subclasses
  - Examples:
    - Opening, reading, writing and closing documents of different types
    - Executing a test suite of test cases

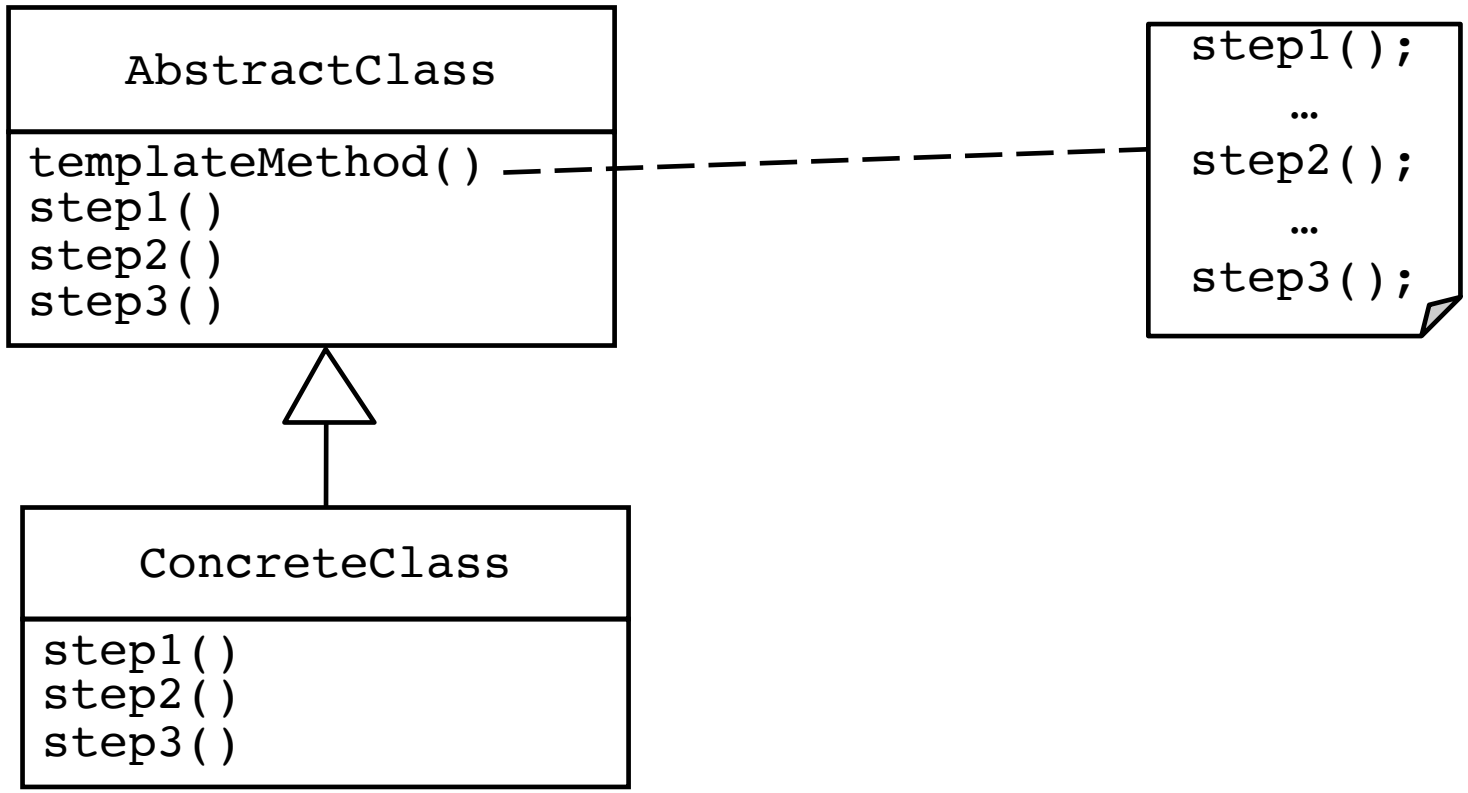


Solution: template pattern

- The common steps (step1, step2, step3,…) of the algorithm are factored out into an abstract class
  - Abstract methods are specified for each of these steps
- Subclasses provide different realizations for each step

