Design – using OO patterns

Some of these slides are adapted from *Software Engineering: A Practitioner's Approach,* by Roger Pressman.

Design Patterns

- Each of us has encountered a design problem and silently thought: *I wonder if anyone has developed a solution to for this?*
  - What if there was a standard way of describing a problem (so you could look it up), and an organized method for representing the solution to the problem?
- *Design patterns* are a codified method for describing problems and their solution allows the software engineering community to capture design knowledge in a way that enables it to be reused.

Design Patterns

- *Each pattern describes a problem that occurs over and over again in our environment and then describes the core of the solution to that problem in such a way that you can use the solution a million times over without ever doing it the same way twice.*
  - Christopher Alexander, 1977

- "a three-part rule which expresses a relation between a certain context, a problem, and a solution."

Basic Concepts

- *Context* allows the reader to understand the environment in which the problem resides and what solution might be appropriate within that environment.

- A set of requirements, including limitations and constraints, acts as a *system of forces* that influences how
  - the problem can be interpreted within its context and
  - how the solution can be effectively applied.

Describing a Pattern

- *Pattern name*—describes the essence of the pattern in a short but expressive name
- *Problem*—describes the problem that the pattern addresses
- *Motivation*—provides an example of the problem
- *Context*—describes the environment in which the problem resides including application domain
- *Forces*—lists the system of forces that affect the manner in which the problem must be solved; includes a discussion of limitation and constraints that must be considered
- *Solution*—provides a detailed description of the solution proposed for the problem
- *Intent*—describes the pattern and what it does
- *Collaborations*—describes how other patterns contribute to the solution
- *Consequences*—describes the potential trade-offs that must be considered when the pattern is implemented and the consequences of using the pattern
- *Implementation*—identifies special issues that should be considered when implementing the pattern
- *Known uses*—provides examples of actual uses of the design pattern in real applications
- *Related patterns*—cross-references related design patterns

Design patterns (GOF)

- Design Patterns communicate solutions to common problems.
  - It's a problem-solution pair.
- The seminal book on design patterns, *Design Patterns, Elements of Reusable Object-Oriented Software* by Gamma et al, identifies three categories of design patterns
  - Creational
  - Structural
  - Behavioral

Kinds of Patterns

- ***Creational patterns*** focus on the "creation, composition, and representation of objects, e.g.,
  - **Singleton pattern:** Control the creation of instances to just on3
  - **Abstract factory pattern:** centralize decision of what factory to instantiate
  - **Factory method pattern:** centralize creation of an object of a specific type choosing one of several implementations

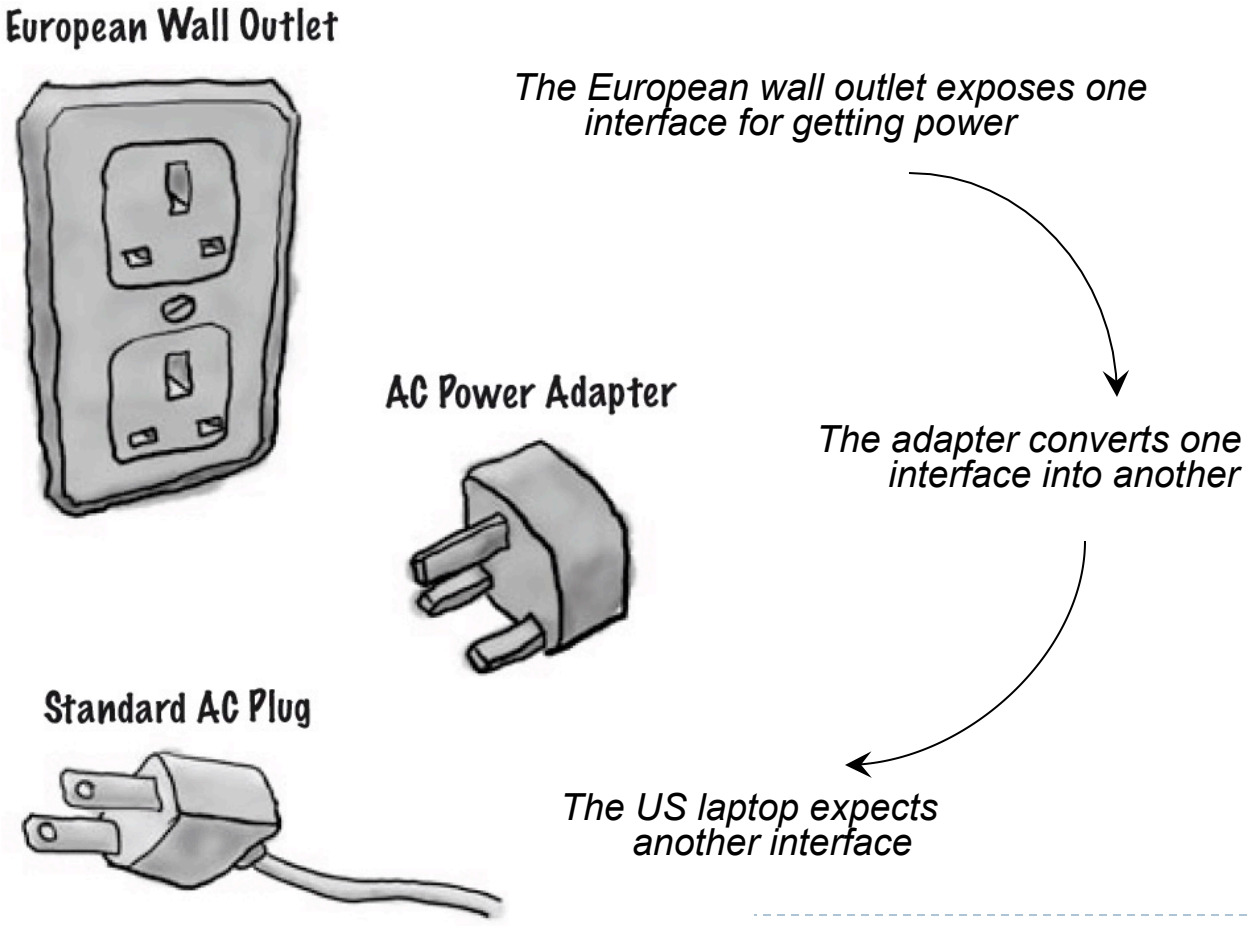
- ***Structural patterns*** focus on problems and solutions associated with how classes and objects are organized and integrated to build a larger structure, e.g.,
  - **Adapter pattern:** 'adapts' one interface for a class into one that a client expects
  - **Aggregate pattern:** a version of the Composite pattern with methods for aggregation of children

