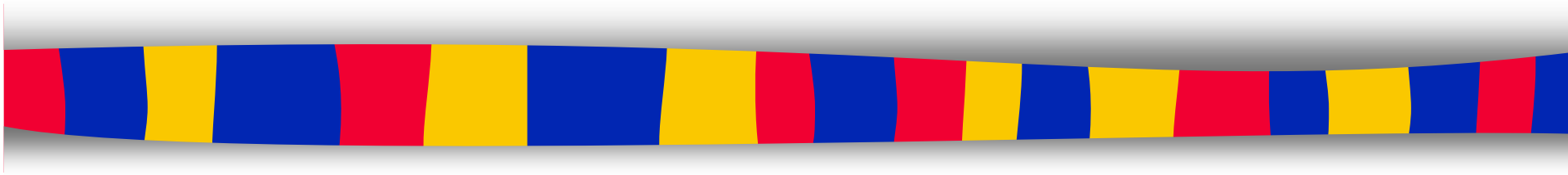
EECS 4314
Advanced Software Engineering



Topic 05:
Design Pattern Review

Zhen Ming (Jack) Jiang

Acknowledgement

- Some slides are adapted from Ahmed E. Hassan, Jonathan Ostroff, Spiros Mancoridis and Emad Shihab

The Evolution of Programming Abstractions

- The first modern programmable computers (1950s) were largely hardwired. The first software was written in machine language
- Next major breakthrough: assembly languages
  - Symbolic assemblers
  - Macro processors
- 1960s: High-level languages (3GLs)
  - Mostly independent of machine and problem domain
    - Level is “generic problem-solving”
  - FORTRAN, COBOL
  - Algol, Pascal, Modula
  - C, PL/1
  - Simula, Smalltalk; C++, Java

Abstraction from Developers' Perspective

- Typed variables and user-defined types [late 1960s]
- Modules [early 1970s]
  - 1968: "The software crisis", need "software engineering"
  - Create explicit interfaces, enforce information hiding
- ADTs and object-oriented computing [mid 1970s]
  - Programming entities as mathematically precise constructs
  - Abstract commonalities to one place
- Object-oriented design patterns, refactoring [1990s]
  - OO is powerful and complex
    - What constitutes a "good" OO design (small to medium-sized programs)?
    - What re-usable "tricks" can help to solve recurring problems?
  - At the level of data structures, algorithms and a few co-operating classes
- Software architecture [1990s, but really since 1960s]
  - Designing large systems is about understanding broad tasks, defining system-wide structures, interfaces and protocols, understanding how non-functional requirements impact on the system
  - At the level of the handful of "big boxes" that comprise the major components of your system, plus their interdependencies

What are Object-Oriented Design Patterns (OODP)?

- Design patterns are reusable solutions to common problems.
  - An OODP typically involves a small set of classes co-operating to achieve a desired end
  - This is done via adding a level of indirection in some clever way, and
  - The new improved solution provides the small functionality as an existing approach, but in the some more desirable way (elegance, efficiency and adaptability)
- OODPs often make heavy use of interfaces, information hiding, polymorphisms and intermediary objects
- Typical presentation of an OODP
  - A motivating problem and its context,
  - Discussion of the possible solutions, and
  - Common variations and tradeoffs

Learning Design Patterns

- Think of OODP as high-level programming abstractions
  - First, you learn the basics (data structures, algorithms, tools and language details)
  - Then, you learn modules, interfaces, information hiding, classes/OO programming
  - Design patterns are the next level of abstraction
  - (… Software Architecture)

Design Patterns help you …

- Design new systems using higher-level abstractions than variables, procedures and classes
- Understand relative tradeoffs, appropriateness, (dis)advantages of patterns
- Understand the nature both of the system you are constructing and of OO design in general
- Communicate about systems with other developers
- Give guidance in resolving non-functional requirements and trade-offs
  - Portability, extensibility, maintainability, re-usability, scalability, …
- Avoid known traps, pitfalls and temptations
- Ease restructuring, refactoring, and
- Foster coherent, directed system evolution and maintenance based on a greater understanding of OO design

Design Patterns – Another form of Reuse

- Someone has already solved your problem
- Exploit the wisdom and lessons learned by other developers who have been down the same design problem road and survived the trip.
- Instead of code reuse, with patterns you get experience reuse.

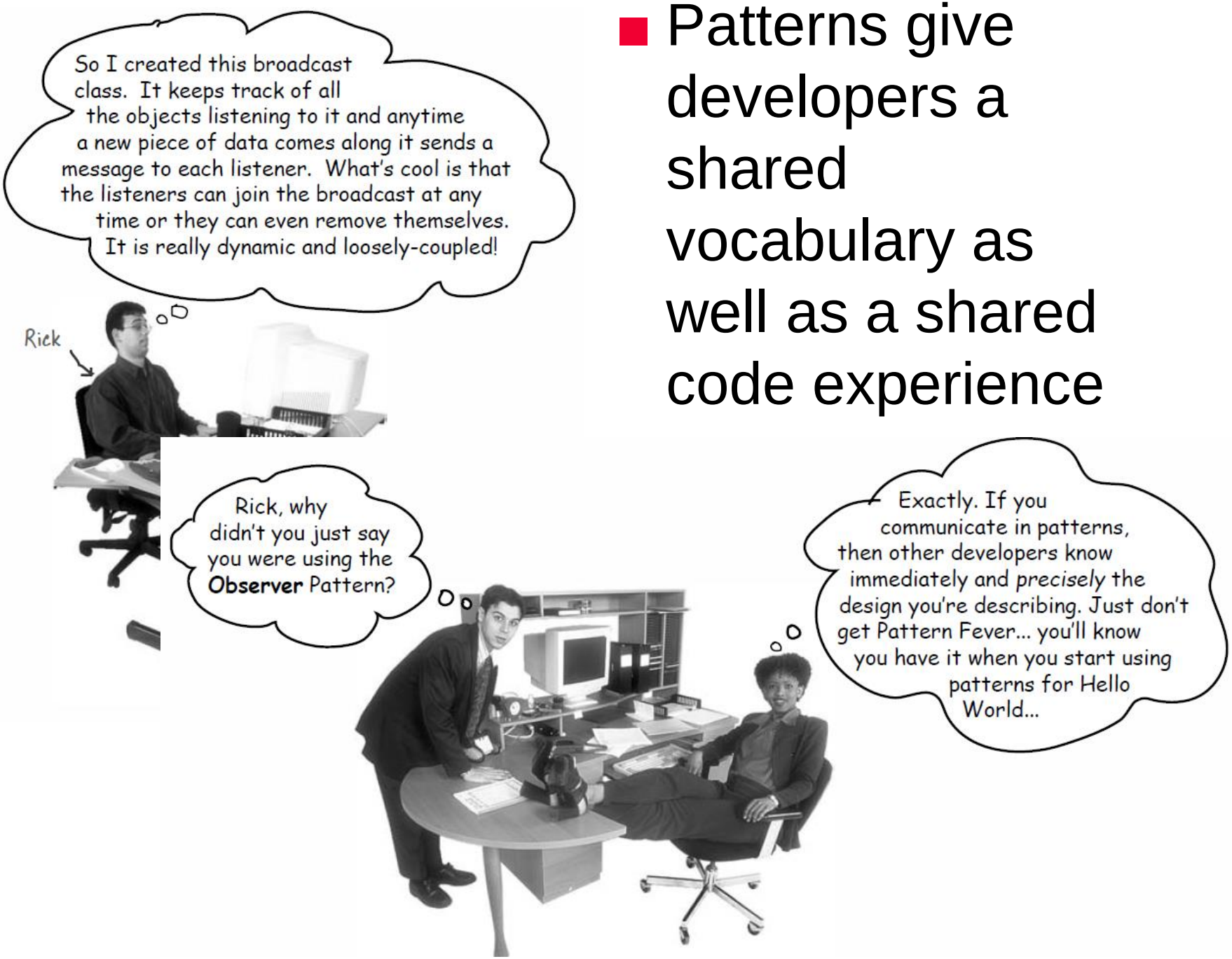
Design Pattern Categories

- Gang of Four (GoF) Design patterns (23 patterns)
  - **Creational patterns**: concern the process of object *creation*
    - Abstract factory, Singleton, Factory method, etc.
  - **Structural patterns**: concern the process of *assembling* objects and classes
    - Adapter, Façade, Composite, Decorator, etc.
  - **Behavioral patterns**: concern the *interaction* between classes or objects
    - Iterator, Observer, Strategy, etc.

There are thousands of patterns "out there" and thousands more waiting to be discovered

- *Some are "domain specific"*
- *Some are a lot like others, special cases, etc.*
- *There is no official person/group who decides what is/isn't a design patterns*

- Patterns give developers a shared vocabulary as well as a shared code experience



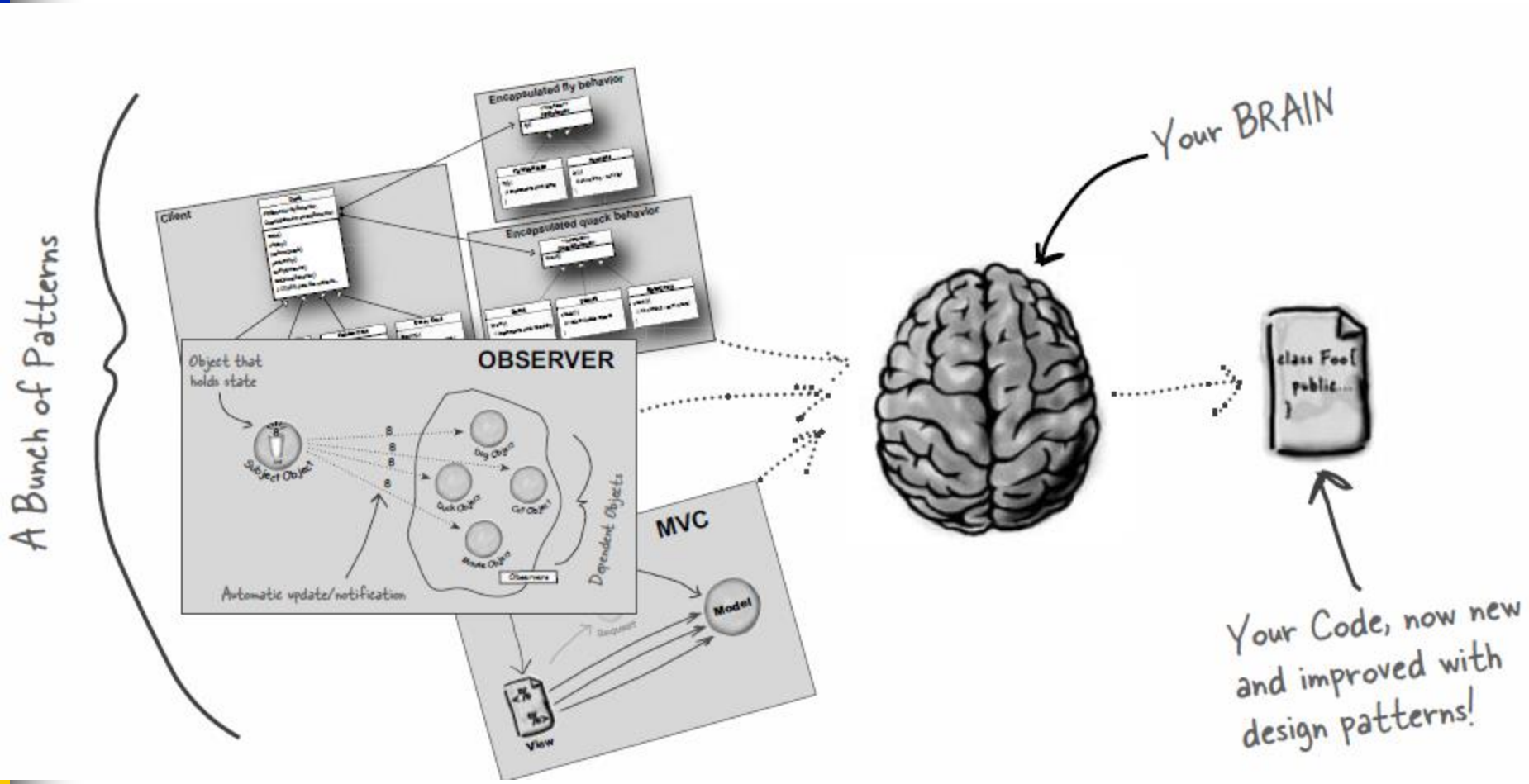
Shared pattern vocabularies are POWERFUL. When you communicate with another developer or your team using patterns, you are communicating not just a pattern name but a whole set of qualities, characteristics and constraints that the pattern represents.

"We're using the strategy pattern to implement the various behaviors of our ducks." This tells you the duck behavior has been encapsulated into its own set of classes that can be easily expanded and changed, even at runtime if needed.

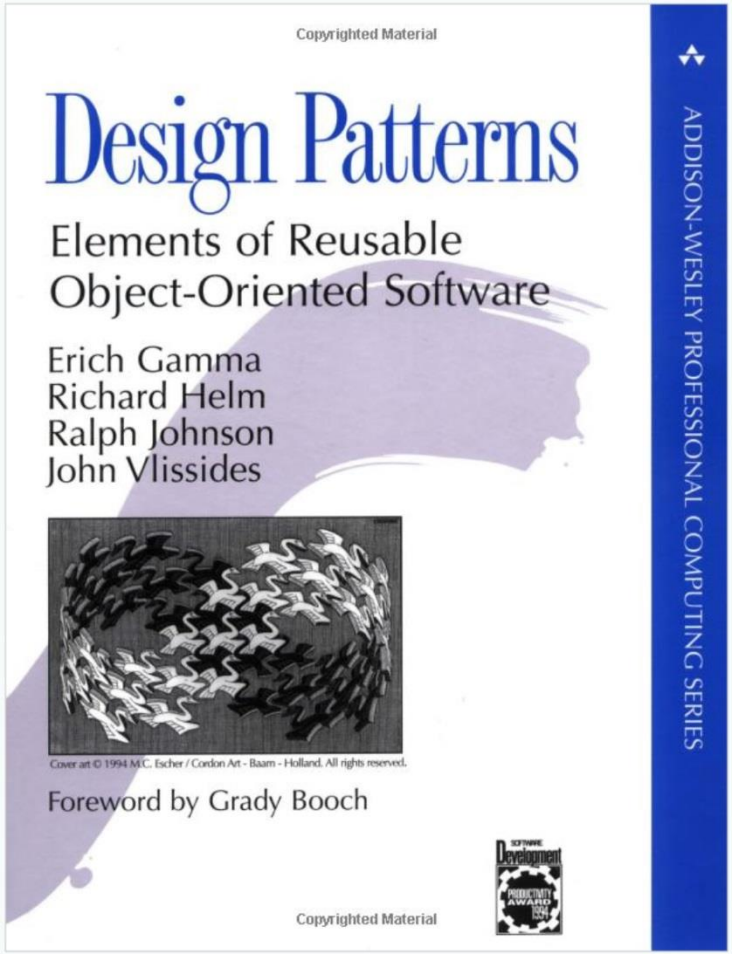
Patterns allow you to say more with less. When you use a pattern in a description, other developers quickly know precisely the design you have in mind.