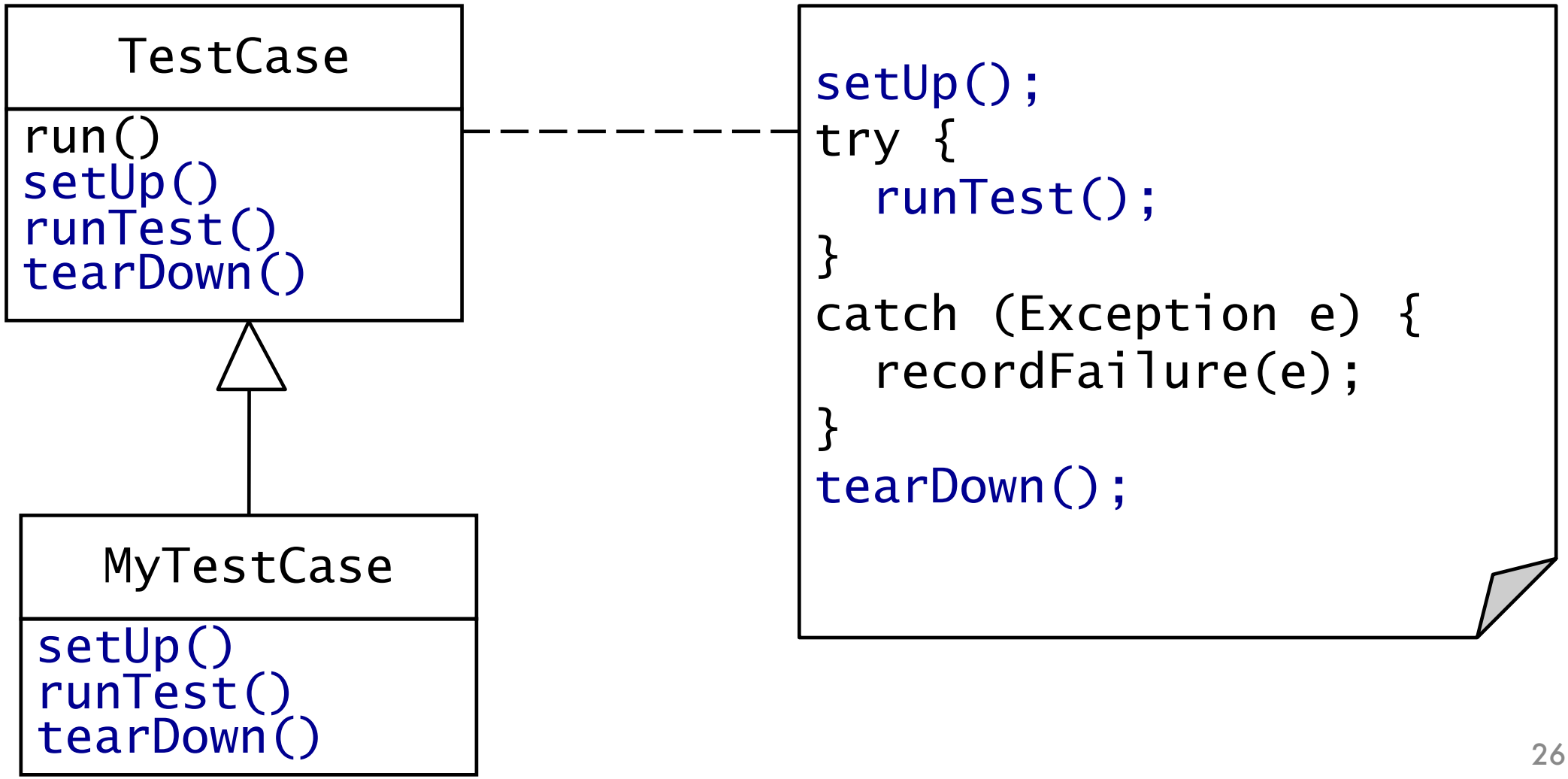


Template method pattern

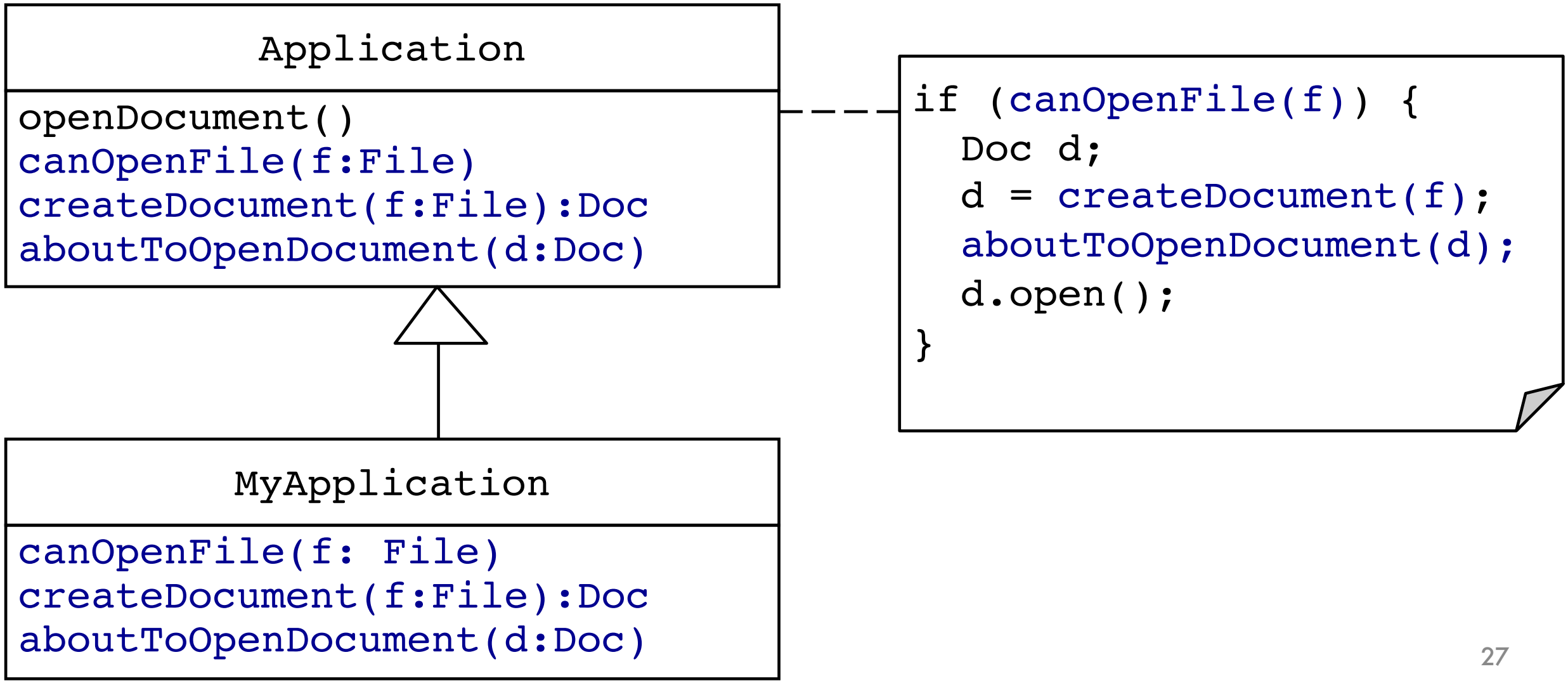


- The common steps (step1, step2, step3,…) of the algorithm are factored out into an abstract class
  - Abstract methods are specified for each of these steps
- Subclasses provide different realizations for each of these steps

Example: modeling test cases with the template method pattern



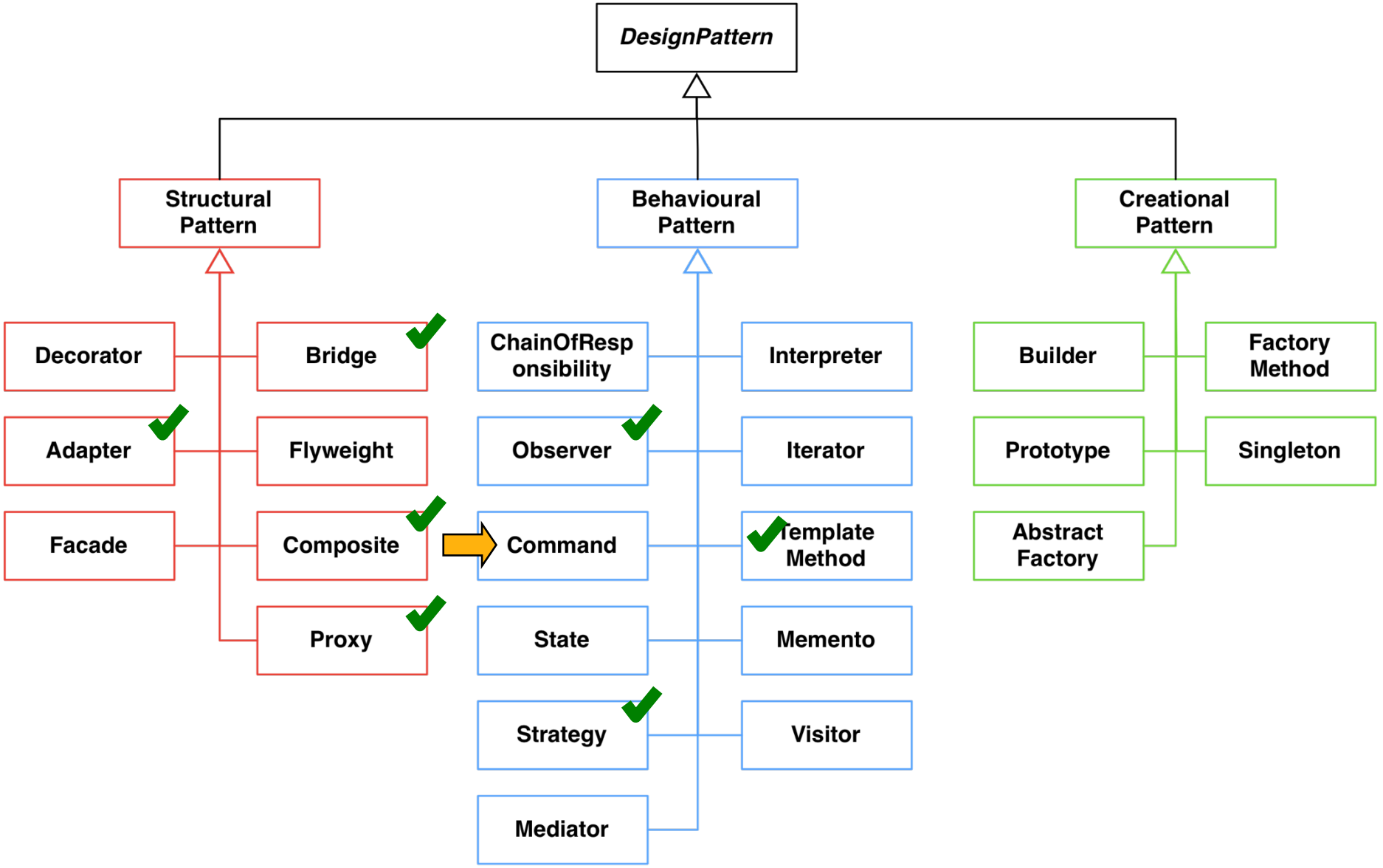
Example: modeling opening documents with the template method pattern