- ***Behavioral patterns*** address problems associated with the assignment of responsibility between objects and the manner in which communication is effected between objects, e.g.,
  - **Chain of responsibility pattern:** Command objects are handled or passed on to other objects by logic-containing processing objects
  - **Command pattern:** Command objects encapsulate an action and its parameters
  - **Observer pattern:** Enable loose coupling between publishers and subscribers

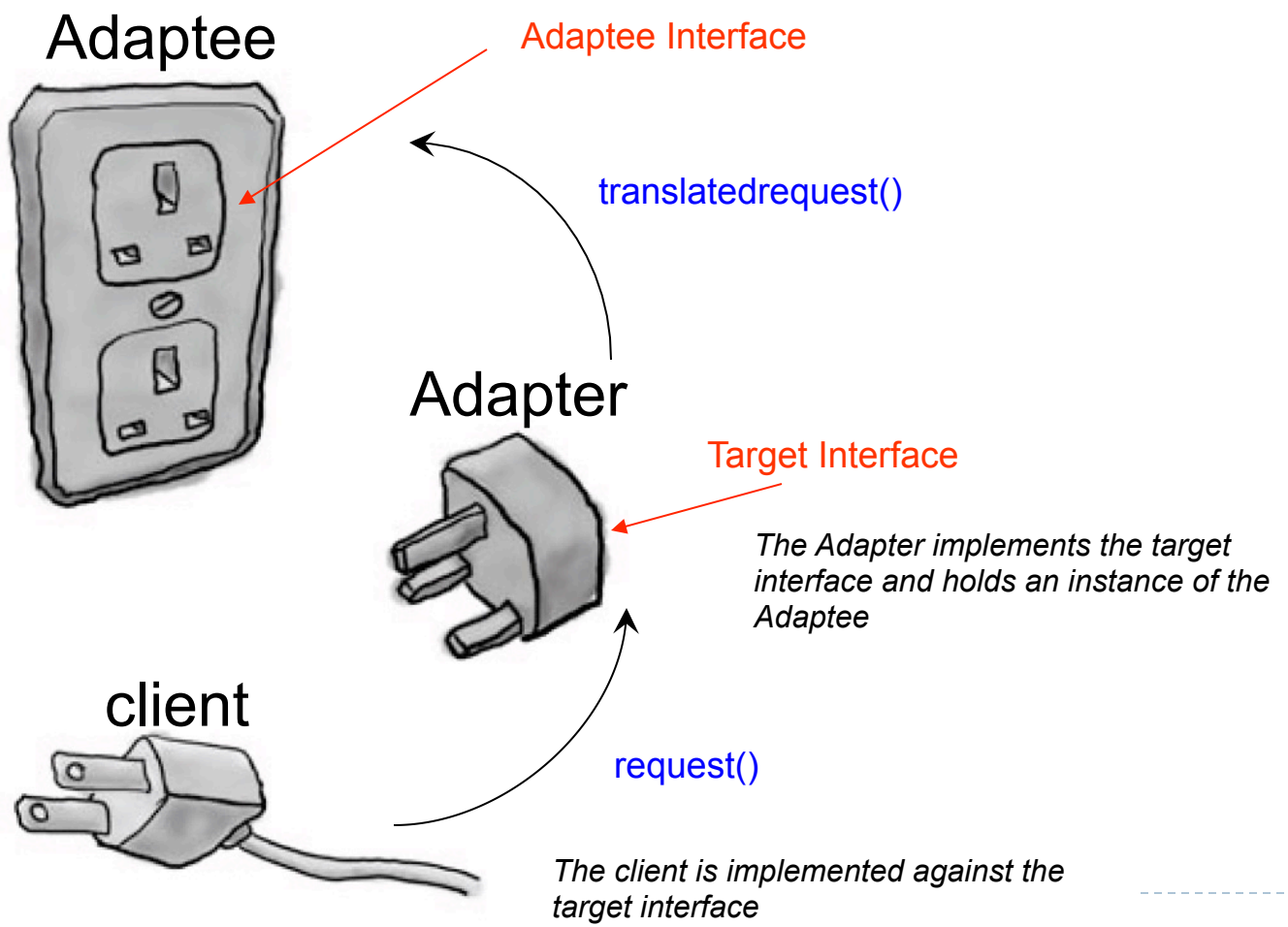
Adapter pattern

Acknowledgement: Head-first Design patterns. Freeman & Freeman

Example Scenario



Adapter Pattern Explained



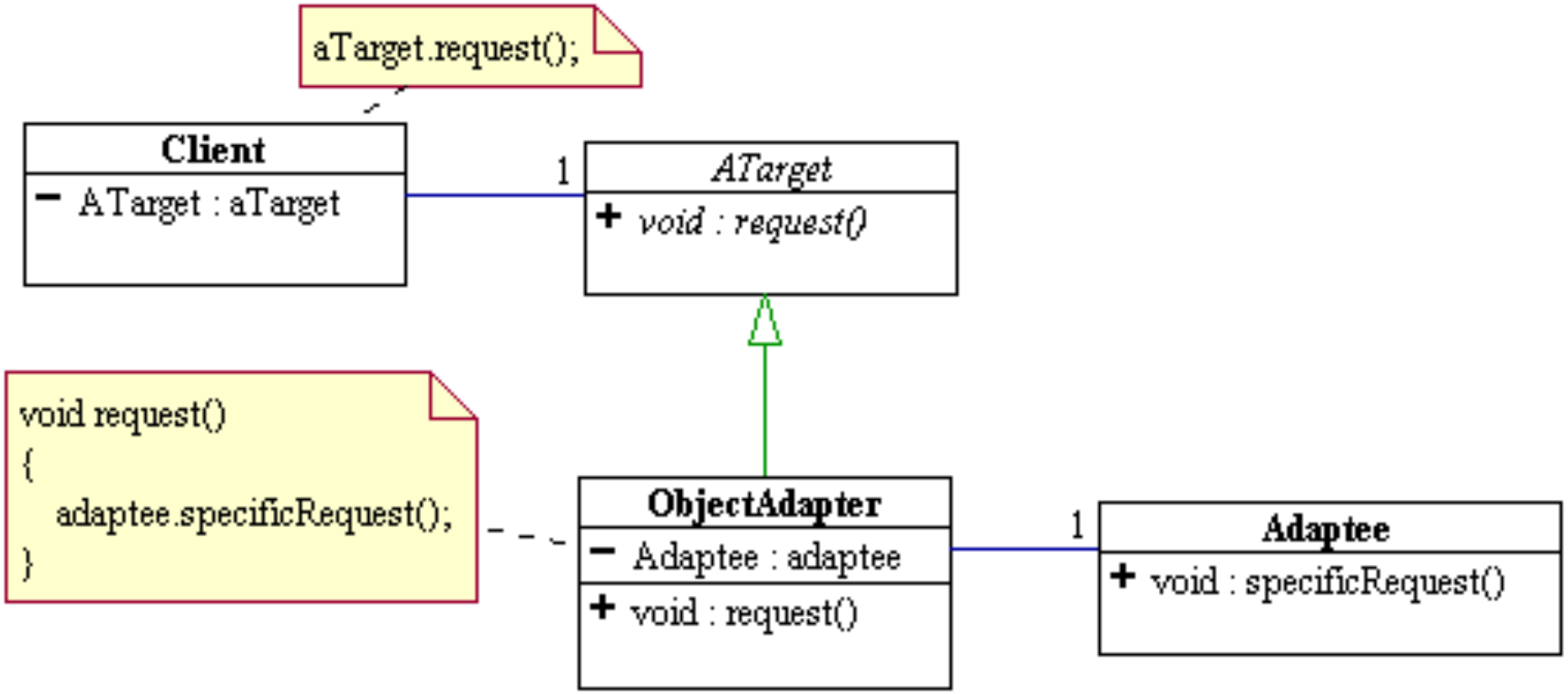
Adapter Pattern

- An adapter pattern converts the interface of a class into an interface that a client expects
- Adapters allow incompatible classes to work together
- Adapters can extend the functionality of the adapted class
- Commonly called “glue” or “wrapper”