Talking at the pattern level allows you to stay "in the design" longer. Talking about software systems using patterns allows you to keep the discussion at the design level, without having to dive down to the nitty gritty details of implementing objects and classes.

How many design meetings have you been in that quickly degrade into implementation details?


A Bunch of Patterns
Encapsulated fly behavior
Client
Encapsulated quack behavior
OBSERVER
Object that holds state
Subject Object
Duck Object
Quack Object
Goose Object
Dependent Objects
Automatic update/notification
MVC
Model
View
Your BRAIN
class Foo{
public...
}
Your Code, now new and improved with design patterns!

Design Pattern References



http://www.hillside.net/patterns/

Design Patterns Covered

- **Structural**
  - Adapter
  - Façade
  - Composite
- **Behavioral**
  - Iterator
  - Strategy
  - State
  - Template
  - Observer
  - Master-Slave
- **Creational**
  - Abstract Factory
  - Singleton

For Each Pattern ….

- **Motivation** – the problem we want to solve using the design pattern
- **Intent** – the intended solution the design pattern proposes
- **Structure** – how the design pattern is implemented
- **Participants** – the components of the design pattern

Terminology

- **Objects** package both data and the procedures that operate on that data
- An object performs an operation when it receives a **request** (or **message**) from a **client**
- Procedures are typically called **methods** or **operations**
- An object's implementation is defined by its **class**. The class specifies
  - Object's **internal data** and representation
  - **Operations** that object can perform
- The set of signatures defined by an object's operations or methods is called the **interface**
- An **abstract class** is one whose main purpose is to define a common interface for its subclass

Structural Design Patterns
- Adapter, Façade, Composite

The Adapter Design Pattern

The Adapter Design Pattern

- **Motivation:**
  - When we want to reuse classes in an application that expects classes with a different interface, we do not want (and often cannot) to change the reusable classes to suit our application.
- **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.

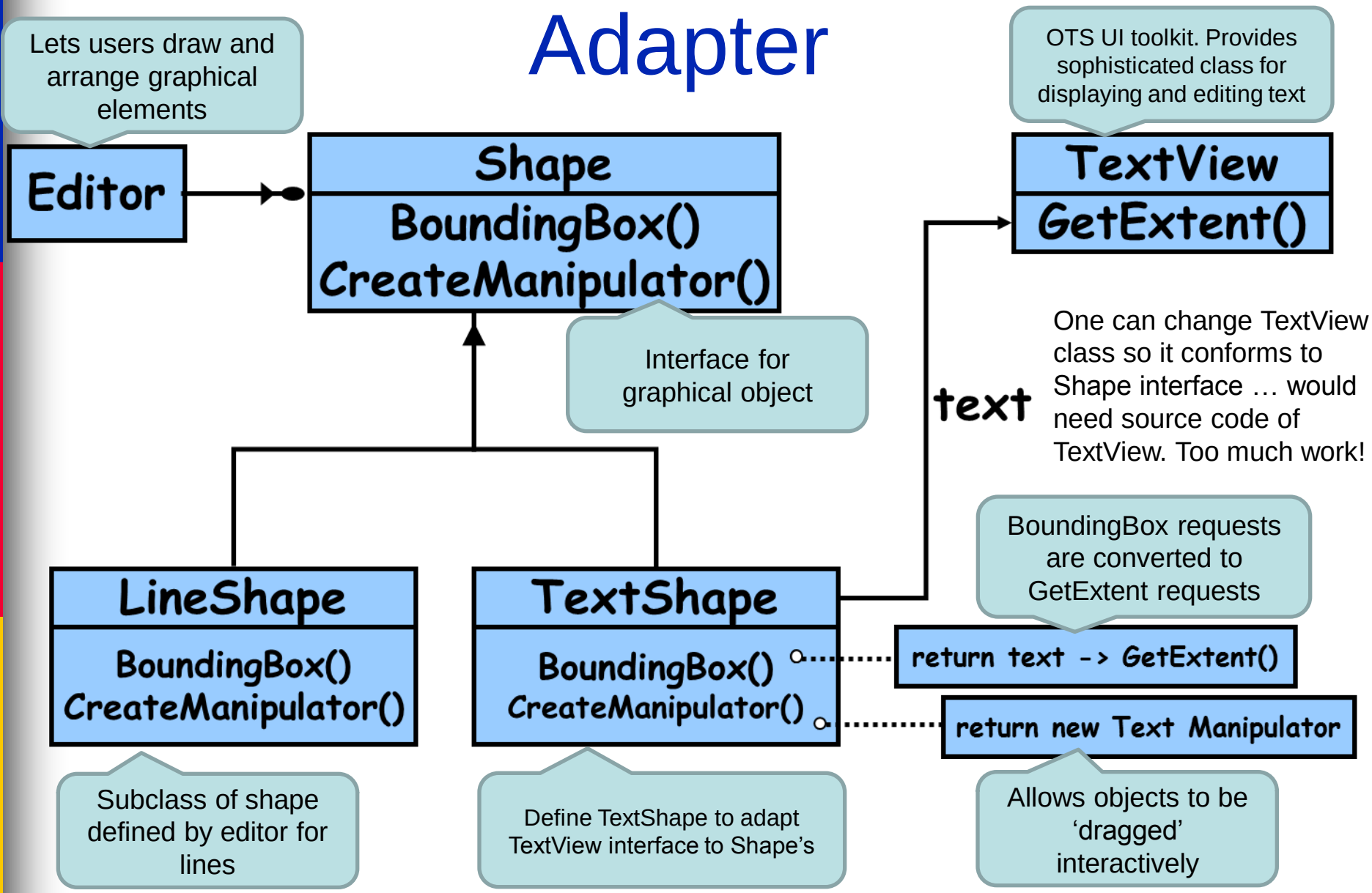
Applicability

- Use an existing class when its interface does not match the one you need
- Create a class that cooperates with unrelated or unforeseen classes with incompatible interfaces

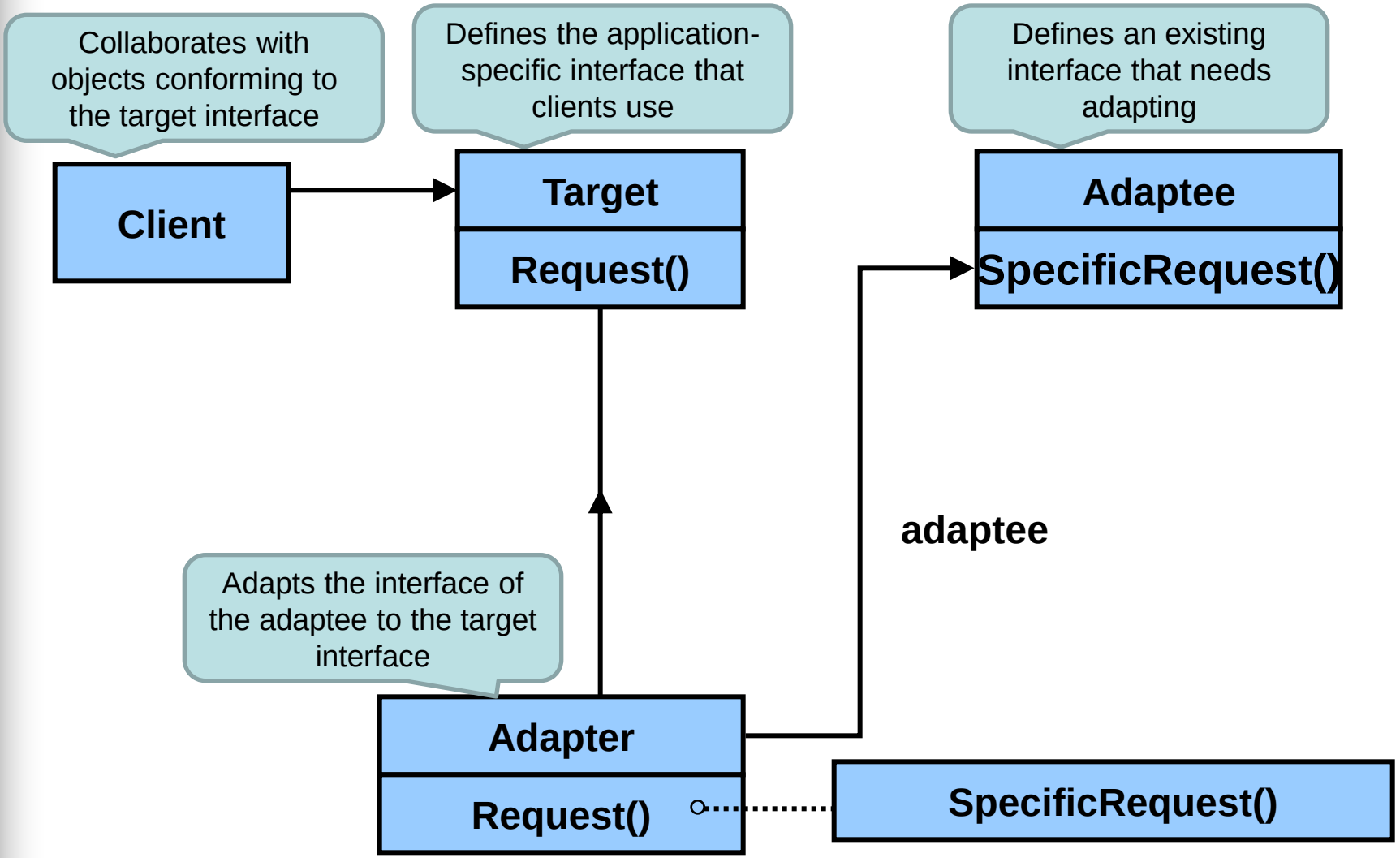
Participants of the Adapter Pattern

- **Target:** Defines the application-specific interface that clients use.
- **Client:** Collaborates with objects conforming to the target interface.
- **Adaptee:** Defines an existing interface that needs adapting.
- **Adapter:** Adapts the interface of the adaptee to the target interface.

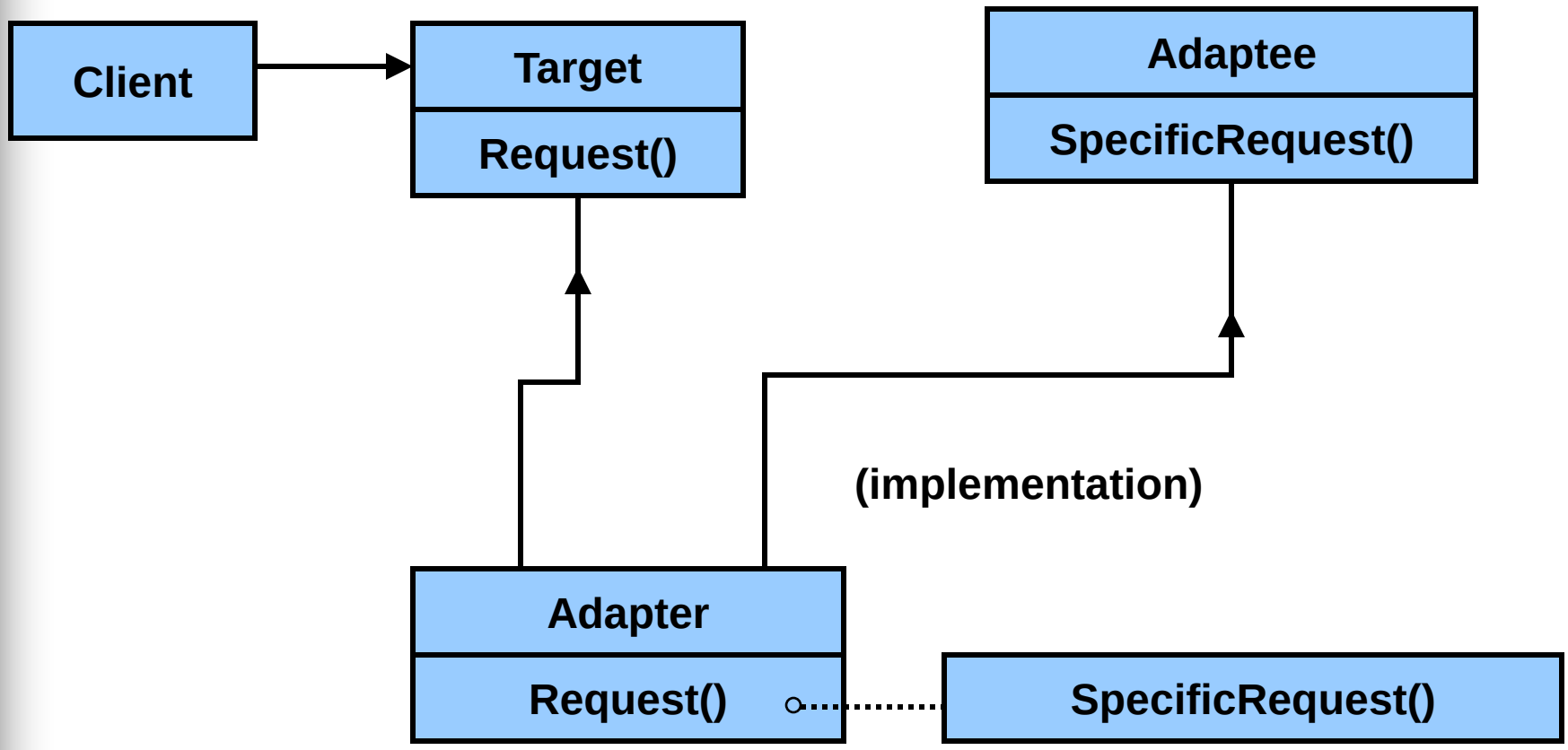
Adapter


Lets users draw and arrange graphical elements
Editor
Shape
BoundingBox()
CreateManipulator()
Interface for graphical object
OTS UI toolkit. Provides sophisticated class for displaying and editing text
TextView
GetExtent()
text
One can change TextView class so it conforms to Shape interface … would need source code of TextView. Too much work!
LineShape
BoundingBox()
CreateManipulator()
Subclass of shape defined by editor for lines
TextShape
BoundingBox()
CreateManipulator()
Define TextShape to adapt TextView interface to Shape's
BoundingBox requests are converted to GetExtent requests
return text -> GetExtent()
return new Text Manipulator
Allows objects to be 'dragged' interactively

Adapter Pattern Structure (Object Adapter)



Structure of the Adapter Pattern Using Multiple Inheritance (Class Adapter)