Template method pattern applicability

- Template method pattern uses inheritance to vary part of an algorithm
  - Difference to Strategy Pattern: The strategy pattern uses delegation to vary the entire algorithm
- The template method pattern is primarily used in frameworks
  - The framework implements the invariants of the algorithm
  - The client customizations provide specialized steps for the algorithm
  - Principle: "Don't call us (the framework), we'll call you (the client)"

Design patterns taxonomy



Example: building a command interpreter

- The ugly way:

```
String command;
command = getUserInput();
if Equals(command, “File”) then
        FileCommand()
else if Equals(command, “Edit”) then
        EditCommand()
else if Equals(command, “Help”) then
        HelpCommand()
…..
```

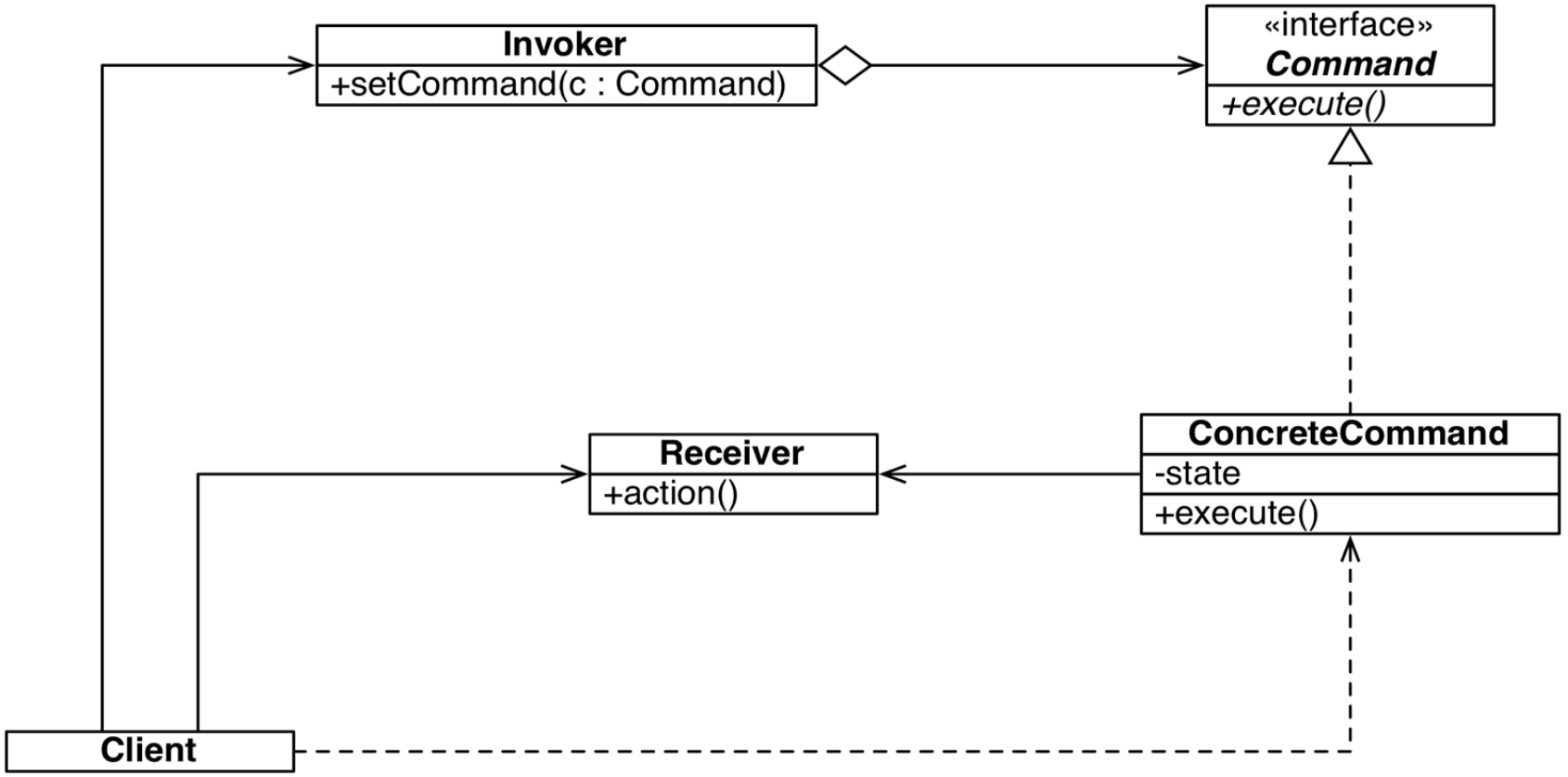
- Is there a better way?

