Design Pattern

Lecture 14

Background

Designing Reusable software is difficult

- finding good objects and abstraction
- flexibility, modularity, elegance ➡ reuse
- takes time to emerge, trial and error

Successful design do exist

- exhibit recurring class and object structure

How to describe these recurring structure?

A Design Pattern

Describe a recurring design structure

- Abstracts from concrete design
- Identify classes, collaborations, responsibilities
- Applicability, trade-offs, consequences

Examples

- Strategy; algorithm as objects
- Composite: recursive structure

Pattern Description

Context: The general situation in which the pattern applies

Problem: A short sentence or two raising the main difficulty

Forces: the issues or concerns to consider when solving the problem

Solution: the recommended way to solve the problem in the context

Antipattern (opt) : Solution that are inferior or do not work in the this context

Related pattern (opt): Pattern that are similar to this pattern

References: who developed or inspired this pattern

Pattern are not design

Must be instantiated

- Evaluate trade-offs and consequences
- Make design and implementation decision
- Implement, combine with other pattern

Pattern are not framework

Framework codify design for solving a family of problem in a specific domain

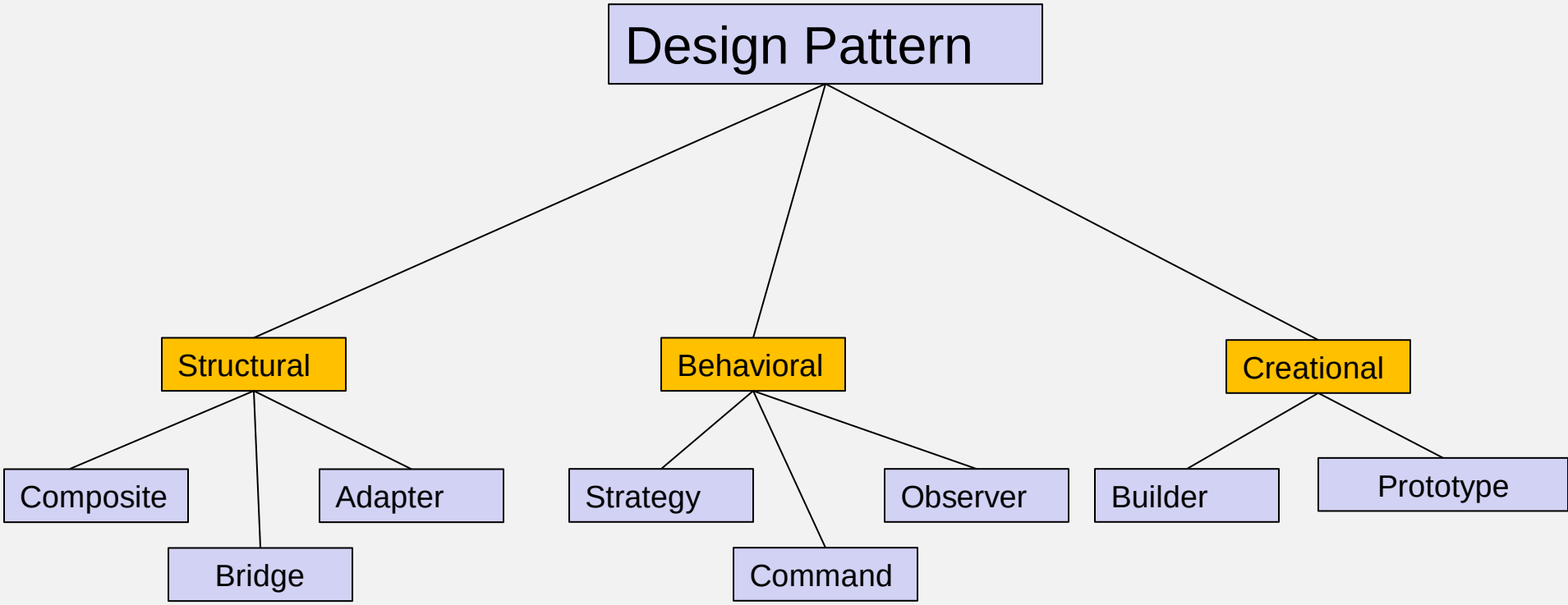
- Abstract, cooperating classes

Customized by

- User defined subclasses
- Composition of objects

Contain instances of multiple pattern

A Taxonomy of Design Patterns



relationship

Obj interaction

Obj creation

Robustness to Change

Change is intrinsic to software

- Requirements, technology, platforms
- Alternative usage scenarios

Robustness to change determines

- Ease of evolution
- Subsequent maintenance costs
- Ultimate reusability

Robustness to Change

Each pattern addresses a particular variation

- algorithms, implementation, instantiated classes
- design based on patterns robust to these variations

Consider variations during life-cycle

- Up-front analysis of variations
- Understand nature of change and “hot-spots”
- Identify and apply pattern

Design Confidence

General inexperience with objects

- Is my design ok?

Pattern engender confidence

- Still leaves room for creativity

Most People know the pattern

- But partially not with full understanding
- Liberating to know that others have similar design
- Pattern improves with use

Common problem

Taming over-enthusiasm

- You ‘win” if you have the most patterns
- Solve the wrong problem
- Associated expense and cost
- Everything solved by the last pattern you learned

Structure instead of Intent

- Everything is a strategy
- Pattern use similar construct

Finding the right Pattern

Not always clear when pattern is applicable

- Hard to ask right questions during design
- Require expertise in both domain and pattern