Applicability

- Use an existing class when its interface does not match the one you need
- Create a class that cooperates with unrelated or unforeseen classes with incompatible interfaces
- Object Adapter Only
  - Need to use several existing subclasses, but it is impractical to adapt by sub-classing each one of them
    - Object adapter adapts the interface of the parent class

Tradeoffs

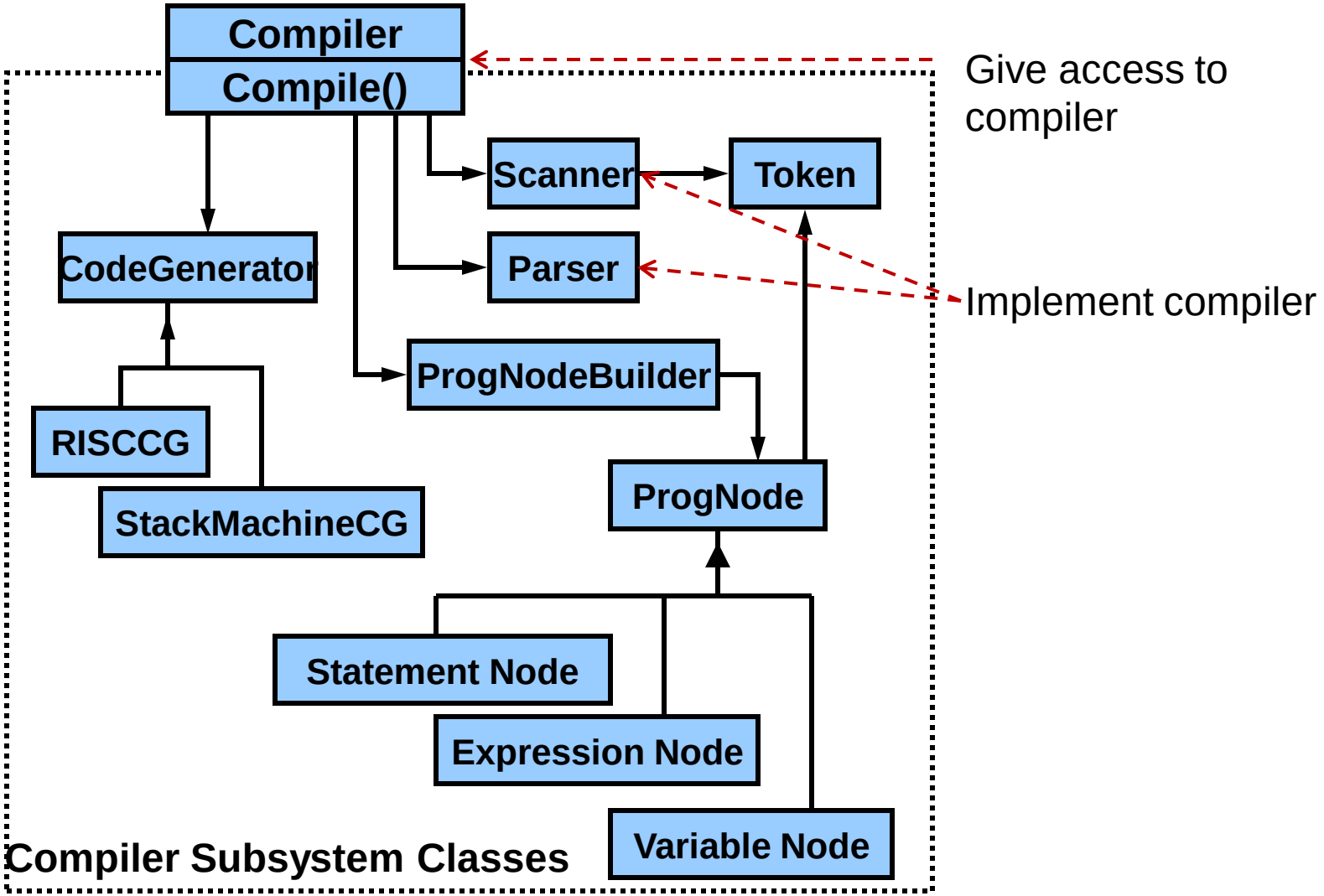
- A class adapter – inheritance
  - Adapts Adaptee to Target by committing to concrete Adapter class
    - Class adapter is not useful when we want to adapt a class and all its subclasses
  - Lets Adapter override some of Adaptee's behaviour
    - Adapter is a subclass of Adaptee
  - Introduces only one object
    - No additional pointer indirection is needed to get to Adaptee
- An object adapter – uses
  - One Adapter can work with many Adaptees
    - Adaptee and all its subclasses
    - Can add functionality to all Adaptees at once
  - Makes it harder to override Adaptee behaviour
    - Requires making ADAPTER refer to the subclass rather than the ADAPTEE itself, Or
    - Subclassing ADAPTER for each ADAPTEE subclass

The Façade Design Pattern

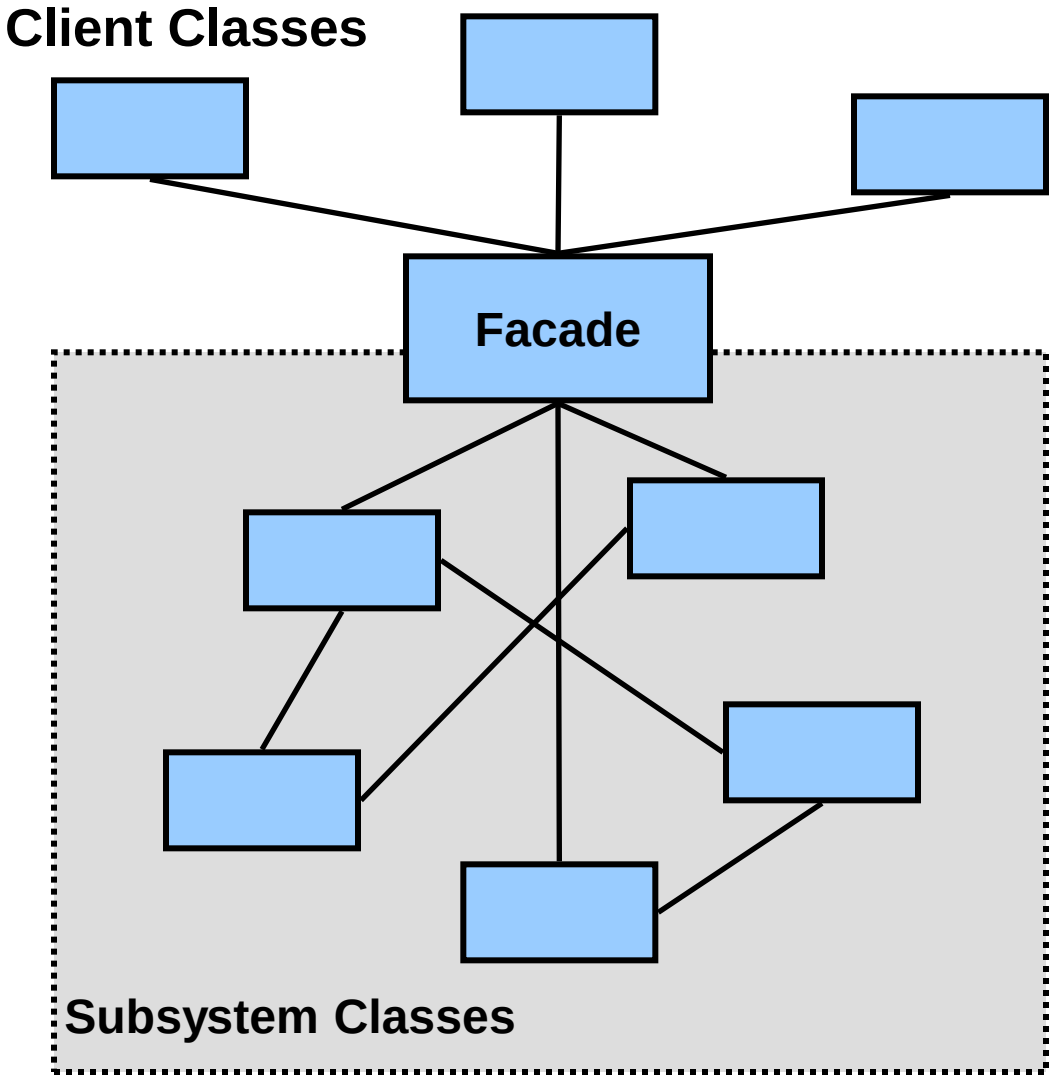
The Façade Pattern

- **Motivation**
  - Structuring a system into subsystems helps reduce complexity.
  - A common design goal is to minimize the communication and dependencies between subsystems.
  - Use a facade object to provide a single, simplified interface to the more general facilities of a subsystem.
- **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.

Façade Example – Programming Environment



Structure of the Facade Pattern



Participants of the Facade Pattern

- **Facade:**
  - Knows which subsystem classes are responsible for a request.
  - Delegates client requests to appropriate subsystem objects.
- **Subsystem Classes:**
  - Implement subsystem functionality.
  - Handle work assigned by the facade object.
  - Have no knowledge of the facade; that is, they keep no references to it.

Participants of Façade Pattern

- Façade (compiler)
  - Knows which subsystem classes are responsible for a request
  - Delegates client requests to appropriate subsystem objects
- Subsystem classes (Scanner, Parser,etc..)
  - Implements subsystem functionality
  - Handles work assigned by the façade object

Façade Pattern Applicability

- Use a façade when
  - To provide a simple interface to a complex subsystem
  - To decouple clients and implementation classes
  - To define an entry point to a layered subsystem

Adapter vs. Façade

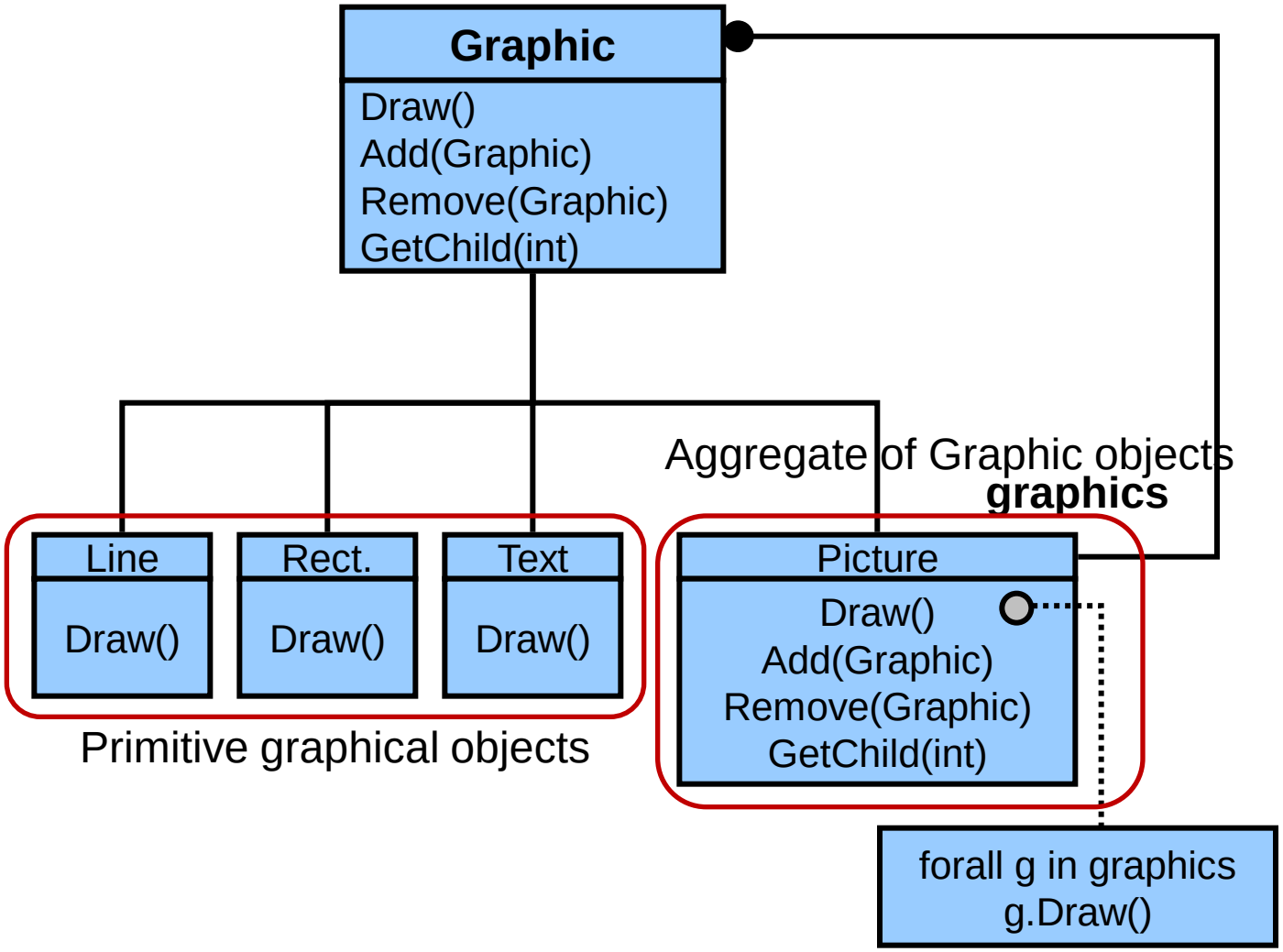
- The basic idea is similar, but …
- Adapter typically changes the interface of a single class into something more natural for clients
  - Adapter adds a “human face” to a grungy abstraction, but does not change the adaptee
  - Adapter are often applied to
    - Library entities external to your system that you want to use, or
    - Components of your system that external clients will use
- Façade puts a human face on a whole system
  - A façade usually requires changing the core code to make all accessing go through the façade
  - Typically this is done to (older) pieces of your system when you are performing a redesign