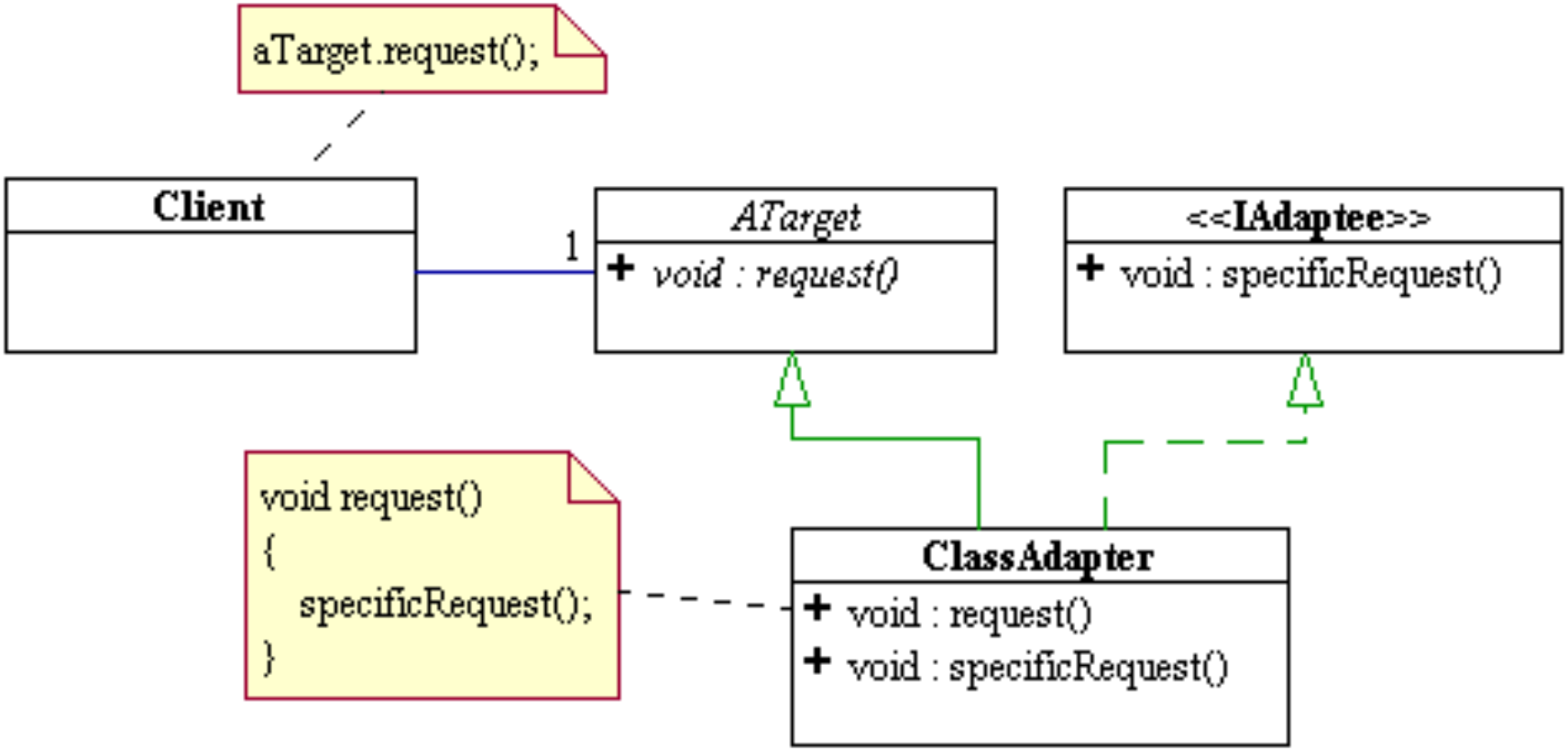
When to Use

- Need to adapt the interface of an existing class to satisfy client interface requirements
  - Adapting Legacy Software
  - Adapting 3rd Party Software

Object Adapter Pattern



Class Adapter Pattern



When to Use Adapters

- Concrete adapter
  When using a class whose interface does not match what you need.
- Abstract adapter
  When creating a reusable class that cooperates with unknown future classes.

Proxy pattern

Acknowledgement: Freeman & Freeman

The Problems

- Expensive & inexpensive pieces of state
  - Example: Large image
  - Inexpensive: size & location of drawing
  - Expensive: load & display
- Remote objects (e.g., another system)
  - Want to access it as if it were local
  - Want to hide all the required communications
  - Example: Java RMI
- Object with varying access rights
  - Some clients can access anything
  - Other clients have subset of functionality available

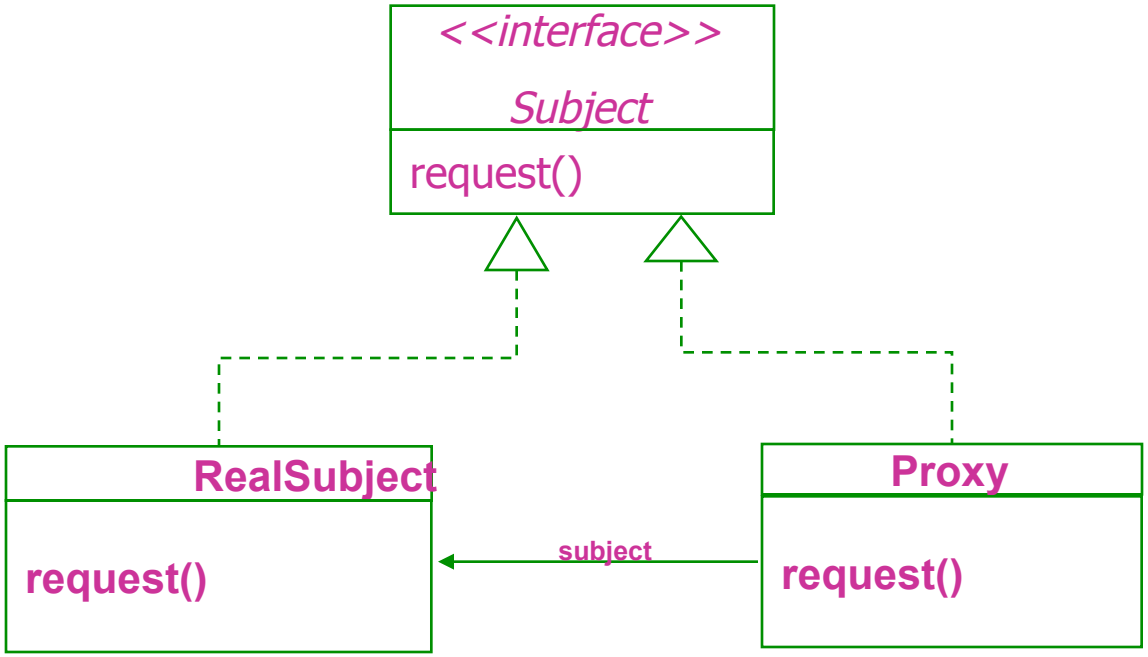
The Design Goal

- In all these cases desire access to object as if it is directly available
- For efficiency, simplicity, or security, put a *proxy* object in front of the real object
- We have a stand-in for the real object to control how the real object behaves