Once pattern is found, design falls in place

- Evaluate tradeoffs and implement variation

Behavioral Pattern: Observer, motivation

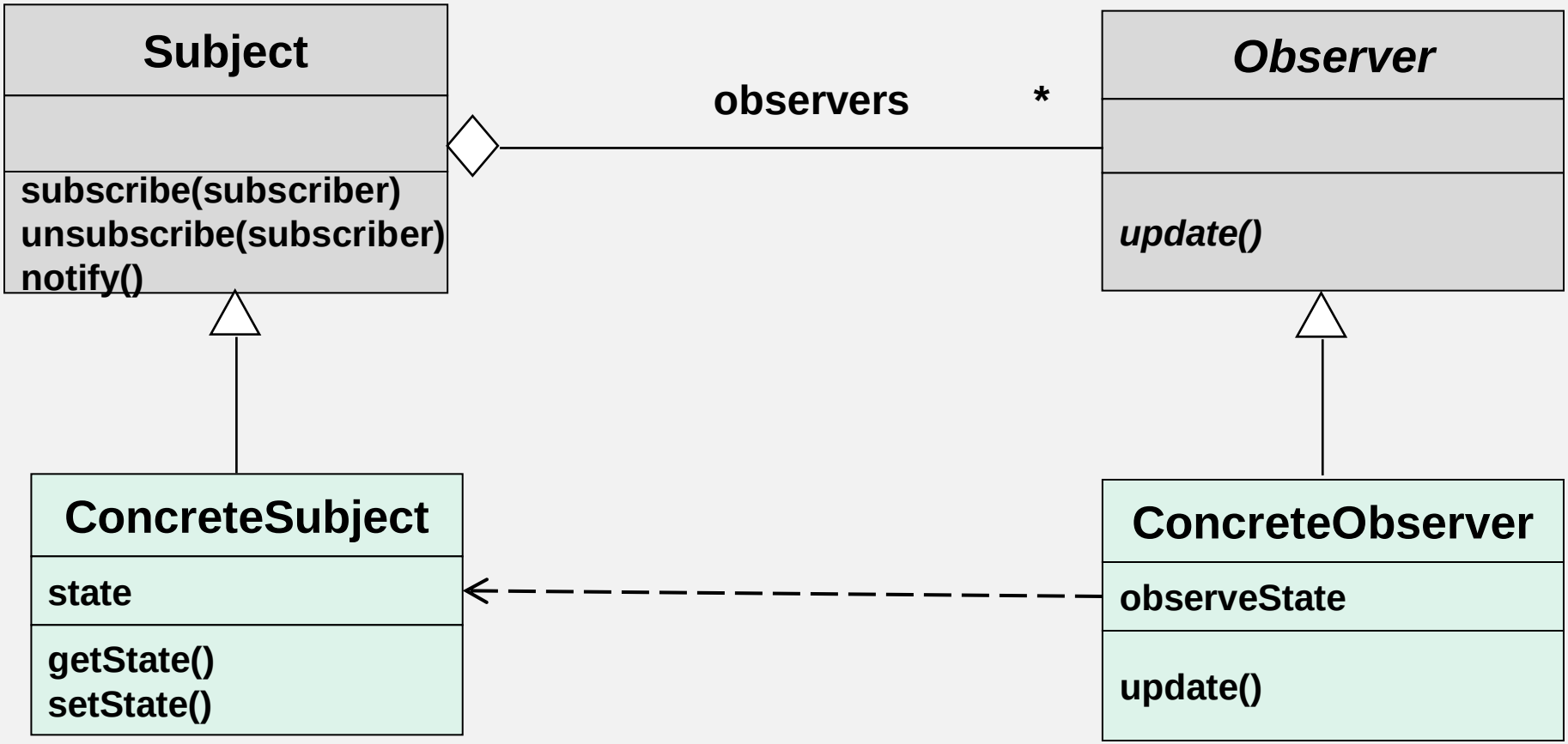
Intent:

- Define One to many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically

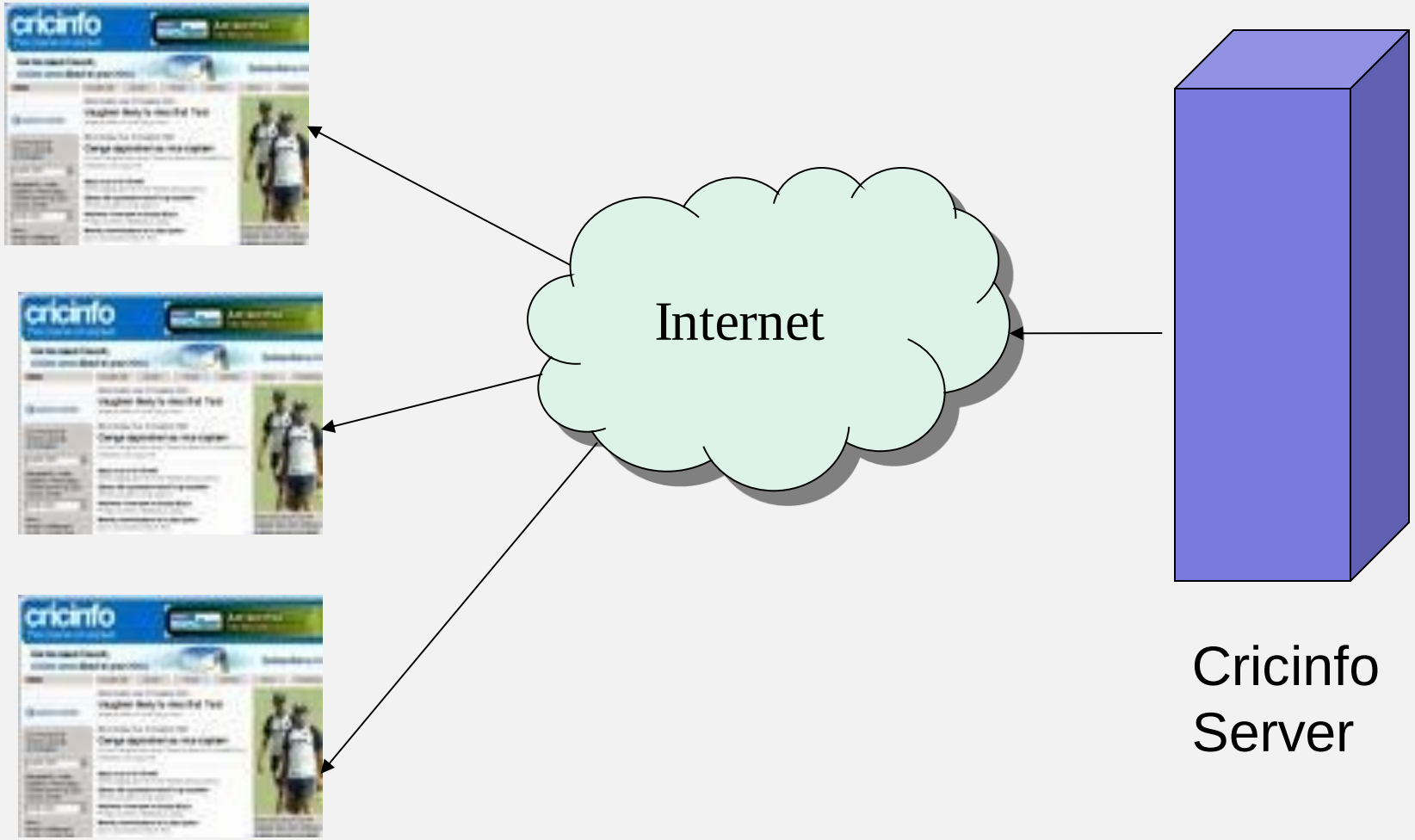
Key Force

- There may be many observer
- Each observer may react differently to the same notification
- The subject should be as decoupled as possible from the observers to allow observers to change independently of the subject.

Observer Pattern: Decouples an Abstraction from its Views



Observer :Example



Observer Pattern Consequences

- Decouple Subject which maintain state, from observers, who make use of state.

- Can result in many spurious broadcasts when the state of Subject changes.