The Composite Design Pattern

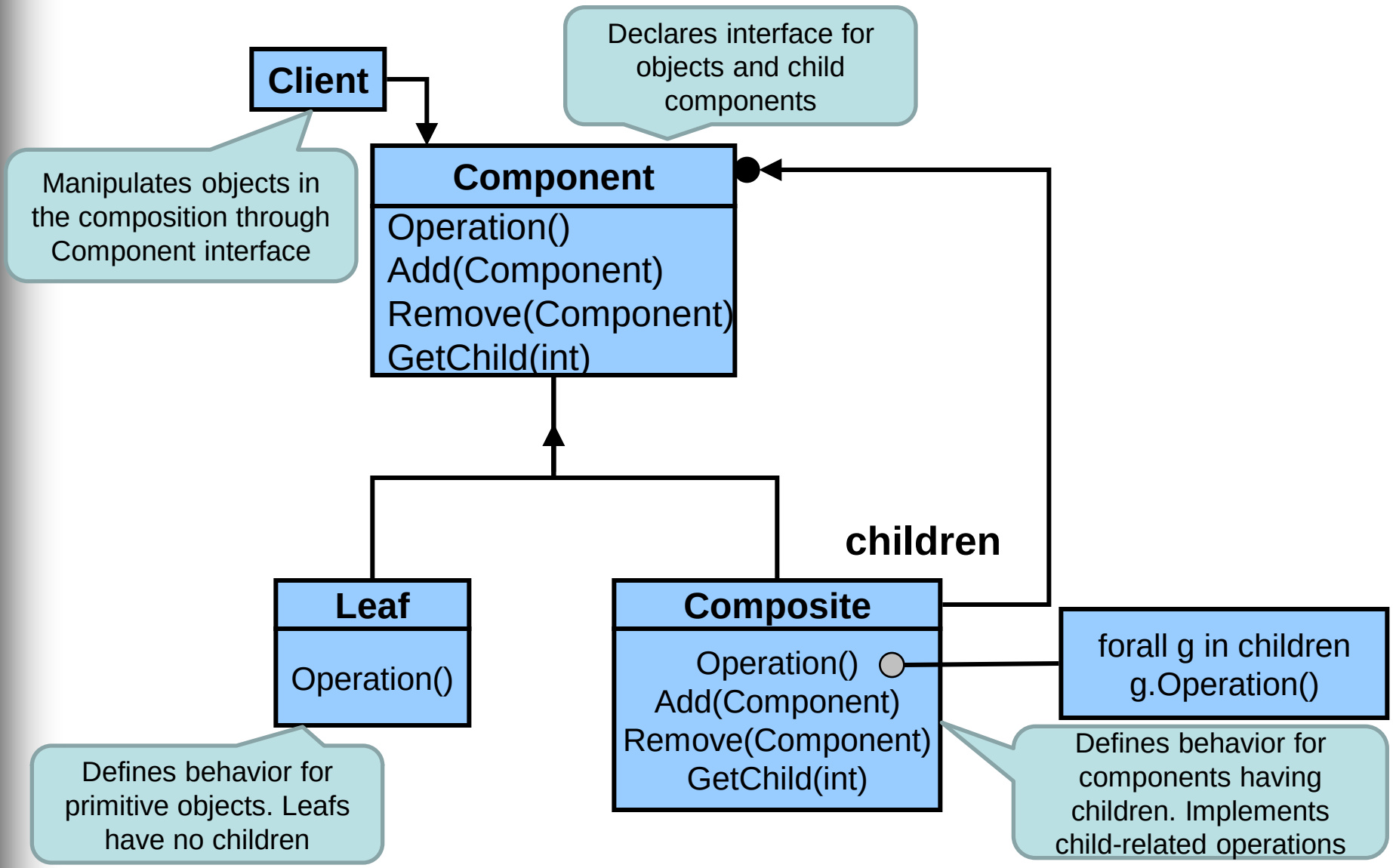
Composite Pattern

- **Motivation**
  - Applications that have recursive groupings of primitives and groups (containers)
    - Drawing programs
      - Lines, text, figures and groups
    - Directory structure
      - Folders and files
  - Operations on groups are different than primitives but clients treat them in the same way
- **Intent**
  - Compose objects into tree structures representing part-whole hierarchies
  - Clients deal uniformly with individual objects and hierarchies of objects

Composite Pattern Example



Structure of Composite Pattern



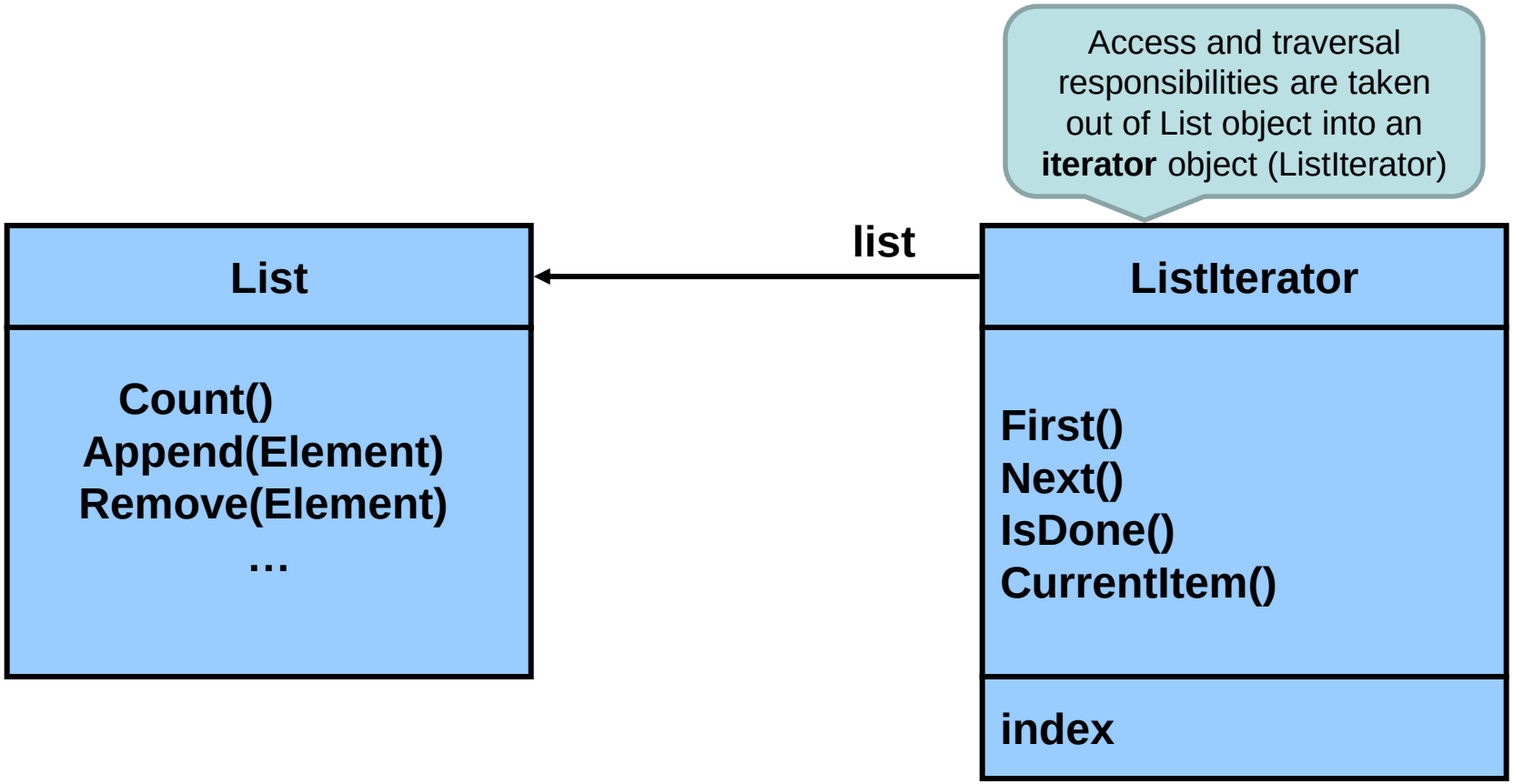
Behavioral Design Patterns - Iterator, Observer, Template, Master/Slave

The Iterator Design Pattern

The Iterator Pattern

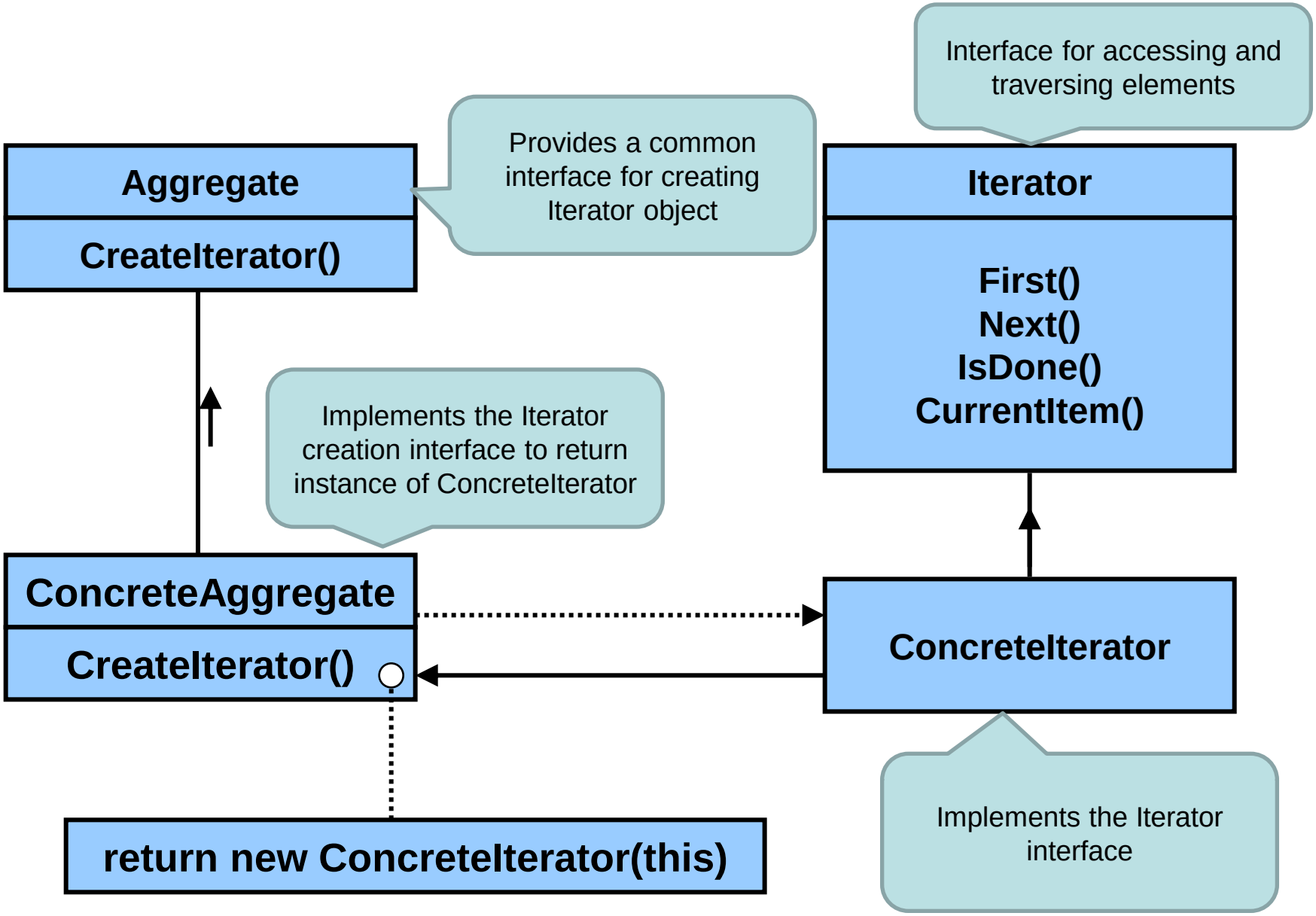
- **Motivation**
  - Don't want to expose implementation details of the container AND want to allow multiple simultaneous traversals
    - Create separate interface/class that provide simple "hooks"
  - Often we want to say to a container (e.g., tree, graph, table, list, graphics):
    - Apply f() to each of your objects
- **Intent**
  - Provide a clean, abstract way to access all of the elements in an aggregate (container) without exposing the underlying representation
  - Move responsibility for access and traversal to a separate "iterator" object

Iterator Pattern Example



Can define different traversal policies without enumerating them in the List interface

Structure of Iterator Pattern

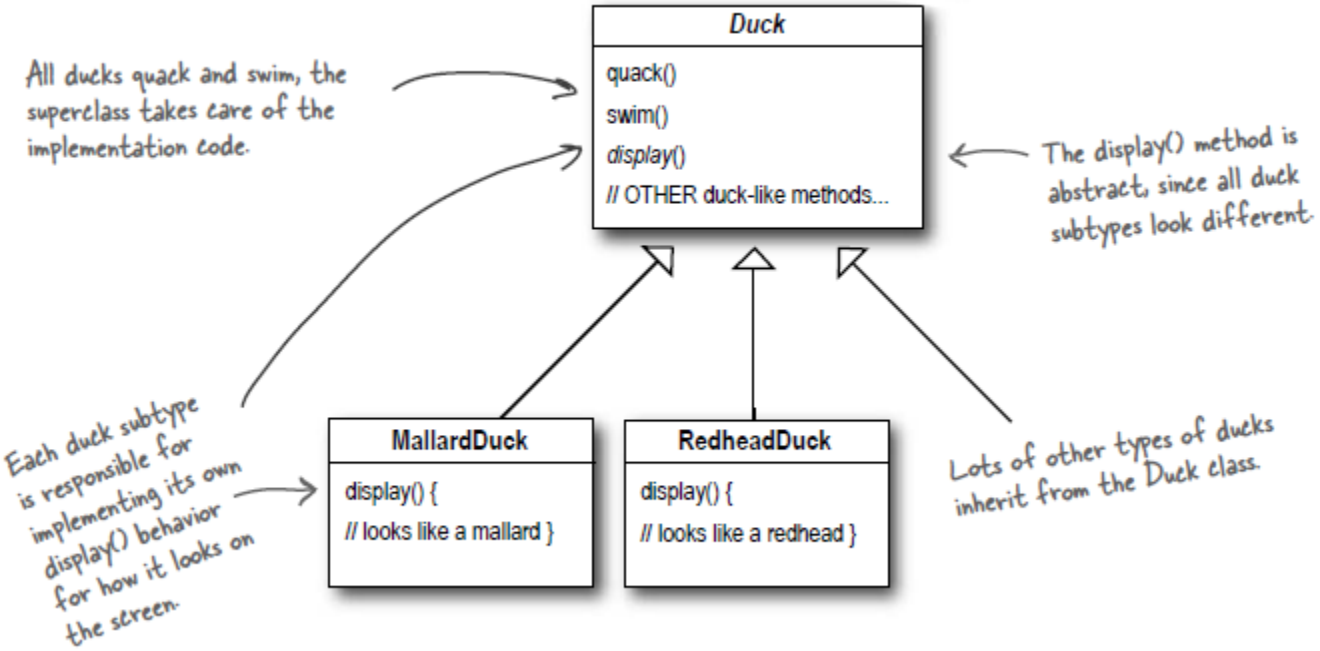


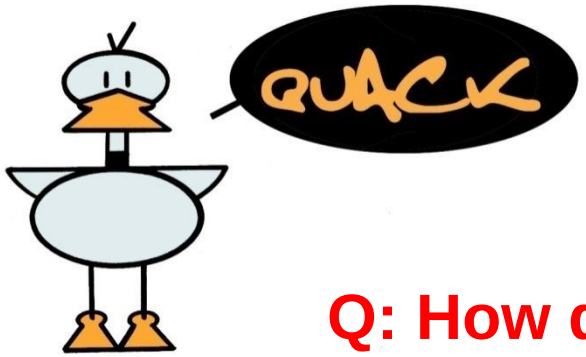
The Strategy Design Pattern

The Strategy Pattern

- **Motivation**
  - Have a problem with multiple well-defined solutions that conform to a common interface
  - Want to let client vary the implementation according to particular needs. Variation due to either:
    - Different functionality (e.g., justification styles)
    - Same functionality, but different non-functional attributes (efficiency, trust, debugging input)
      - E.g., sorting routines, resource allocation strategies
  - Implementation is mostly decoupled from client code. Most binding is to abstract parent interface
- **Intent**
  - Define a family of related algorithms behind a common interface

superclass DUCK





Q: How do we make the ducks fly?