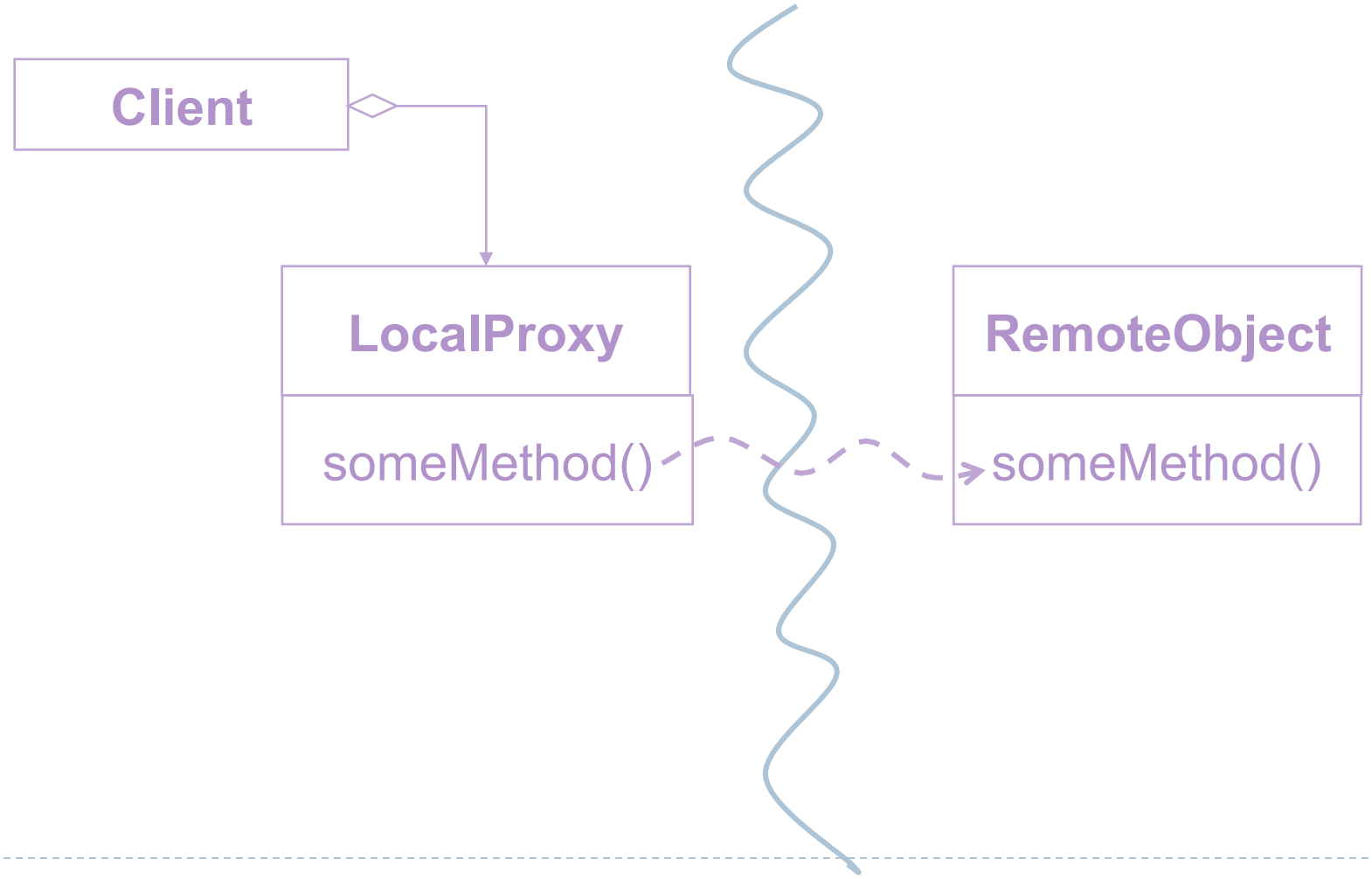
Proxy pattern Defined

- Proxy patterns provides a surrogate or placeholder for another object to control access to it
  - Remote proxy controls access to a remote object
  - Virtual proxy controls access to a resource that is expensive to create
  - Protection proxy controls access to a resource based on access rights