Observer Pattern Motivation

Requirements:

- The system should maintain consistency across the (redundant) views, whenever the state of the observed object changes

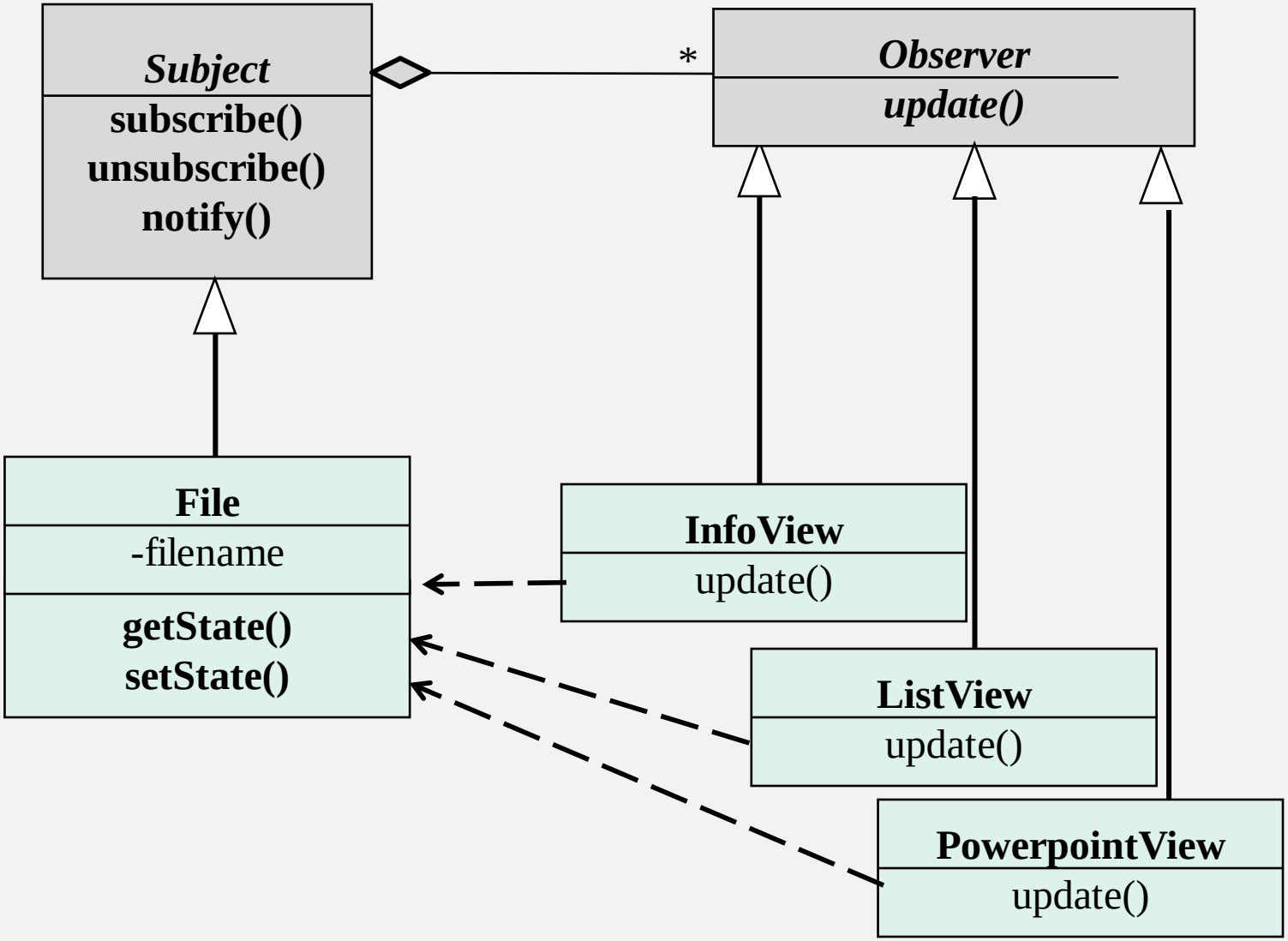
- The system design should be highly extensible
  - It should be possible to add new views without having to recompile the observed object or the existing views.

Observer Pattern: Decouples an Abstraction from its Views

- The Subject (“Publisher”) represents the entity object
- Observers (“Subscribers”) 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.

Applying the Observer Pattern to maintain Consistency across Views



Observer Pattern

- Models a 1-to-many dependency between objects

  Connects the state of an observed object, the subject with many observing objects, the observers

Usage:

- Maintaining consistency across redundant states
- Optimizing a batch of changes to maintain consistency

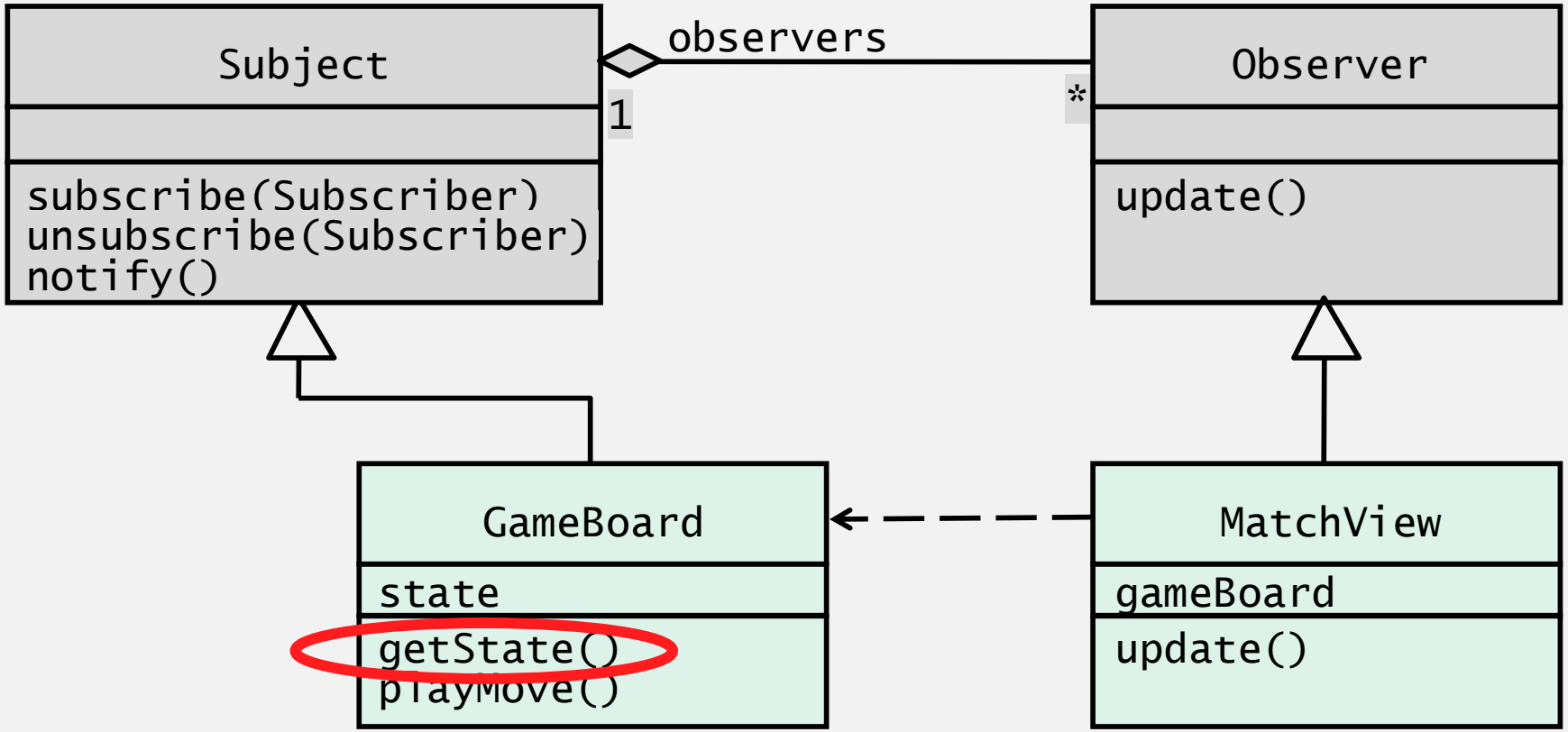
Three variants for maintaining the consistency:

Push Notification: Every time the state of the subject changes, *all* the observers are notified of the change

Push-Update Notification: The subject also sends the state that has been changed to the observers

Pull Notification: An observer inquires about the state the of the subject

Consistency across MatchViews



Push, Pull or Push-Update Notification?