Problem: Find a design?

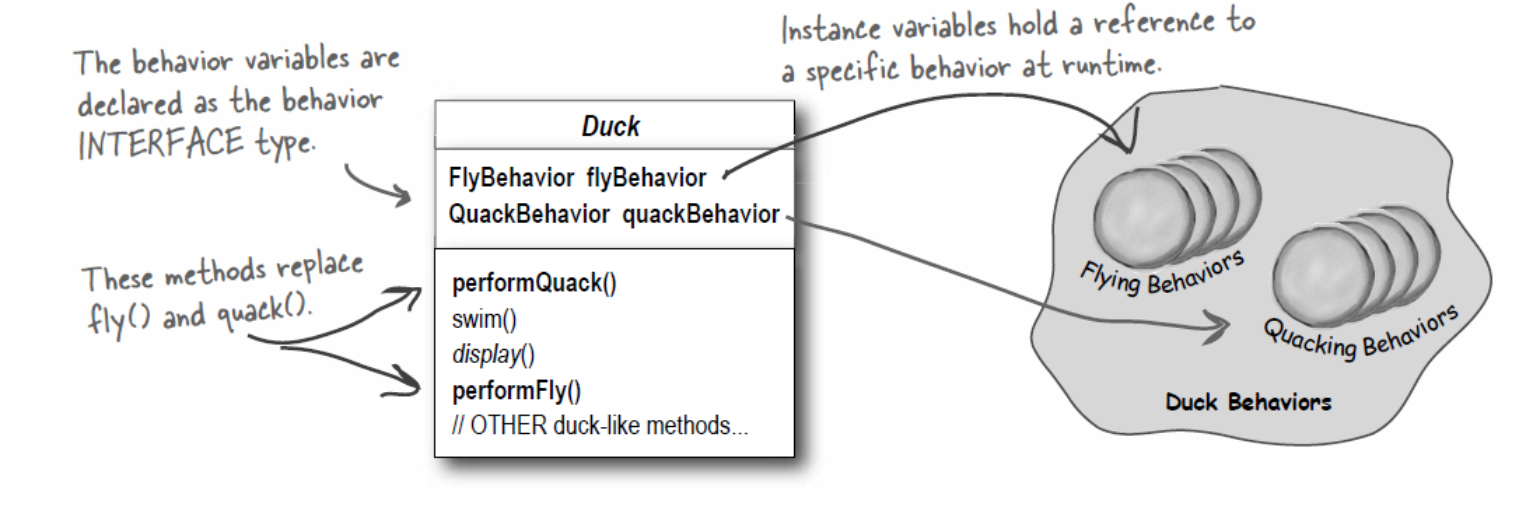
- Most ducks fly in the same way, but a few species of duck have different flyable behaviour, or perhaps they do not fly at all
- Most ducks quack in the same way, but a few species have a different quackable behaviour or they do not quack at all
- Must be able to change flyable and quackable behavior dynamically

- Inheritance hasn't worked out very well, since the duck behavior keeps changing across the subclasses, and it's not appropriate for all subclasses to have those behaviors.

- The Flyable and Quackable interface sounded promising at first — only ducks that really do fly will be Flyable, etc., — except Java interfaces have no implementation code, so no code reuse.

- And that means that whenever you need to modify a behavior, you're forced to track down and change it in all the different subclasses where that behavior is defined, probably introducing new bugs along the way!

Delegation



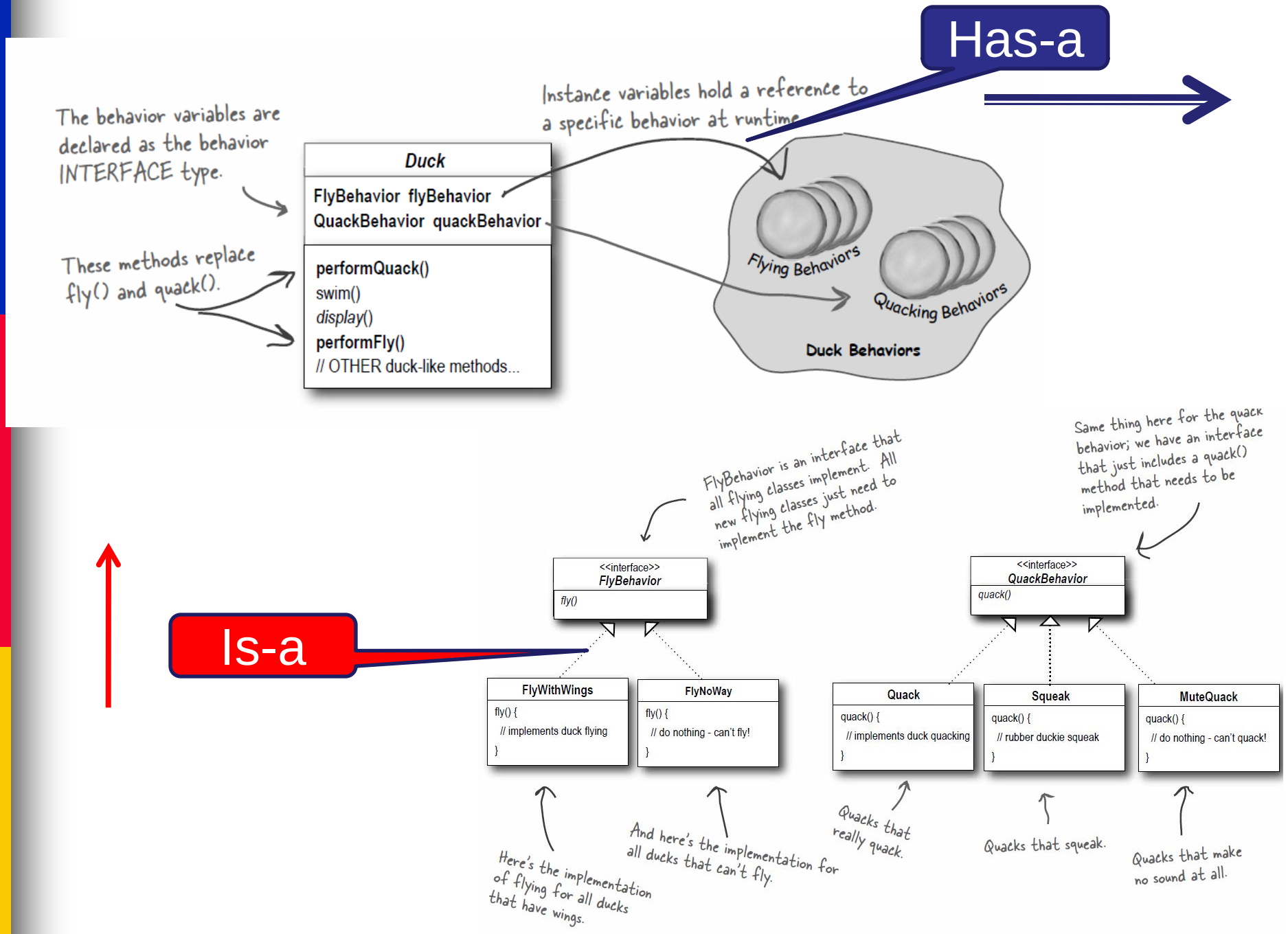
Now we implement performQuack():

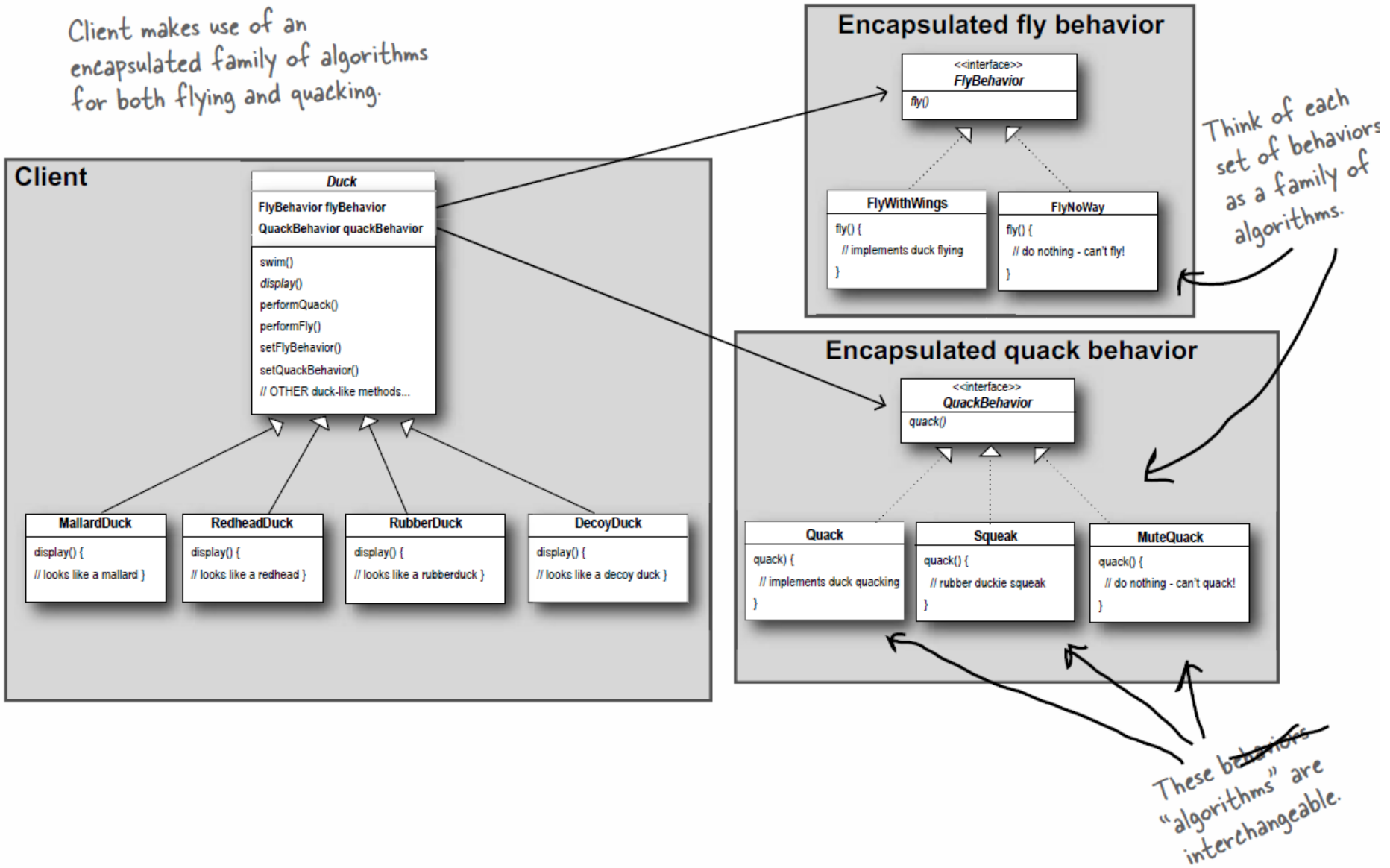
```
public class Duck {
    QuackBehavior quackBehavior;
    // more

    public void performQuack() {
        quackBehavior.quack();
    }
}
```

Each Duck has a reference to something that implements the QuackBehavior interface.

Rather than handling the quack behavior itself, the Duck object delegates that behavior to the object referenced by quackBehavior.


Has-a
The behavior variables are declared as the behavior INTERFACE type.
Instance variables hold a reference to a specific behavior at runtime.
Duck
FlyBehavior flyBehavior
QuackBehavior quackBehavior
These methods replace fly() and quack().
performQuack()
swim()
display()
performFly()
// OTHER duck-like methods...
Flying Behaviors
Quacking Behaviors
Duck Behaviors
Is-a
FlyBehavior is an interface that all flying classes implement. All new flying classes just need to implement the fly method.
Same thing here for the quack behavior; we have an interface that just includes a quack() method that needs to be implemented.
<<interface>>
FlyBehavior
fly()
<<interface>>
QuackBehavior
quack()
FlyWithWings
fly() {
// implements duck flying
}
FlyNoWay
fly() {
// do nothing - can't fly!
}
Quack
quack() {
// implements duck quacking
}
Squeak
quack() {
// rubber duckie squeak
}
MuteQuack
quack() {
// do nothing - can't quack!
}
Here's the implementation of flying for all ducks that have wings.
And here's the implementation for all ducks that can't fly.
Quacks that really quack.
Quacks that squeak.
Quacks that make no sound at all.