Command pattern: motivation

- You want to build a graphical user interface with menus
- Bad: You hardcode the menu commands
- Good:
  - Create a command taxonomy
  - Map the taxonomy into a menu structure
  - Let the application know, when a command from the menu is selected
- Such a user interface can easily be implemented with the Command Pattern
  - This makes menus reusable across many applications

Command pattern

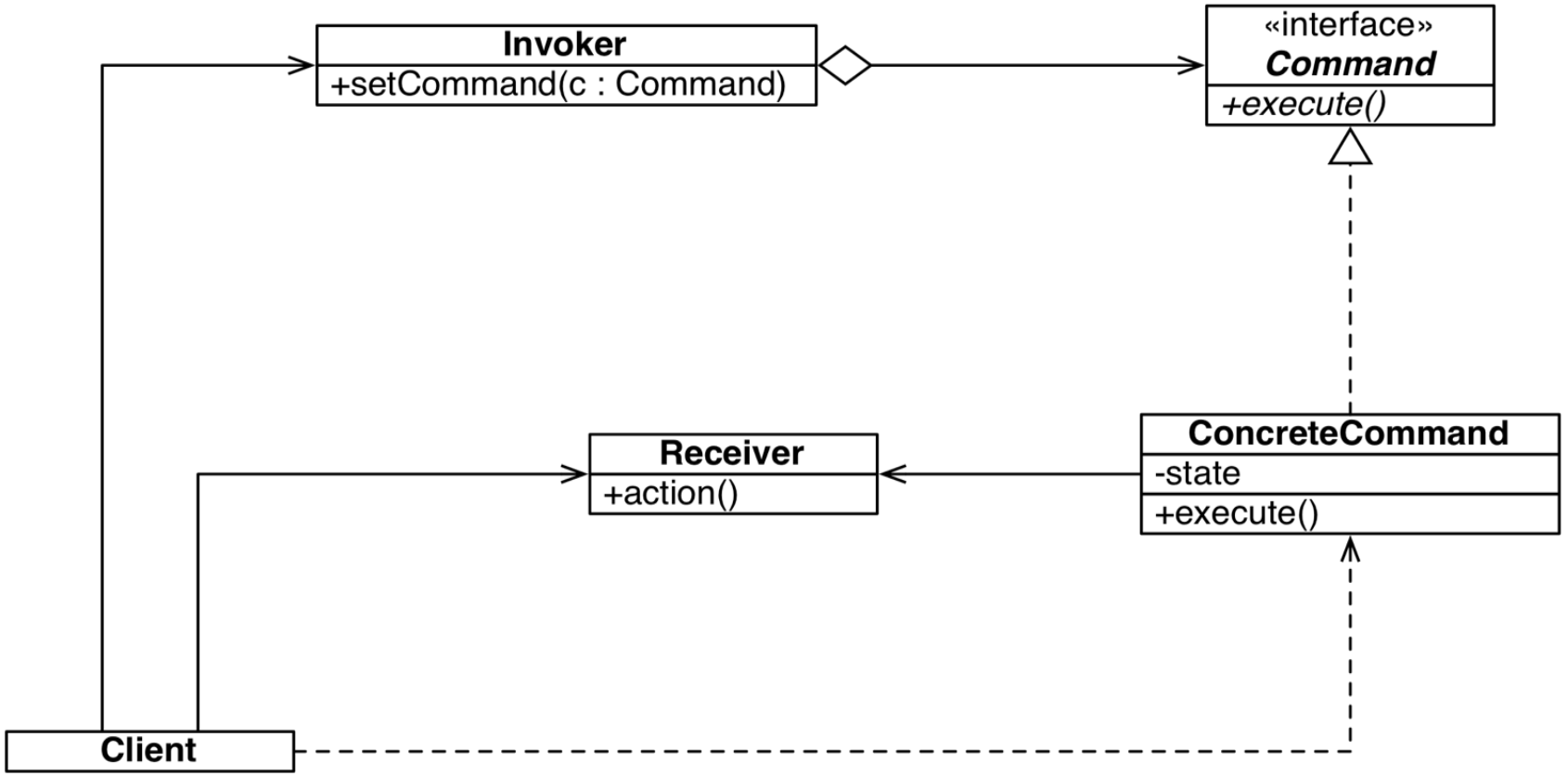
- Client is a class in a user interface builder or in a class executing during startup of the application to build the user interface
- Client creates instances of ConcreteCommand for "file", "edit", "help", etc. and binds them to specific actions in the corresponding Receiver



- All concrete commands are sub-classes of the abstract class Command

Command pattern (ctd)

- When a user clicks a button or a menu entry, the associated information is passed from the **Invoker** to the **ConcreteCommand** by invoking execute(), which calls the action() method in the **Receiver** object, which contains the code that does the work
- The **Command** object can notify the **Invoker** object when the availability of a command/action has changed
  - This allows buttons and menu items to become inactive