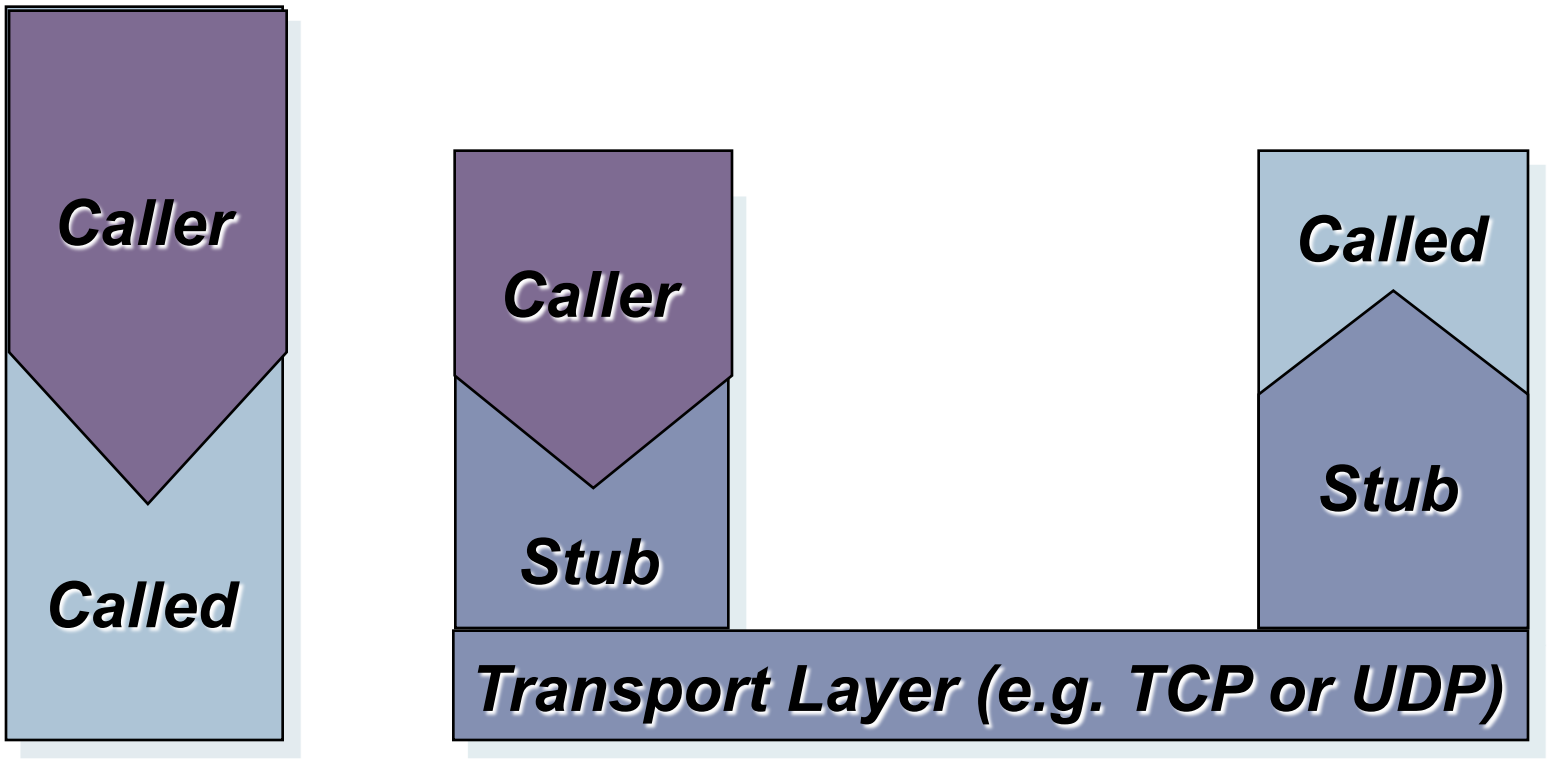
Proxy pattern structure



Example: Accessing Remote Object



Java RMI, the big picture



Categories of Proxies

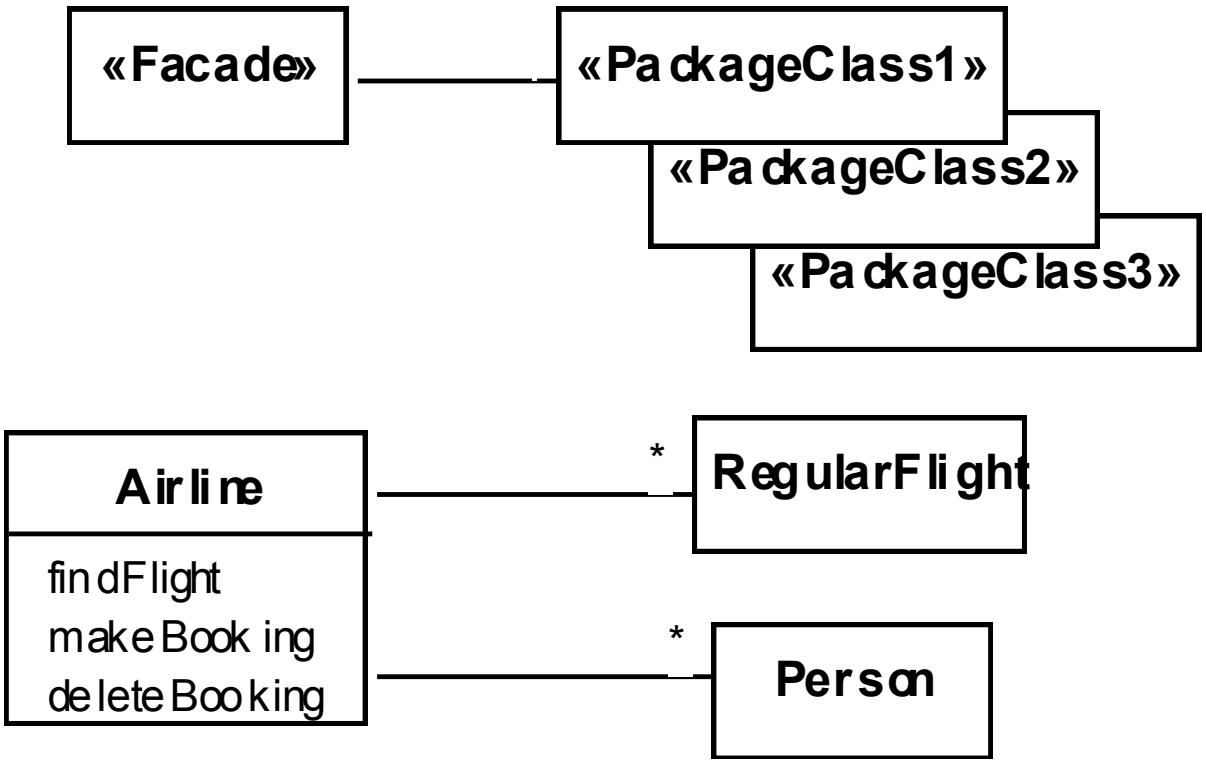
- **Remote proxy** - as above
  - Local representative for something in a different address space
  - Java RMI tools help set these up automatically
  - Object brokers handle remote objects (CORBA or DCOM)
- **Virtual proxy**
  - Stand-in for an object that is expensive to implement or completely access
  - Example – image over the net
  - May be able to access some state (e.g., geometry) at low cost
  - Defer other high costs until it must be incurred
- **Protection proxy**
  - Control access to the "real" object
  - Different proxies provide different rights to different clients
  - For simple tasks, can do via multiple interfaces available to clients
  - For more dynamic checking, need a front-end such as a proxy

The Controller Façade Pattern

- ***Context***:
  - Often, an application contains several complex packages.
  - A programmer working with such packages has to manipulate many different classes
- ***Problem***:
  - How do you simplify the view that programmers have of a complex package?
- ***Forces***:
  - It is hard for a programmer to understand and use an entire subsystem
  - If several different application classes call methods of the complex package, then any modifications made to the package will necessitate a complete review of all these classes.