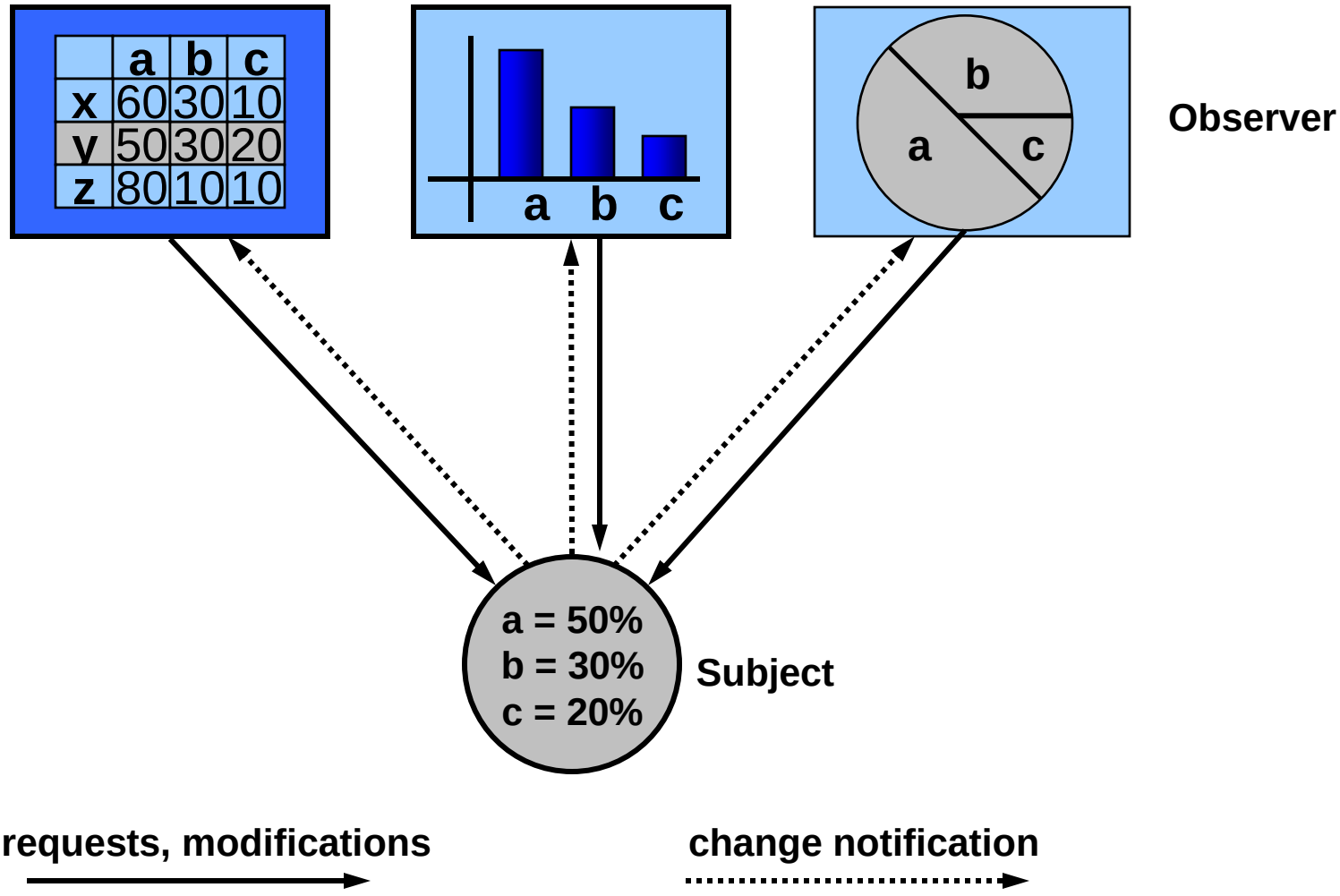

Client makes use of an encapsulated family of algorithms for both flying and quacking.
Client
Duck
FlyBehavior flyBehavior
QuackBehavior quackBehavior
swim()
display()
performQuack()
performFly()
setFlyBehavior()
setQuackBehavior()
// OTHER duck-like methods...
MallardDuck
display() {
// looks like a mallard }
RedheadDuck
display() {
// looks like a redhead }
RubberDuck
display() {
// looks like a rubberduck }
DecoyDuck
display() {
// looks like a decoy duck }
Encapsulated fly behavior
<<interface>>
FlyBehavior
fly()
FlyWithWings
fly() {
// implements duck flying
}
FlyNoWay
fly() {
// do nothing - can't fly!
}
Think of each set of behaviors as a family of algorithms.
Encapsulated quack behavior
<<interface>>
QuackBehavior
quack()
Quack
quack) {
// implements duck quacking
}
Squeak
quack() {
// rubber duckie squeak
}
MuteQuack
quack() {
// do nothing - can't quack!
}
These behaviors "algorithms" are interchangeable.

The Observer Design Pattern

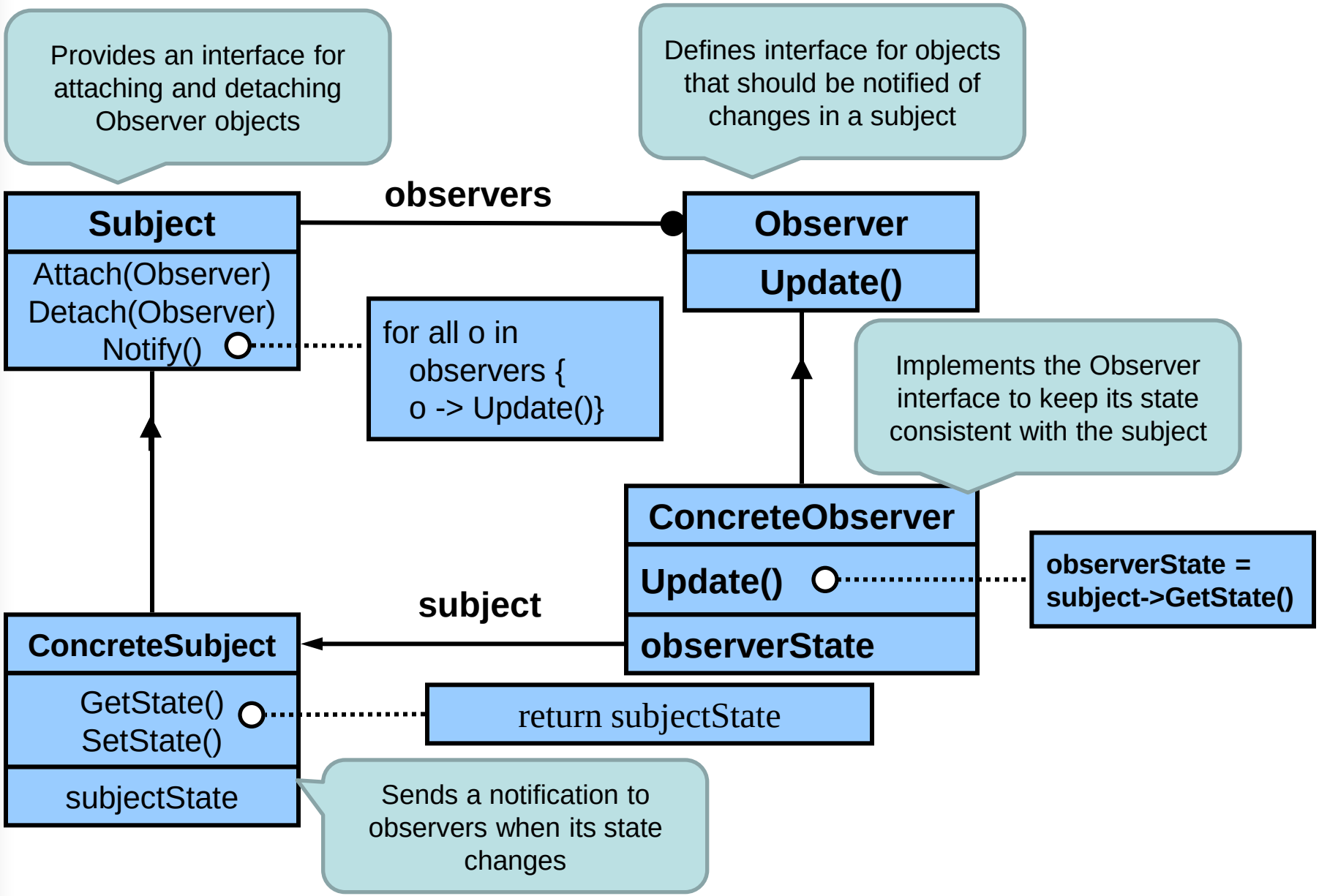
The Observer Pattern

- **Motivation**
  - A common side-effect of partitioning a system into a collection of cooperating classes is the need to maintain consistency between related objects.
  - How can you achieve consistency?
- Intent
  - Define a one-to-many dependency between objects so that when **one object changes state**, **all its dependents** are **notified** and **updated automatically**.

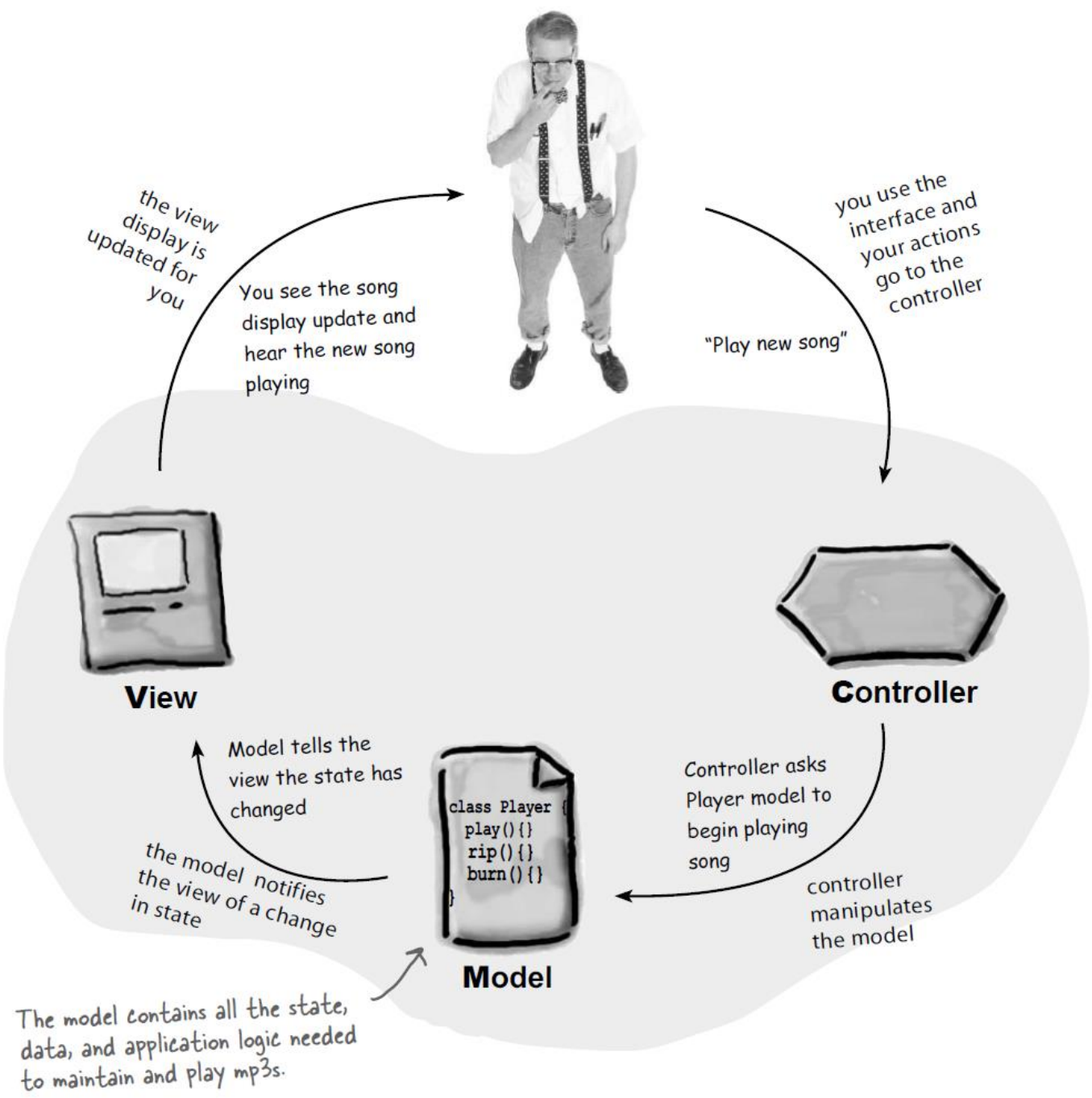
Observer Pattern Example

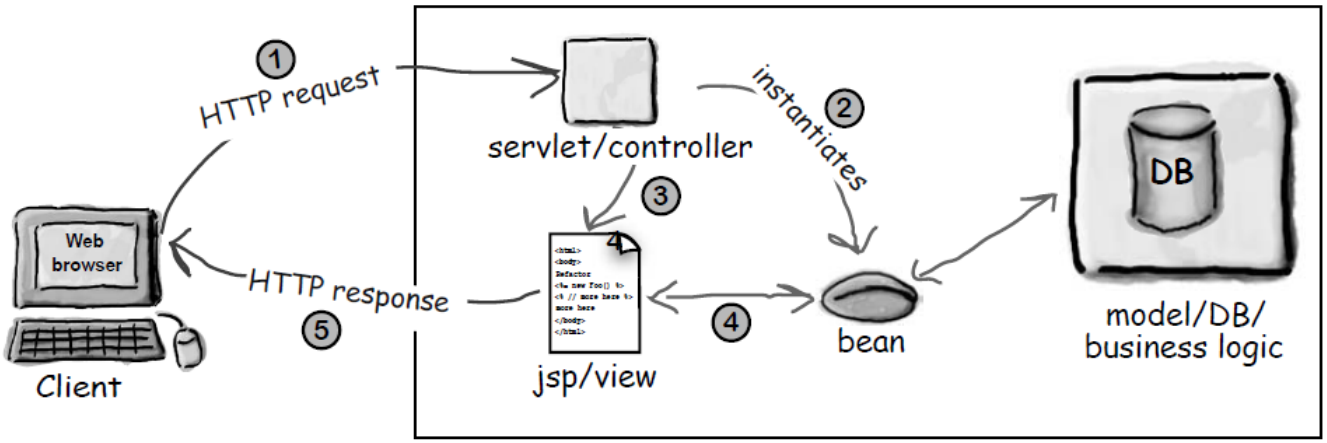


Observer Pattern Structure



MVC
iTunes
mp3 player





1. **You make an HTTP request, which is received by a servlet.**
   Using your web browser you make an HTTP request. This typically involves sending along some form data, like your username and password. A servlet receives this form data and parses it.

2. **The servlet acts as the controller.**
   The servlet plays the role of the controller and processes your request, most likely making requests on the model (usually a database). The result of processing the request is usually bundled up in the form of a JavaBean.

3. **The controller forwards control to the view.**
   The View is represented by a JSP. The JSP's only job is to generate the page representing the view of model (4 which it obtains via the JavaBean) along with any controls needed for further actions.

5. **The view returns a page to the browser via HTTP.**
   A page is returned to the browser, where it is displayed as the view. The user submits further requests, which are processed in the same fashion.

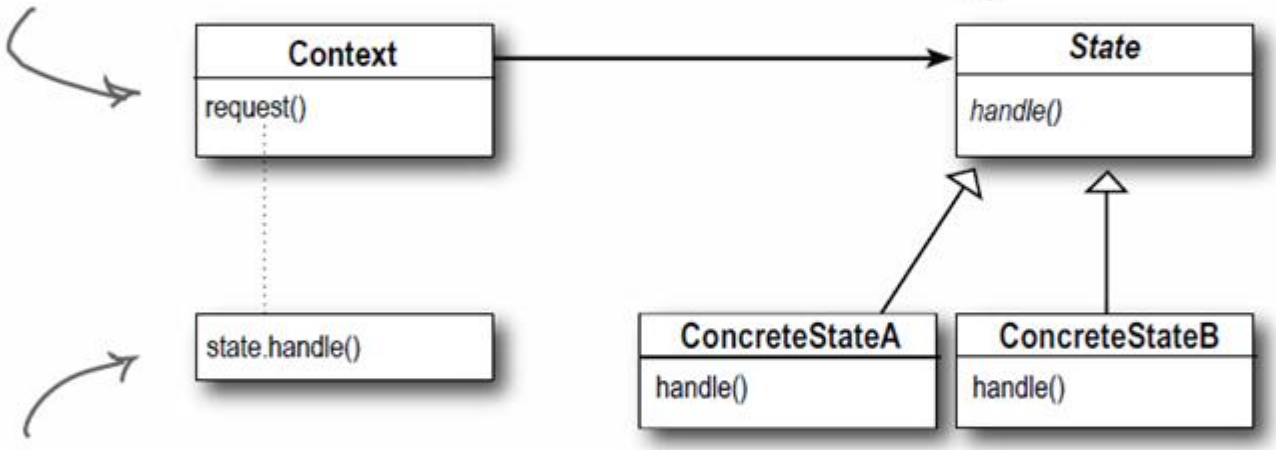
The State Design Pattern

The State Design Pattern

- **Motivation**
  - An object may be in one of many states. It responds differently depending upon its current state
- **Intent**
  - Alter behaviour of an object when its internal state changes
  - Object appears to change its class

The Context is the class that can have a number of internal states.