Observer Pattern

A simple weather monitoring system where we have a ‘WeatherStation’ as the subject, and various display devices (observers) like ‘CurrentConditionsDisplay’ and ‘StatisticsDisplay’ that want to be notified whenever the weather changes

```
class WeatherStation {
    private List<Observer> observers = new ArrayList<>();
    private float temperature;

    public void addObserver(Observer observer) {
        observers.add(observer);
    }

    public void removeObserver(Observer observer) {
        observers.remove(observer);
    }

    public void setTemperature(float temperature) {
        this.temperature = temperature;
        notifyObservers();
    }

    private void notifyObservers() {
        for (Observer observer : observers) {
            observer.update(temperature);
        }
    }
}
```

```
// Observer
interface Observer {
    void update(float temperature);
}
```

```
class CurrentConditionsDisplay implements Observer {
    private float temperature;

      public void update(float temperature) {
        this.temperature = temperature;
        display();
    }

    private void display() {
        System.out.println("Current Conditions Display:
Temperature = " + temperature);
    }
}
```

Observer Pattern

a simple weather monitoring system where we have a ‘WeatherStation’ as the subject, and various display devices (observers) like ‘CurrentConditionsDisplay’ and ‘StatisticsDisplay’ that want to be notified whenever the weather changes

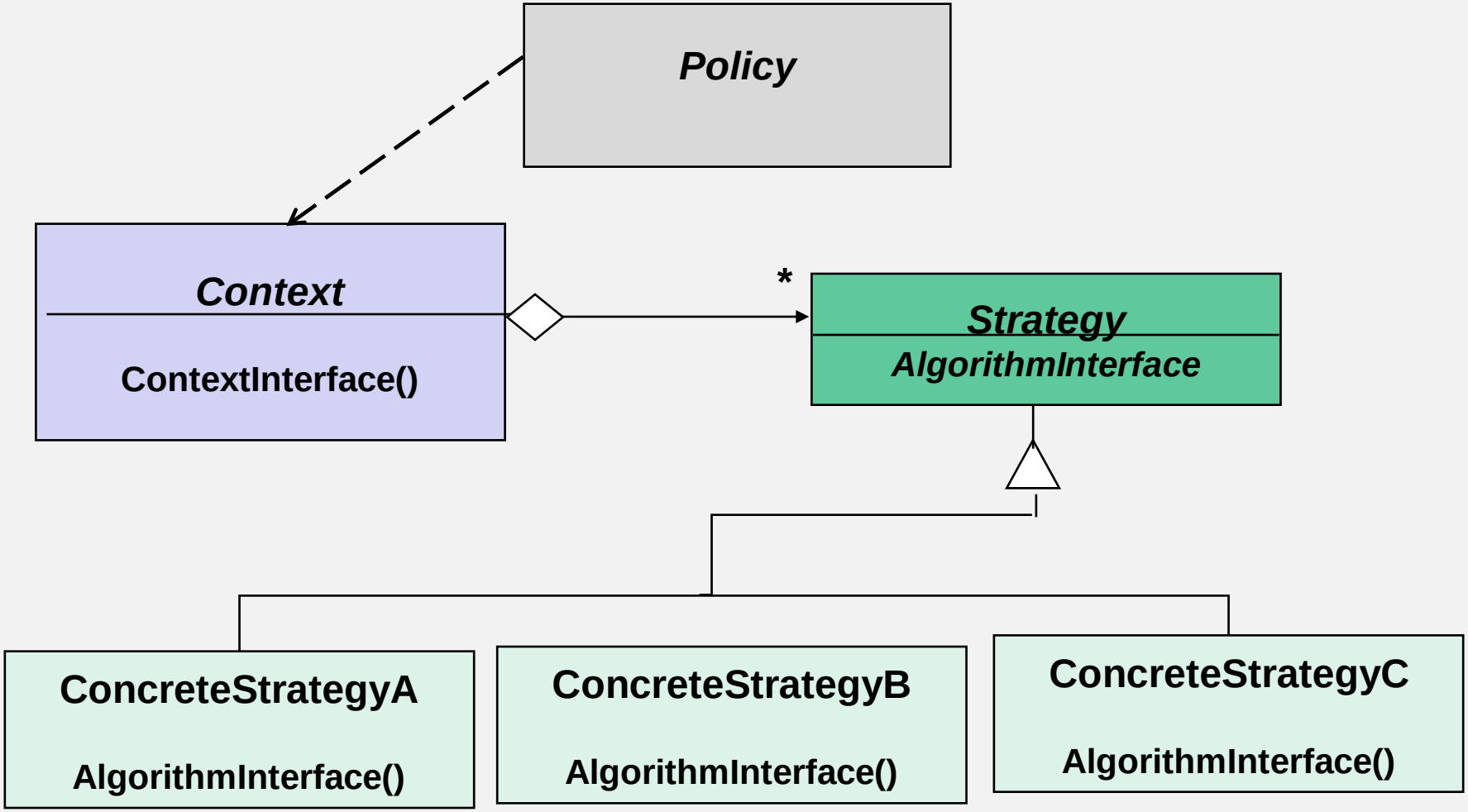
```
// Concrete Observer
class StatisticsDisplay implements Observer {
    private float temperature;

        public void update(float temperature) {
        this.temperature = temperature;
        display();
    }

    private void display() {
        System.out.println("Statistics Display: Temperature
= " + temperature);
    }
}
```

```
public class ObserverPatternExample {
    public static void main(String[] args) {
        WeatherStation weatherStation = new WeatherStation();

        CurrentConditionsDisplay currentConditionsDisplay = new
CurrentConditionsDisplay();
        StatisticsDisplay statisticsDisplay = new StatisticsDisplay();

        weatherStation.addObserver(currentConditionsDisplay);
        weatherStation.addObserver(statisticsDisplay);

        // Simulate a change in temperature
        weatherStation.setTemperature(25.5f);

        // Simulate another change in temperature
        weatherStation.setTemperature(30.0f);

        // Remove an observer
        weatherStation.removeObserver(currentConditionsDisplay);

            }
}
```