Façade

- ***Solution:***



Proxies/Adapters/Facades

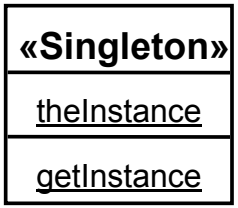
- Proxies and Adapters both place a stand-in object between the client and the real object
- Adapters do so to change the real object's interface
- Proxies do so to optimize access to the object via the same interface.
- Facades ease the use of sub-systems of objects

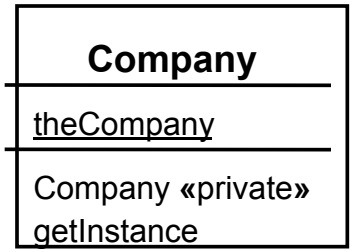
The Singleton Pattern

- ***Context***:
  - It is very common to find classes for which only one instance should exist (*singleton)*
- ***Problem***:
  - How do you ensure that it is never possible to create more than one instance of a singleton class?
- ***Forces***:
  - The use of a public constructor cannot guarantee that no more than one instance will be created.
  - The singleton instance must also be accessible to all classes that require it

Singleton

- ***Solution:***






```
if (theCompany==null)
  theCompany= new Company();

return theCompany;
```

Activity (Alternative designs)

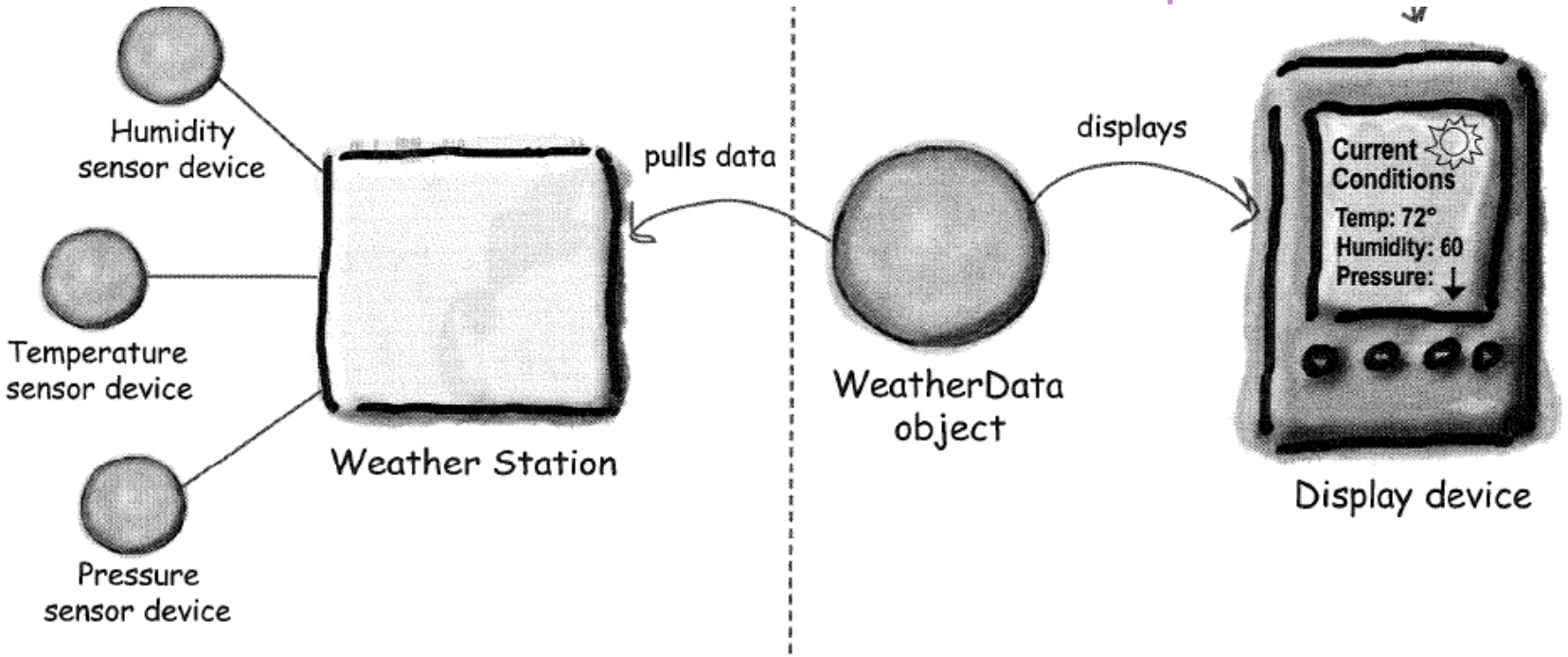
- You are developing software that provides a "sports ticker" service – users can bring up a score box for games they are interested in, and when the score for any game changes, the score box for all users who are interested in that game updates automatically.