The State interface defines a common interface for all concrete states; the states all implement the same interface, so they are interchangeable.

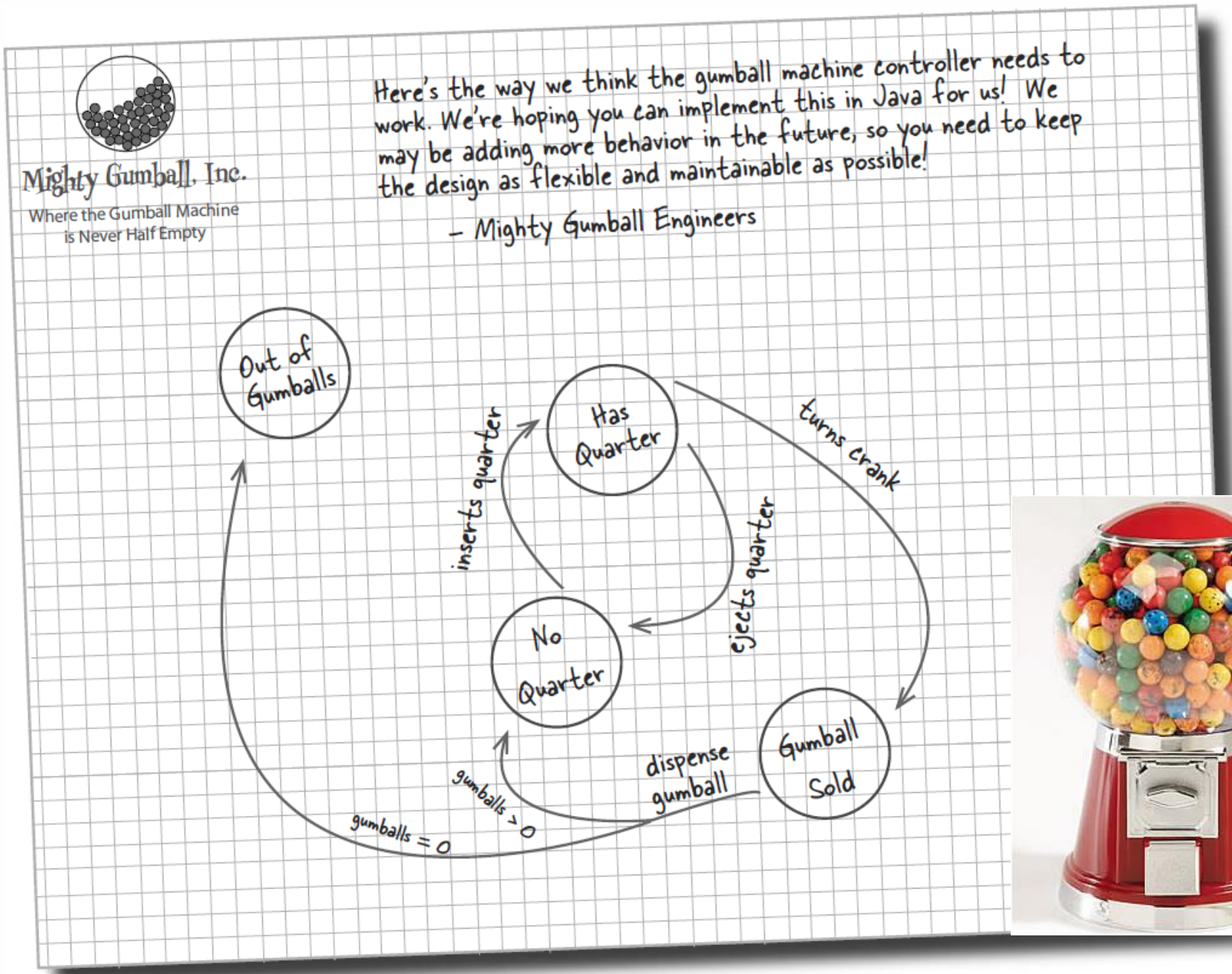


Whenever the request() is made on the Context it is delegated to the state to handle.

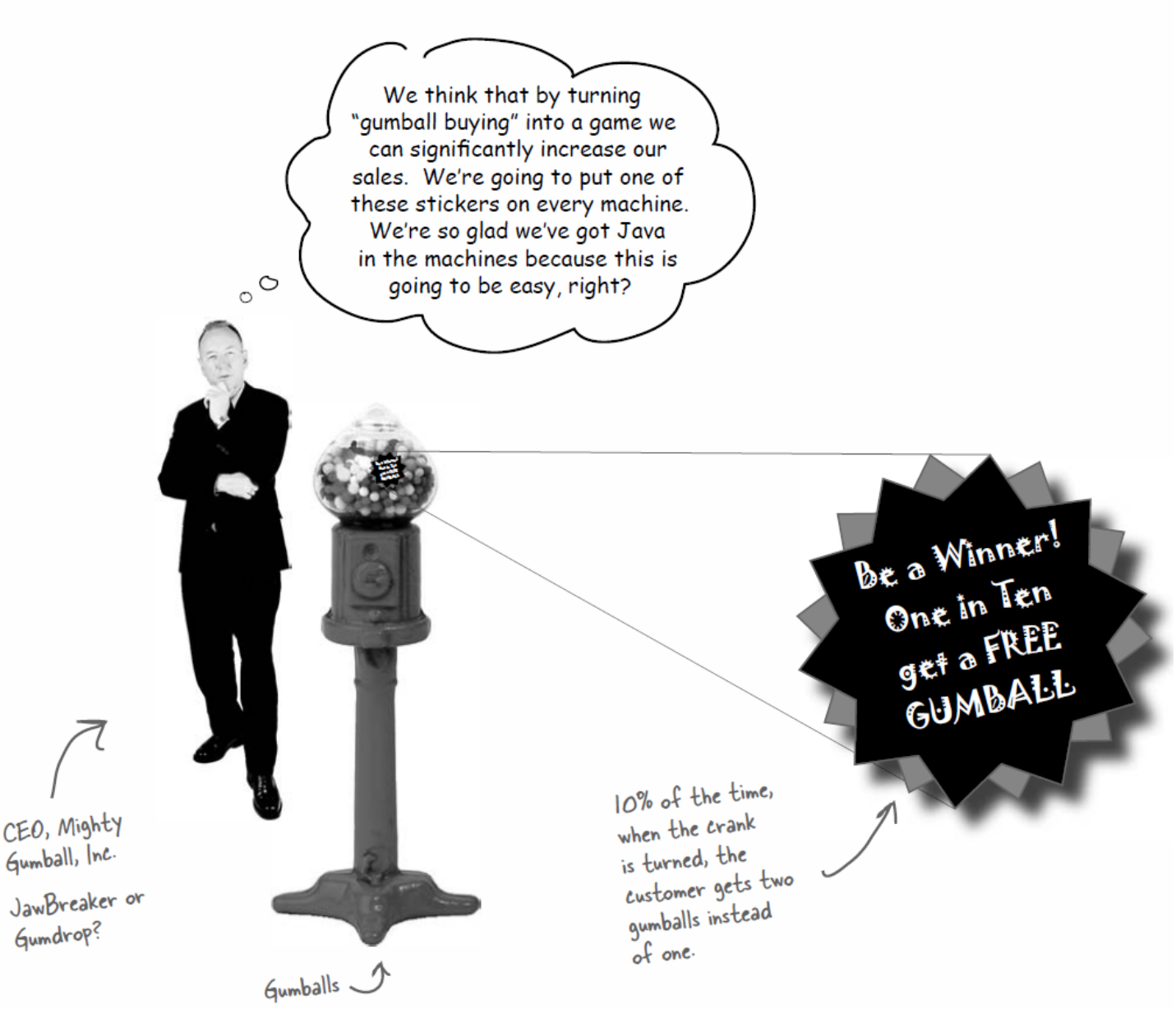
Many concrete states are possible.

ConcreteStates handle requests from the Context. Each ConcreteState provides its own implementation for a request. In this way, when the Context changes state, its behavior will change as well.

Exercise: Gumball Example



adding new states



State vs. Strategy

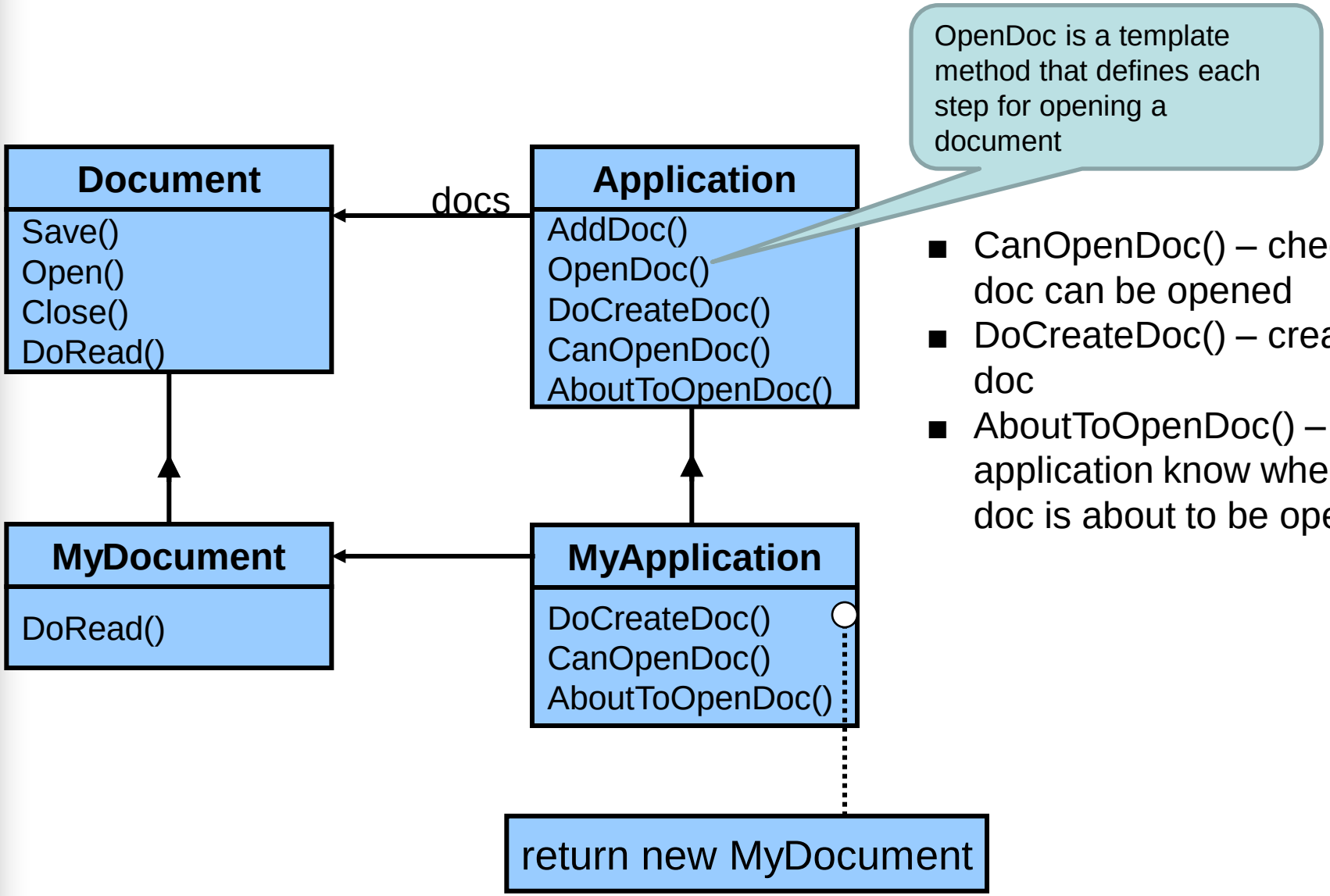
- The class diagrams are similar but they differ in intent
- State
  - Behaviours are constantly changing over time and the client (context) knows very little about how those different behaviours work
  - Encapsulate behaviours in state objects and set change in the context
  - Alternative to putting a lot of conditional statements in the context
- Strategy
  - Client knows quite a lot about what behaviour (strategy) is most appropriate e.g., we know that a mallard duck has typical flying behaviour and a decoy duck never flies
  - Change in state less usual
  - Flexible alternative to subclassing