Behavioral Pattern: Strategy

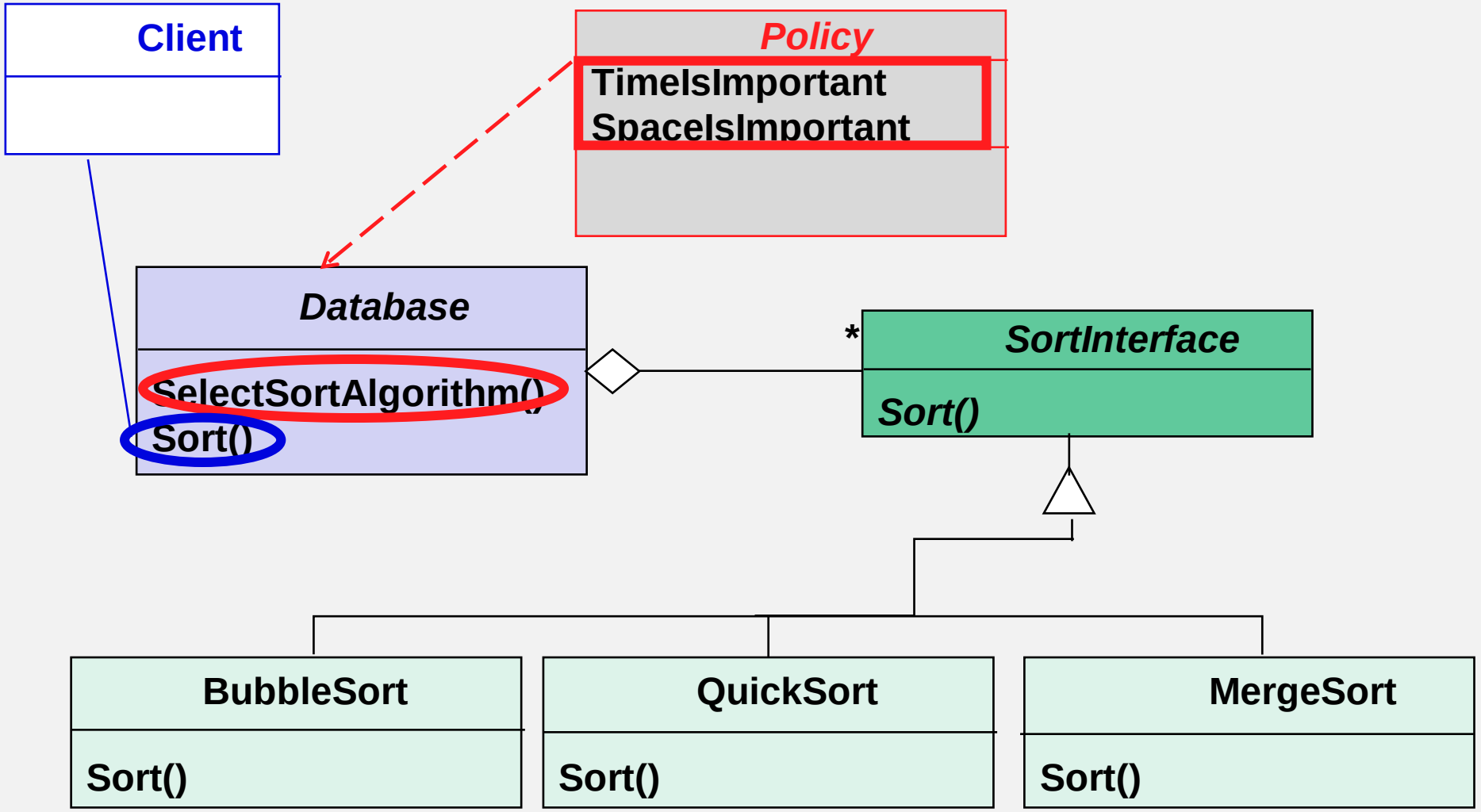
- Different algorithms exists for a specific task
  - We can switch between the algorithms at run time
- Examples of tasks:
  - Sorting a list of customers (Bubble sort, mergesort, quicksort)
  - Different collision strategies for objects in video games
  - Parsing a set of tokens into an abstract syntax tree (Bottom up, top down)
- Different algorithms will be appropriate at different times
  - First build, testing the system, delivering the final product
- If we need a new algorithm, we can add it without disturbing the application or the other algorithms.

Strategy Pattern

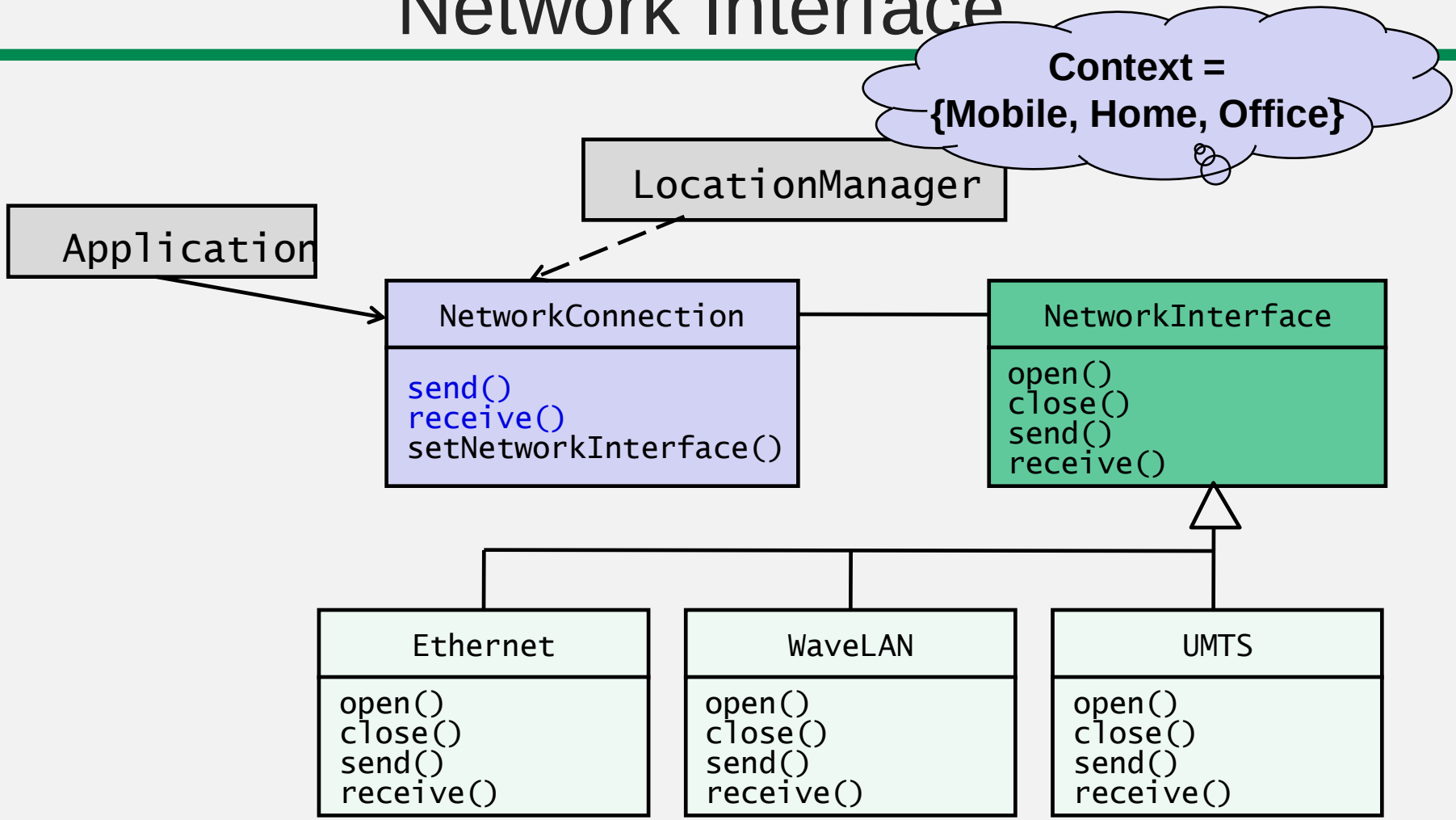


Policy decides which ConcreteStrategy is best in the current Context.