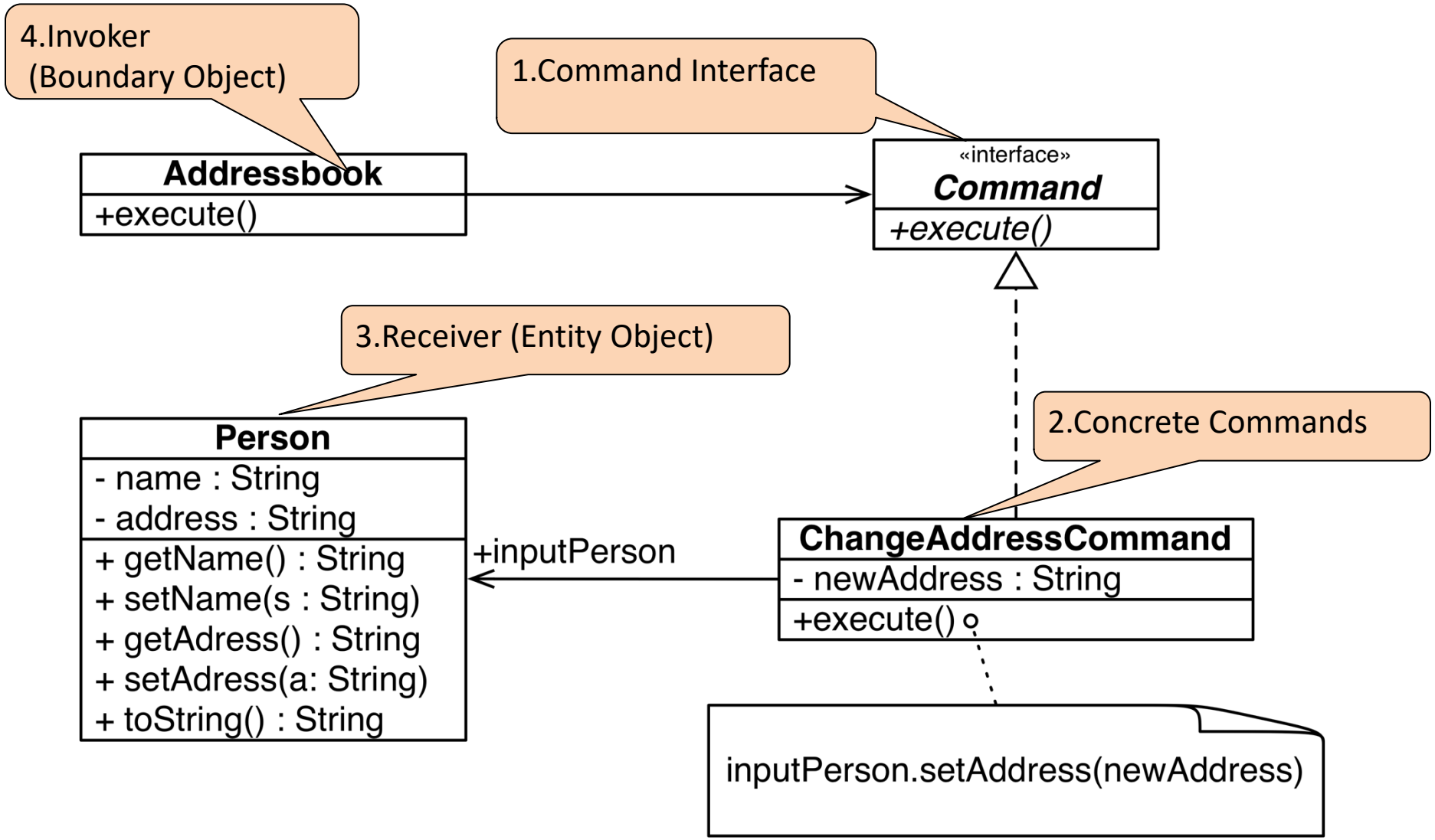
Invoker in the command pattern

- Can be used in a variety of tasks typical for interactive and real-time systems:
- Command Manager
  - Central repository for all the commands available for an application („Client“)
  - Performs operations based on the commands and responds to command events
  - Application can dynamically add/remove commands from the manager
- Redo/Undo Manager
  - Pushes the command or events on a stack for possible redo/undo operations issued by the application
- Queue
  - Holds command or events until other objects are ready to do something with them. Useful for schedulers, etc.
- Dispatcher (also called Callback Handler)
  - Routes commands or events to the appropriate receiver that deals with the command or event.
  - Example: Keyboard event loop in an interactive system

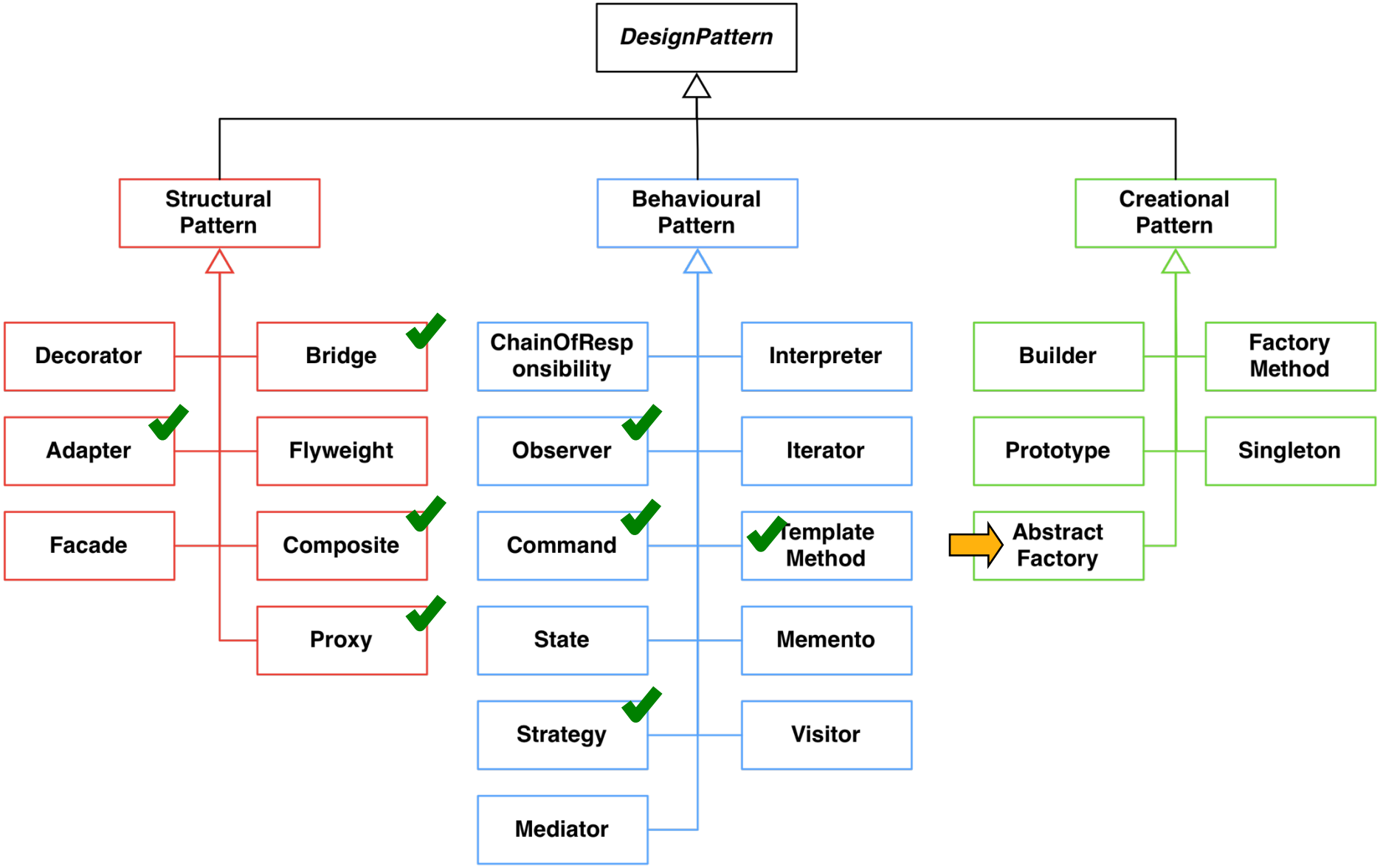
Command pattern decouples boundary and control objects

- Decouples boundary objects from control objects and separates them completely from entity objects
  - Boundary objects such as menu items and buttons, send messages to the control objects (subclasses of type command)
  - Only ConcreteCommand objects can modify entity objects
- If the user interface is changed only the boundary objects need to be modified, but not the control and entity objects

UML model for an address book application



Design patterns taxonomy



Example: using a factory class

- Consider the following implementation of a PizzaStore:

```
1  public class PizzaStore {
2      public Pizza orderPizza(String type) {
3          Pizza pizza = null;
4          if (type.equals("cheese")) {
5              pizza = new CheesePizza();
6          } else if (type.equals("pepperoni")) {
7              pizza = new PepperoniPizza();
8          } else if (type.equals("veggie")) {
9              pizza = new VeggiePizza();
10         }
11         pizza.bake();
12         return pizza;
13     }
14 }
```

- What are the problems?
  - Creation of pizzas and baking of pizzas is coupled too tightly
- How can we minimize the coupling?