The Template Design Pattern

The Template Pattern

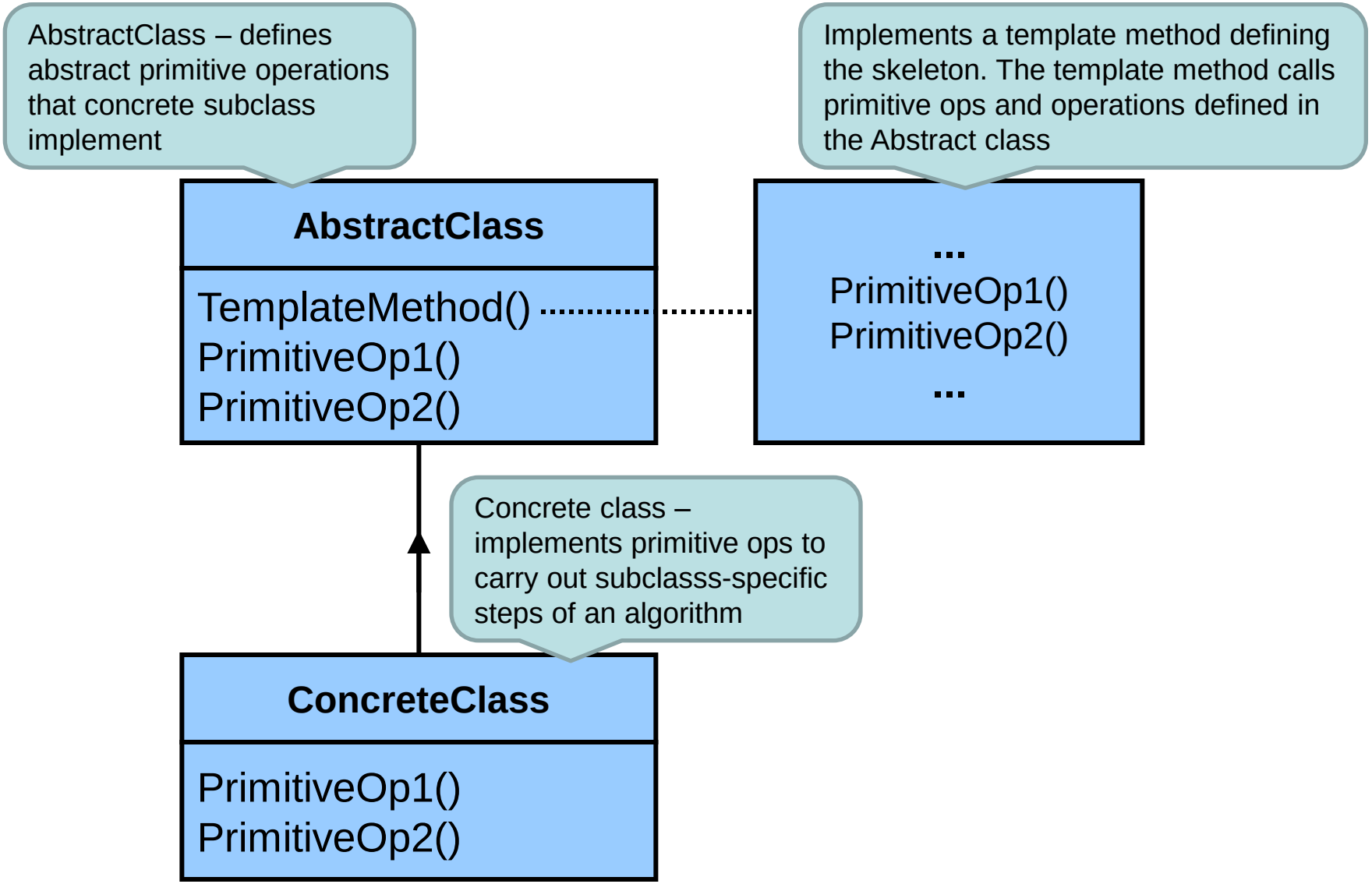
- **Motivation**
  - Consider an application that provides Application and Document classes
    - Application: opens existing document
    - Document: represents the information in a doc
  - By defining some of the steps of an algorithm, using abstract operations, the template method fixes their ordering.
  - Specific applications can subclass Application and Document to suit their specific needs
    - Drawing application: defines DrawApplication and DrawDocument sublclasees
    - Spreadsheet application: defines SpreadsheetApplication and SpreadsheetDocument sublclasees
- **Intent**
  - Define the skeleton of an algorithm in an operation, deferring some steps to subclasses.
  - The *Template pattern* lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

Template Pattern Example



- CanOpenDoc() – check if doc can be opened
- DoCreateDoc() – create doc
- AboutToOpenDoc() – lets application know when a doc is about to be opened

Template Pattern Structure

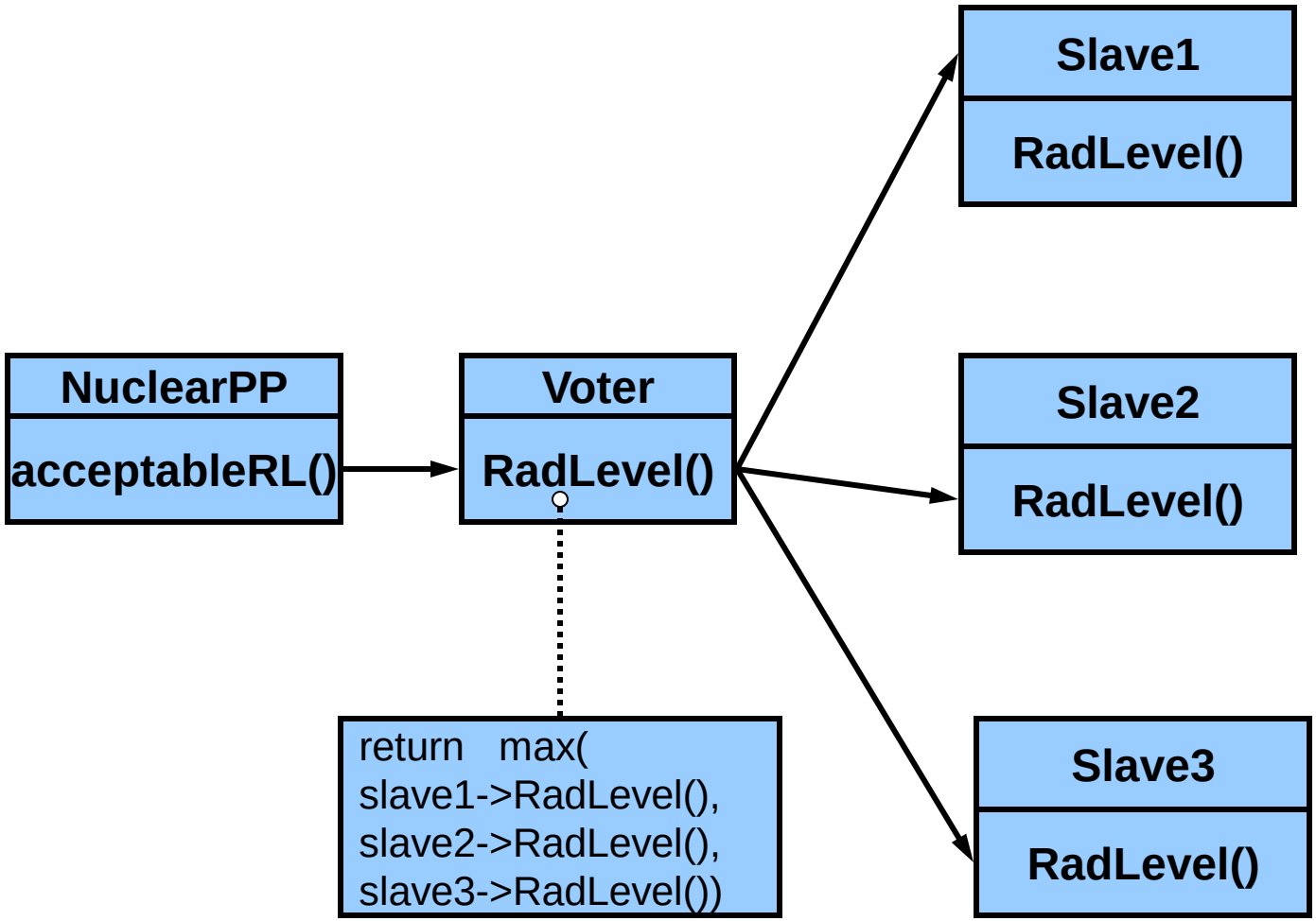


The Master-Slave Design Pattern

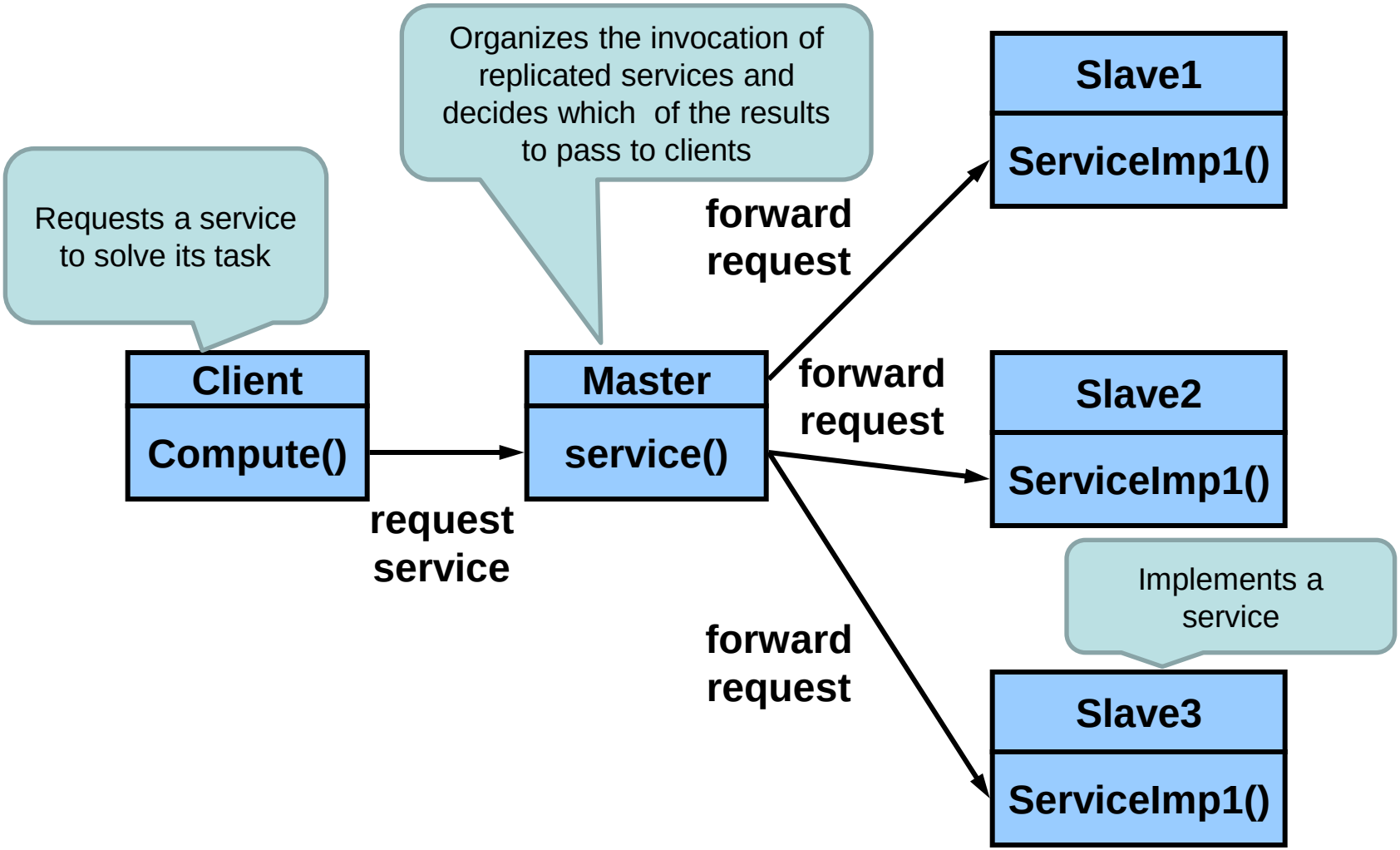
The Master-Slave Pattern

- **Motivation**
  - Fault tolerance is a critical factor in many systems.
  - Replication of services and delegation of the same task to several independent suppliers is a common strategy to handle such cases.
- **Intent**
  - Independent components providing the same service (slaves) are separated from a component (master) responsible for invoking them and for selecting a particular result from the results returned by the slaves.
  - (Master) Handles the computation of replicated services within a software system to achieve fault tolerance and robustness.

Master-Slave Pattern Example



Master-Slave Pattern Structure



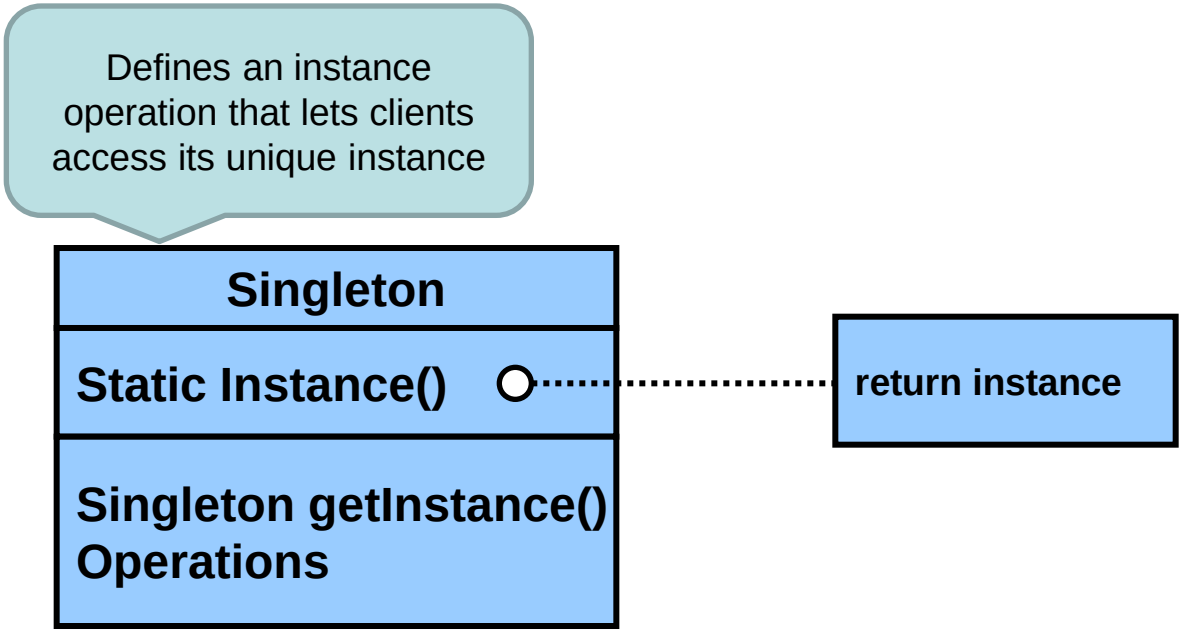
Creational Design Patterns
- Singleton, Abstract Factory

The Singleton Design Pattern

The Singleton Pattern

- **Motivation**
  - Some classes must only have one instance file system, window manager
- **Intent**
  - Ensure a class has only one instance
  - Provide a global point of access
- **Applicability**
  - Must have only one instance of a class
  - Must be accessible from a known location

Singleton Pattern Structure



Singleton example (Java)

```
public class SimpleSingleton {
    private SimpleSingleton singleInstance = null;

    //Marking default constructor private
    //to avoid direct instantiation.
    private SimpleSingleton() {
    }

    //Get instance for class SimpleSingleton
    public static SimpleSingleton getInstance() {

        if(null == singleInstance) {
            singleInstance = new SimpleSingleton();
        }

        return singleInstance;
    }
}
```

http://viralpatel.net/blogs/2009/01/java-singleton-design-pattern-tutorial-example-singleton-j2ee-design-pattern.html

The Abstract Factory Design Pattern

The Abstract Factory Pattern

- **Motivation**
  - Sometimes we have systems that support different representations depending on external factors.
  - The *Abstract Factory* pattern provides an interface for the client. In this way the client can obtain a specific object through this abstract interface.
- **Intent**
  - Provides an interface for creating families of related or dependent objects without specifying their concrete classes

Abstract Factory Example

- UI toolkit supports multiple look-and-feel standards
  - Motif and Presentation Manager
- Look and feel (LnF) standards define appearance and behavior of UI widgets (e.g. scroll bars and windows)
- To be portable, should not hard code LnF standards

Abstract Factory Example