Using a Strategy Pattern to Decide between Algorithms at Runtime



Supporting Multiple implementations of a Network Interface



Creational Pattern: Builder

The construction of a complex object is common across several representations

Example

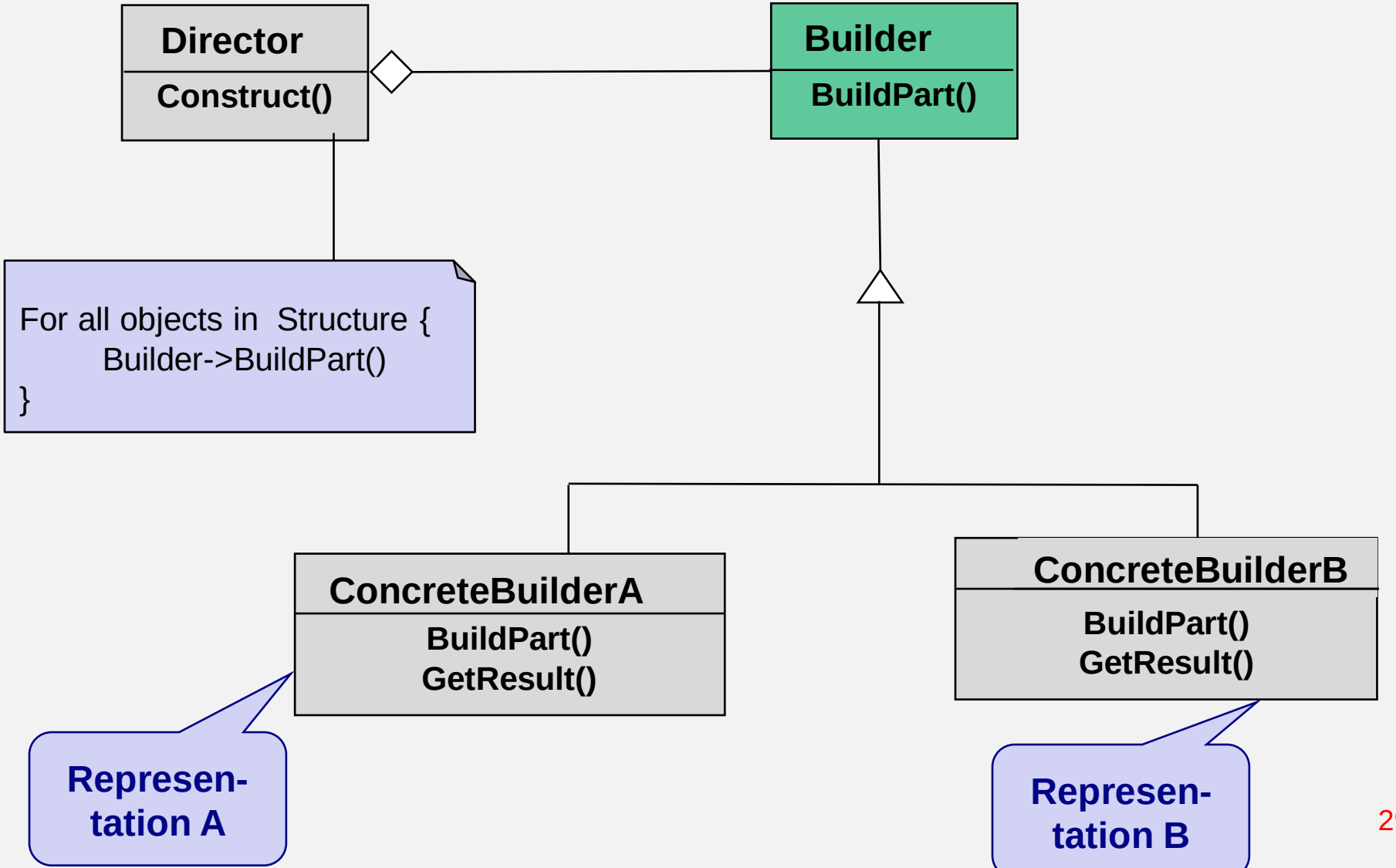
Converting a document to a number of different formats

- the steps for writing out a document are the same
- the specifics of each step depend on the format

Approach

- The construction algorithm is specified by a single class (the “director”)
- The abstract steps of the algorithm (one for each part) are specified by an interface (the “builder”)
- Each representation provides a concrete implementation of the interface (the “concrete builders”)