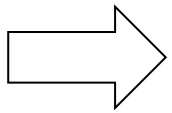
Detour: factory pattern

- Good object-oriented design means high coherence and low coupling
  - Using the new operator creates a dependency on a concrete class, which means high coupling
  - From SOLID, we know that it is better to program against super classes (e.g. Java interfaces) than against concrete implementations

- How can we instantiate new objects without depending on concrete implementations?
  - One possibility is the use of a factory class
- Factory class: a class responsible for the instantiation of objects

Decoupling by use of a factory class

```
public class PizzaStore {
    public Pizza orderPizza(String type) {
        Pizza pizza = null;
        if (type.equals("cheese")) {
            pizza = new CheesePizza();
        } else if (type.equals("pepperoni")) {
            pizza = new PepperoniPizza();
        } else if (type.equals("veggie")) {
            pizza = new VeggiePizza();
        }
        pizza.bake();
        return pizza;
    }
}
```



```
public class PizzaStore {
    PizzaFactory factory;
    public PizzaStore(PizzaFactory factory) {
        this.factory = factory;
    }
    public Pizza orderPizza(String type) {
        Pizza pizza = factory.createPizza(type);
        pizza.bake();
        return pizza;
    }
}
```

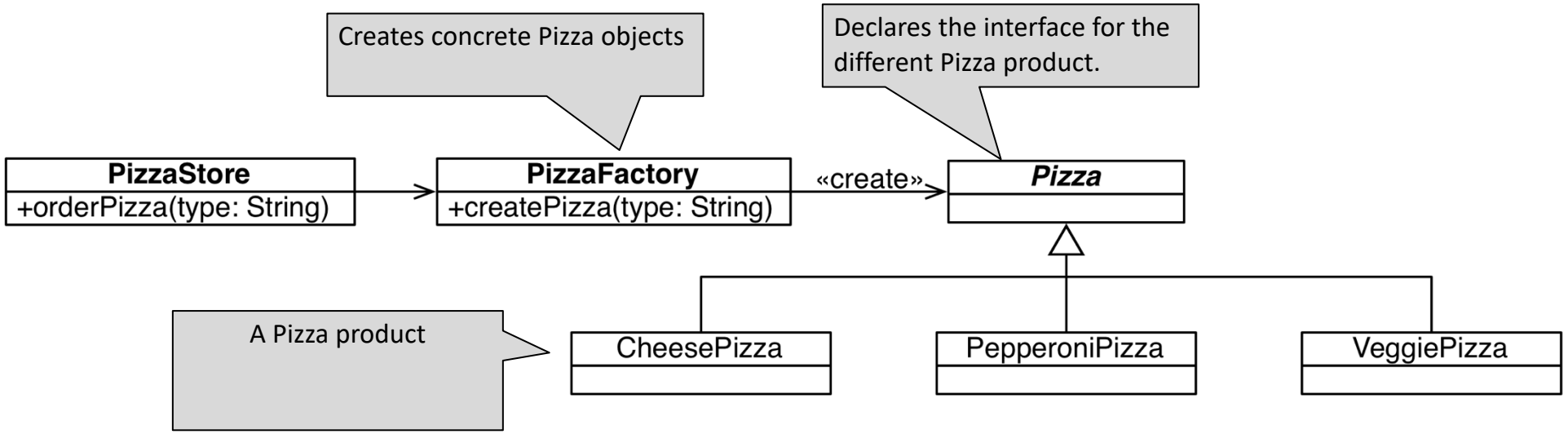
```
public class PizzaFactory {
    public Pizza createPizza(String type) {
        Pizza pizza = null;
        if (type.equals("cheese")) {
            pizza = new CheesePizza();
        } else if (type.equals("pepperoni")) {
            pizza = new PepperoniPizza();
        } else if (type.equals("veggie")) {
            pizza = new VeggiePizza();
        }
        return pizza;
    }
}
```

Example of source code refactoring

UML model for the pizza factory

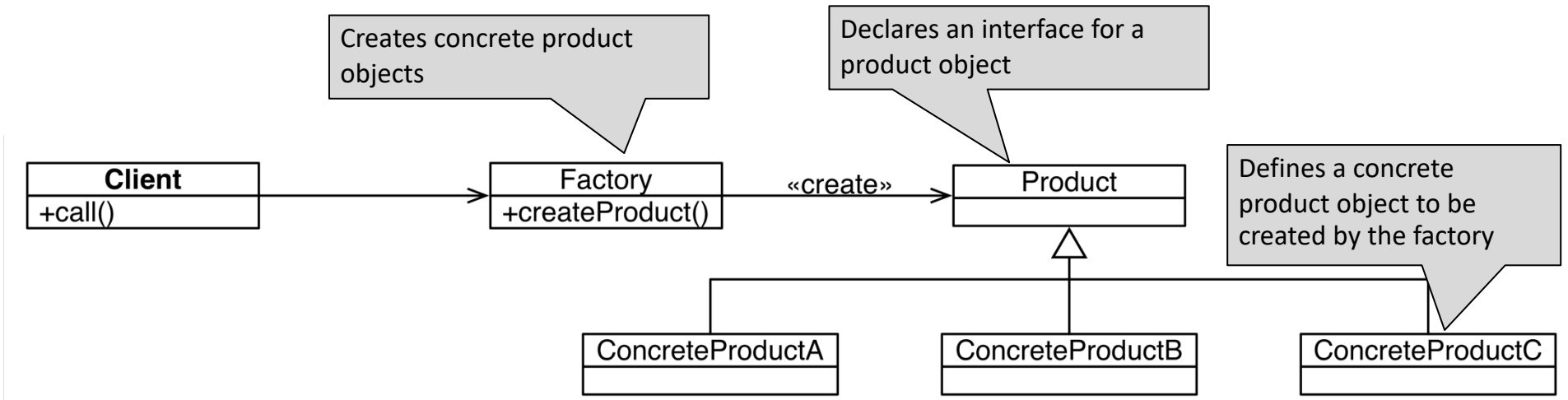
```
public class PizzaStore {
    PizzaFactory factory;
    public PizzaStore(PizzaFactory factory) {
        this.factory = factory;
    }
    public Pizza orderPizza(String type) {
        Pizza pizza = factory.createPizza(type);
        pizza.bake();
        return pizza;
    }
}
```

```
public class PizzaFactory {
    public Pizza createPizza(String type) {
        Pizza pizza = null;
        if (type.equals("cheese")) {
            pizza = new CheesePizza();
        } else if (type.equals("pepperoni")) {
            pizza = new PepperoniPizza();
        } else if (type.equals("veggie")) {
            pizza = new VeggiePizza();
        }
        return pizza;
    }
}
```



What does a factory class do?

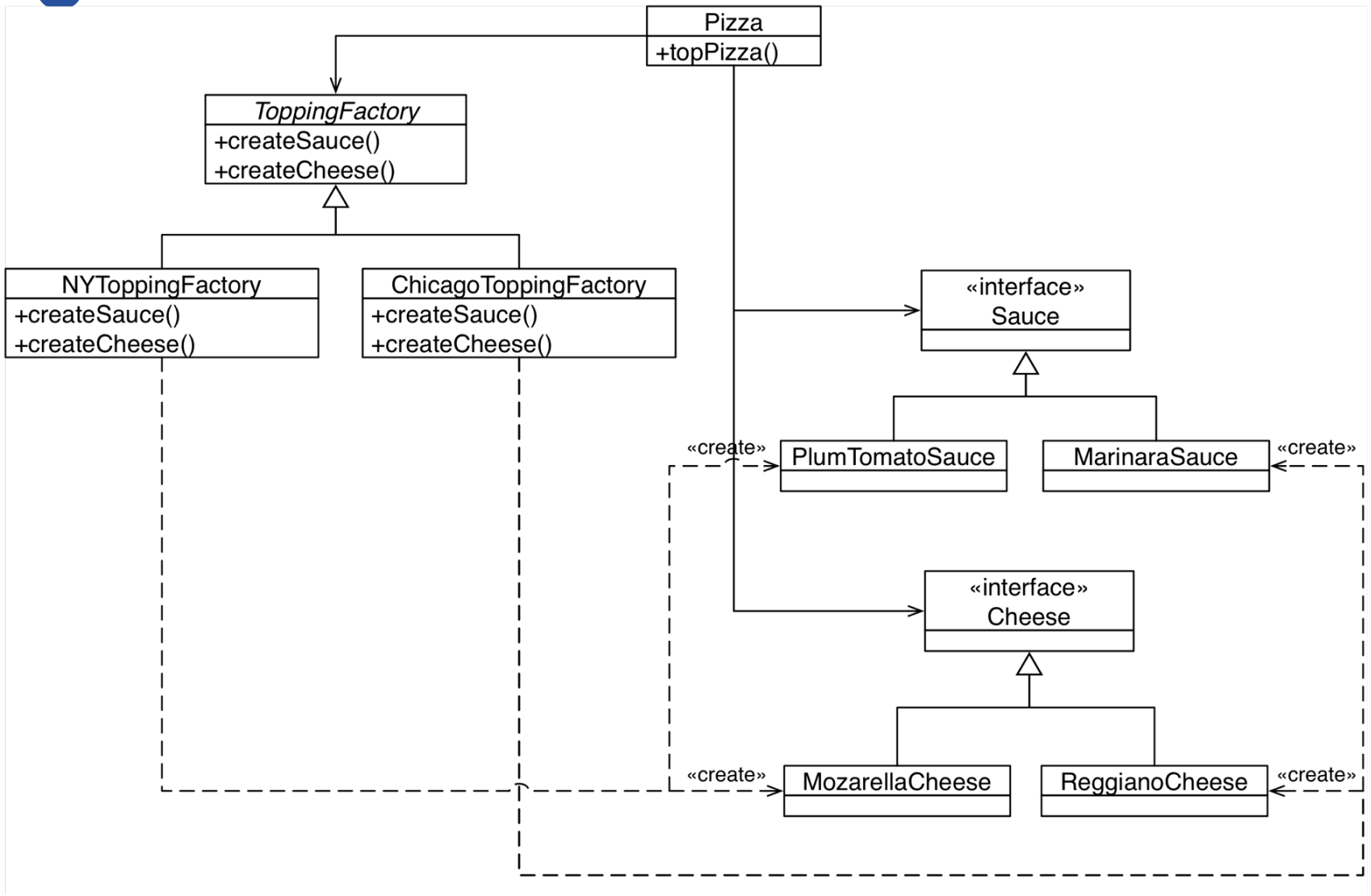
- Acts as delegate for the creation of products
- Allows the Client to use a single interface to the products of a factory
- Makes it is easy to add additional products (extensibility)
- Polymorphism allows the client to use each of the concrete products in a uniform way



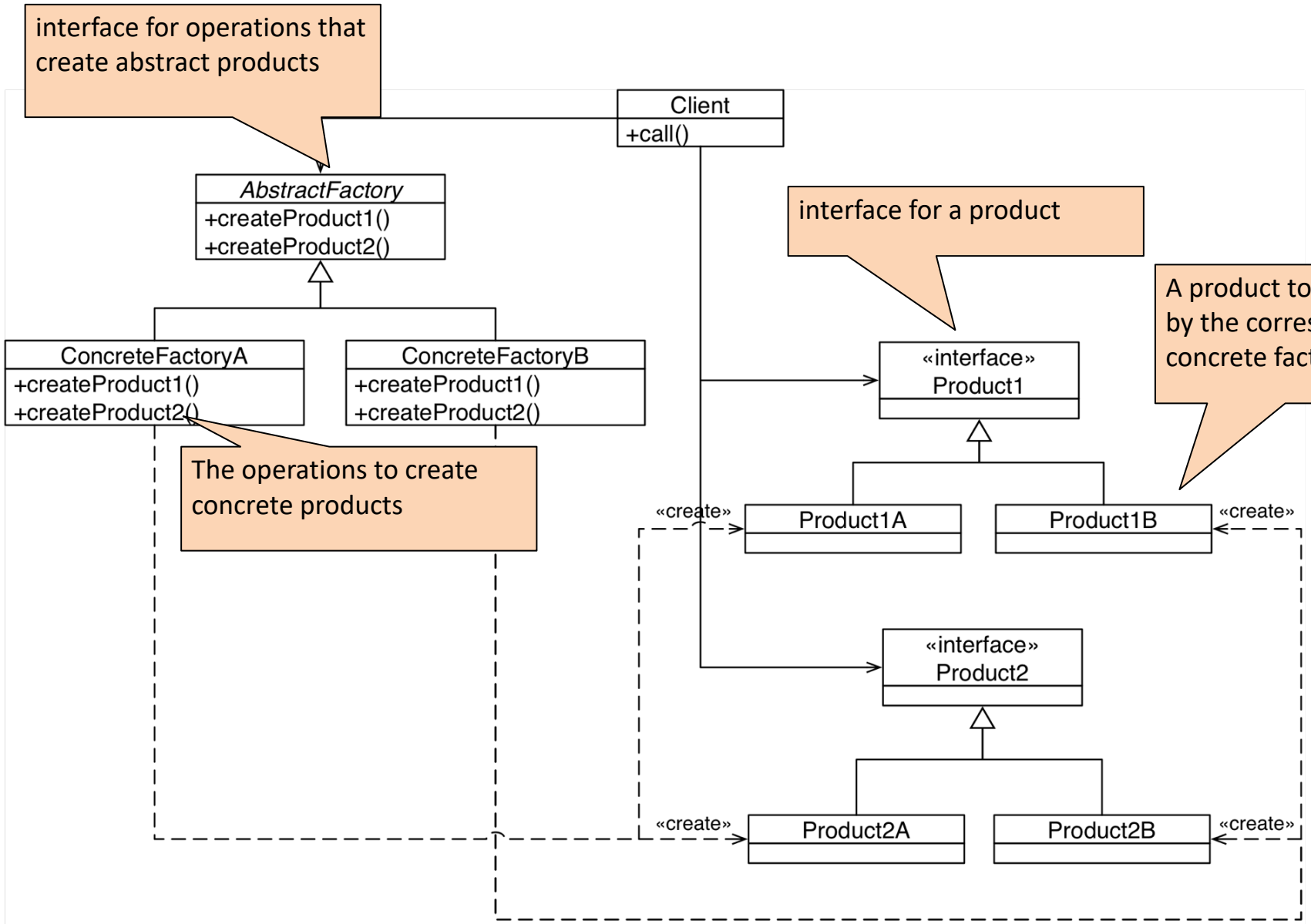
From the factory class to abstract factory pattern

- The factory class calls new inside the factory method, reducing the dependency of the client code on a concrete implementation
- But we are bound to a single Factory
- How can we generalize this, if:
  - Cheese: New Yorkers like Mozarella / Chicagoers prefer Reggiano cheese
  - Sauce: New Yorkers like Plumsauce / Chicagoers prefer Marina
- We want an application for pizza toppings which can be customized for people living in different cities
  - Can we create factories for Chicago style pizzas and NY style pizzas?