Builder Pattern

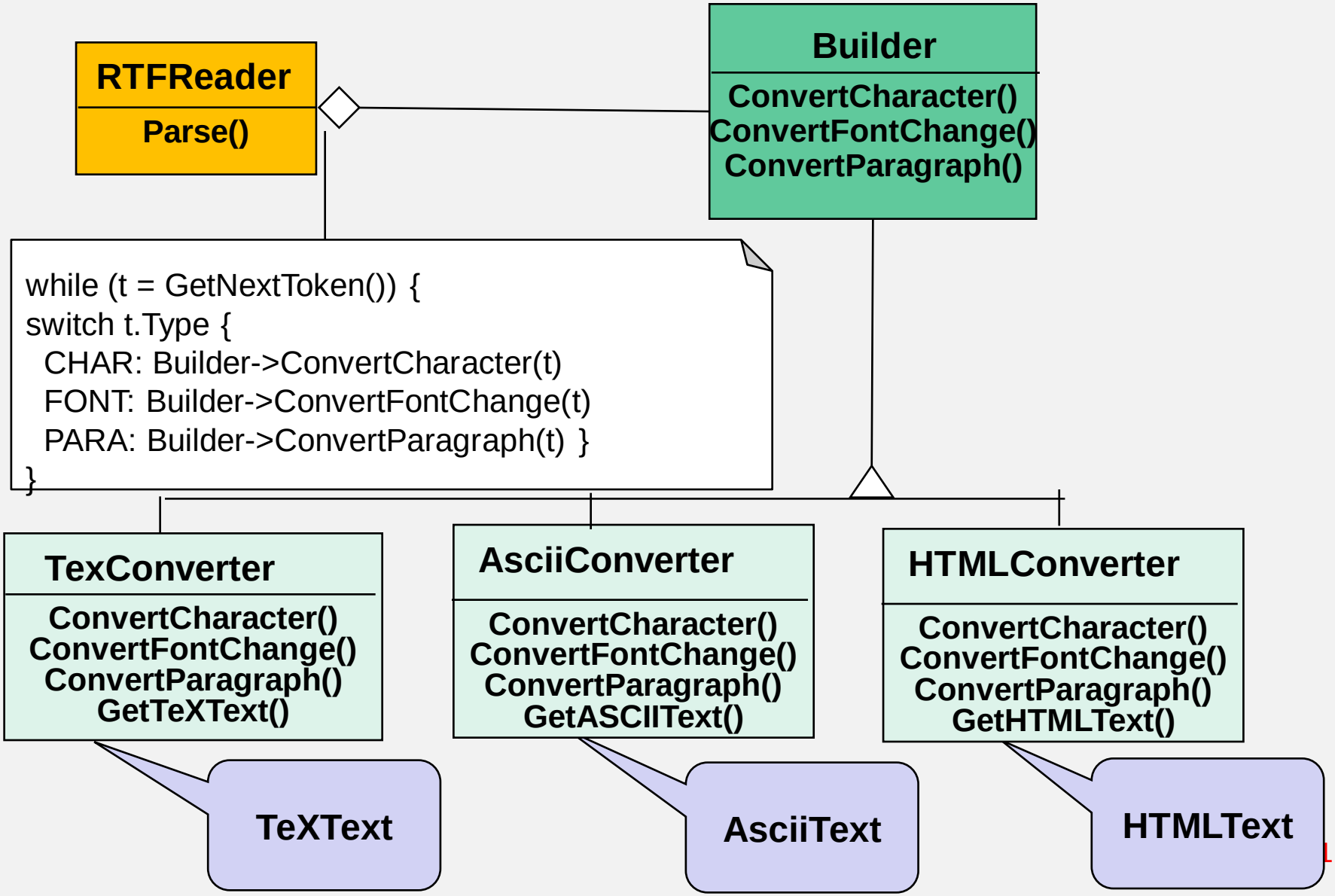


Applicability of Builder Pattern

- The creation of a complex product must be independent of the particular parts that make up the product

- The creation process must allow different representations for the object that is constructed.

Example: Converting an RTF Document into different representations



Creational Pattern: Prototype

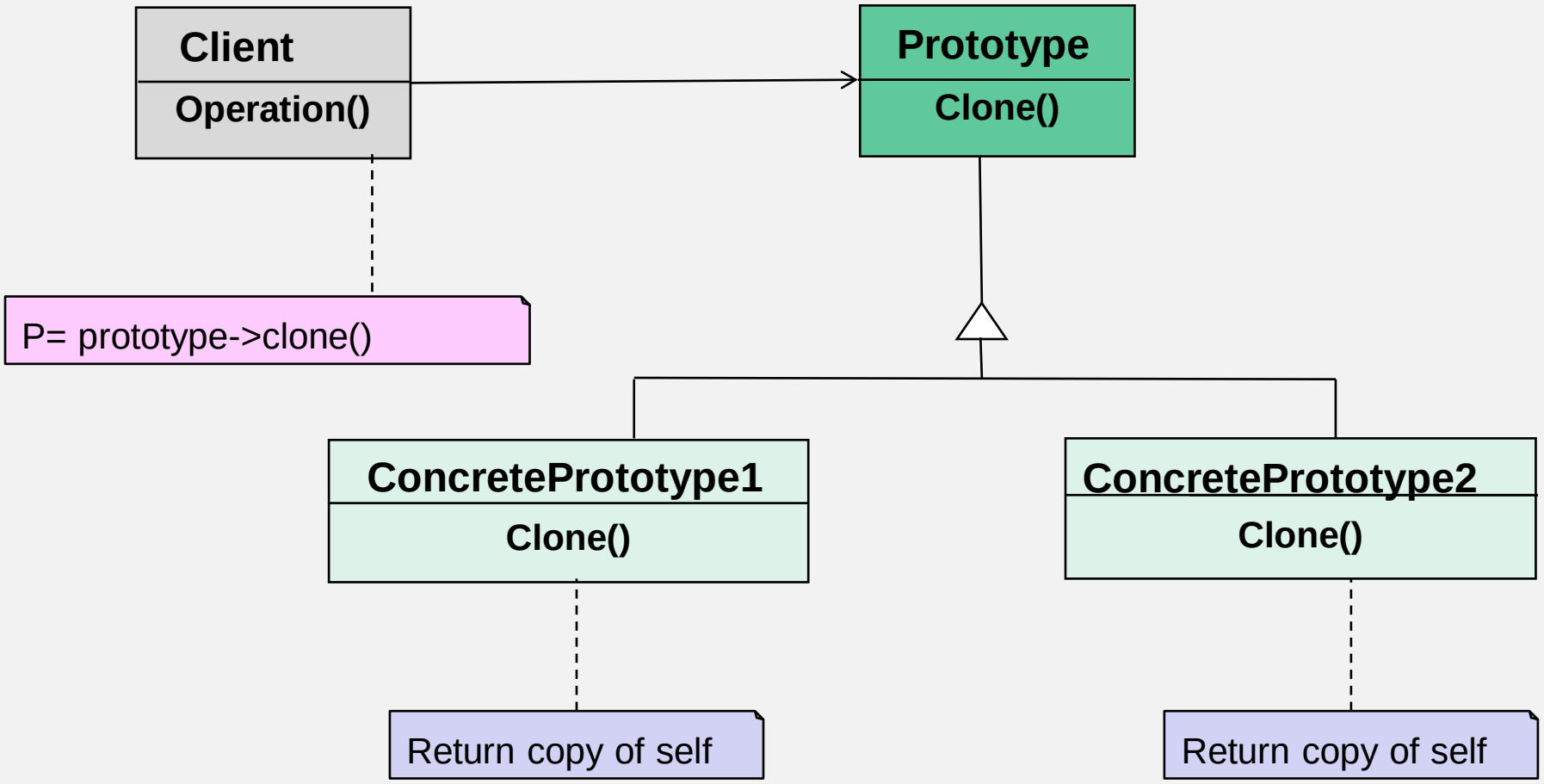
Intent

- Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

Applicability

- when instances of a class can have one of only a few different combinations of state. It may be more convenient (e.g. to avoid complex conditional logic) to install a corresponding number of prototypes and clone them.