A pizza as a family of related objects of toppings and sauces



Abstract factory pattern

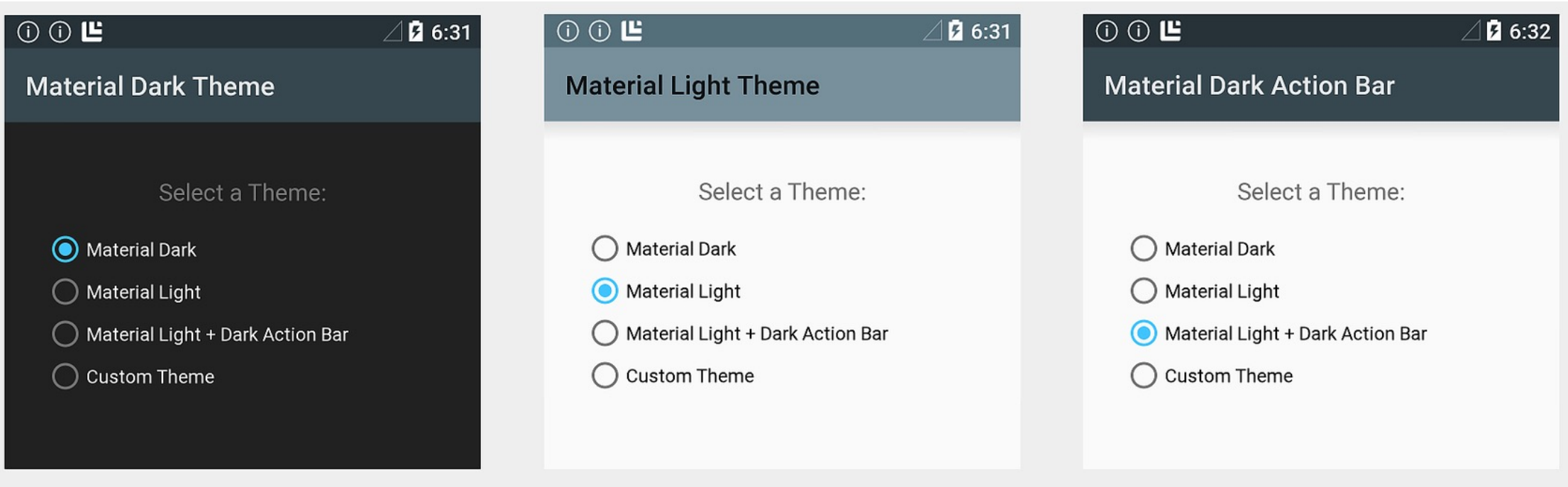


Note for the Pizza example:

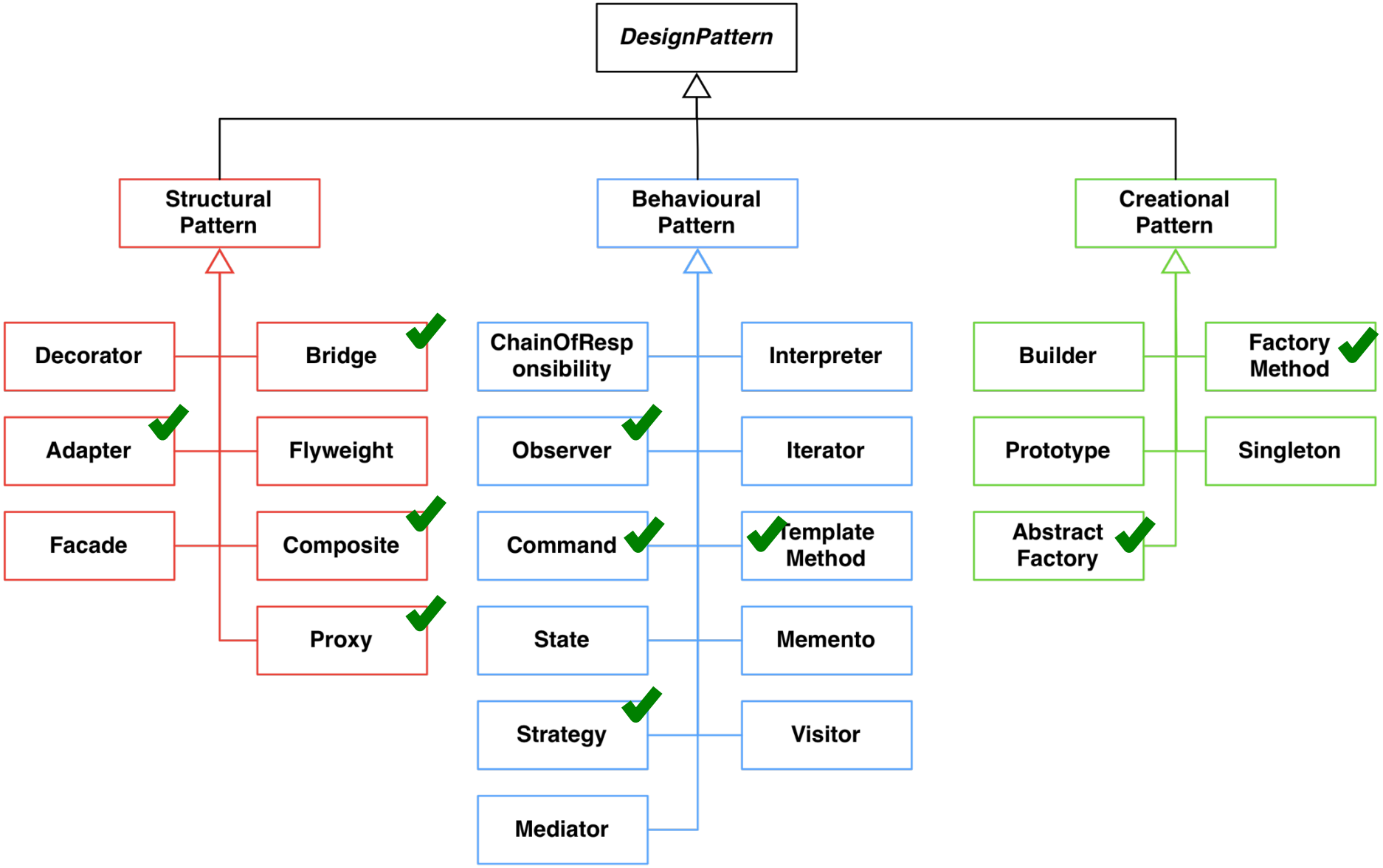
- Product1 is Cheese
- Product2 is Sauce
- Before, the Pizza was the Product

Abstract factory pattern motivation

- User interface that supports different themes
  - How to write a single user interface that supports different themes?
  - Example: Light theme, Dark theme



Summary


DesignPattern
Structural Pattern
Behavioural Pattern
Creational Pattern
Decorator
Bridge
Adapter
Flyweight
Facade
Composite
Proxy
ChainOfResponsibility
Interpreter
Observer
Iterator
Command
Template Method
State
Memento
Strategy
Visitor
Mediator
Builder
Factory Method
Prototype
Singleton
Abstract Factory