Prototype Pattern



Structural Pattern: Adapter

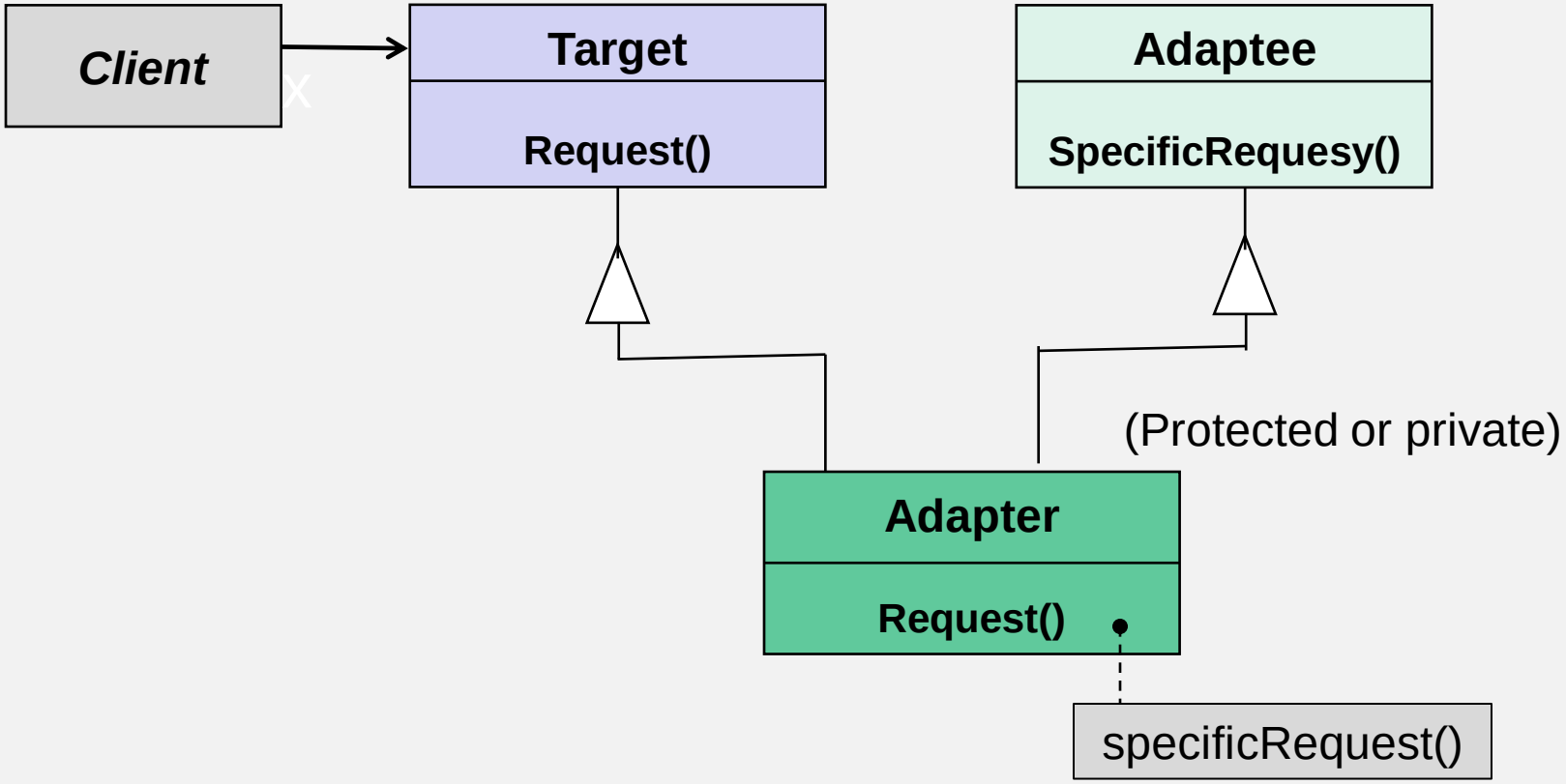
Intent:

- Change the interface of a class into another interface which is expected by the client.

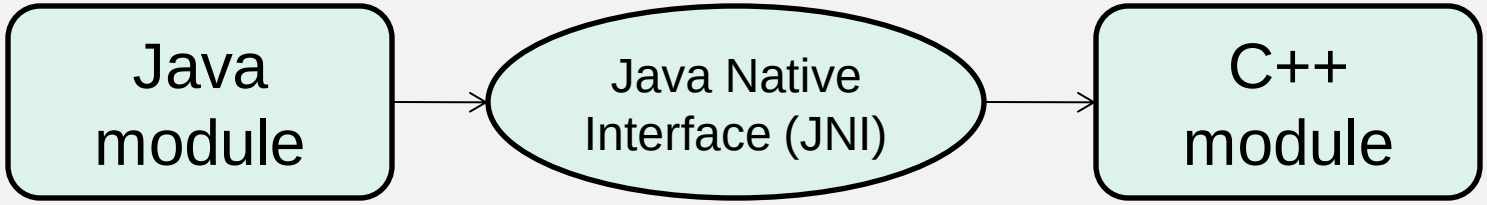
- Also Know As: Wrapper

- Use an existing class whose interface does not match the requirement
- Create a reusable class though the interfaces are not necessary compatible with callers

Adapter Pattern



Adapter Pattern

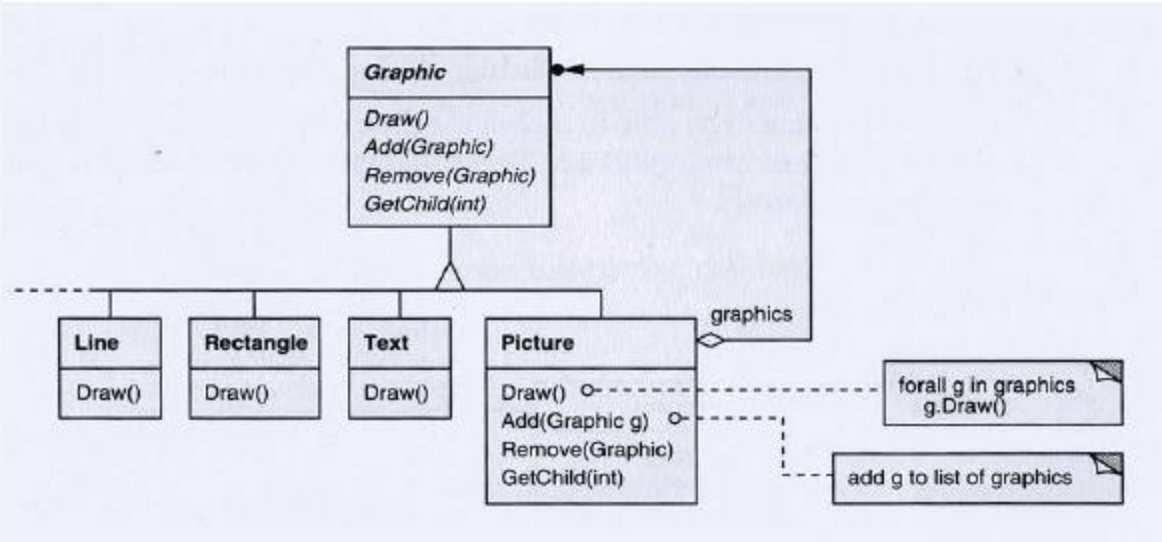


Structural Pattern: Composite

Intent:

- Compose objects into tree structures to represent part whole hierarchies.

- Composite lets clients treat individual objects and compositions of objects uniformly.



Composite Pattern

- Creates a class that contains group of its own objects.
- The Component(Graphic) is an abstract class that declares the interface for the objects in the pattern. As the interface, it declares methods(such as Draw) that are specific to the graphical objects.
- •Line, Rectangle, and Text are so called Leafs, which are subclasses that implement Draw to draw lines, rectangles, and text, respectively.

Thank You