Observer pattern

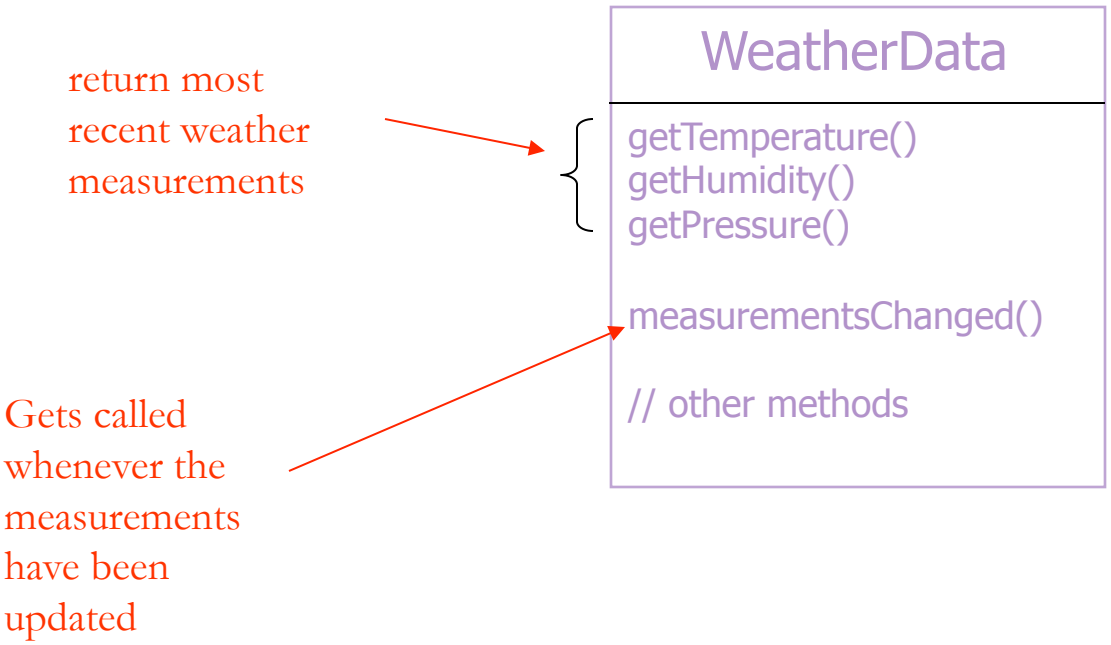
Acknowledgement: Freeman & Freeman

Weather Monitoring application

Display elements:
Current conditions,
Weather statistics,
Simple forecast



WeatherData class



WeatherData Implementation

```
public class WeatherData {
    // instance variable declarations

    public void measurementsChanged() {
            float temp = getTemperature();
            float humidity = getHumidity();
            float pressure = getPressure();

            currentConditionsDisplay.update (temp, humidity, pressure);
            statisticsDisplay.update (temp, humidity, pressure);
            forecastDisplay.update (temp, humidity, pressure);
    }

    // other WeatherData methods here
}
```

The Problem

- Given
  - Clusters of related classes
  - Tight connections within each cluster of classes
  - Loose state dependency between clusters
- Desired
  - Keep each cluster state consistent when state changes in cluster it depends on
  - Provide isolation such that changed cluster has no knowledge of specifics of dependent clusters

Example: UI & Application

- Application classes represent information being manipulated.
- UI provides way to view and alter application state.
- May have several views of state (charts, graphs, numeric tables).
- Views may be added at any time.
- How to tell views when application state has changed?

Approach One – Direct Connect

- Application knows about each View object.
- On state change, call appropriate method in the View object affected
- Issues
  - Application needs to know which method to call in each View
  - Application aware of changes to UI (e.g., add/delete/change Views).

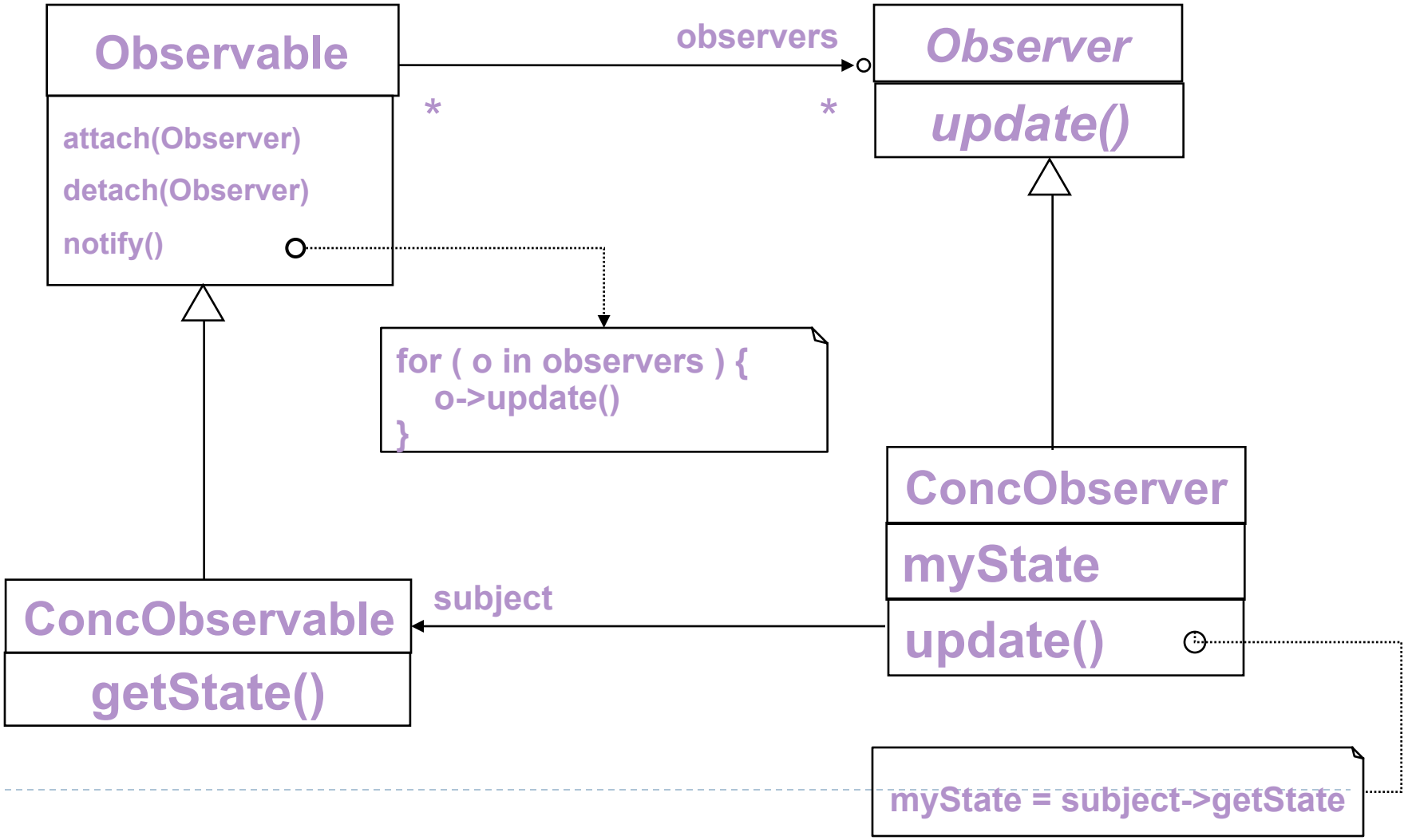
Approach Two : Observer Pattern

- The observer pattern defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically

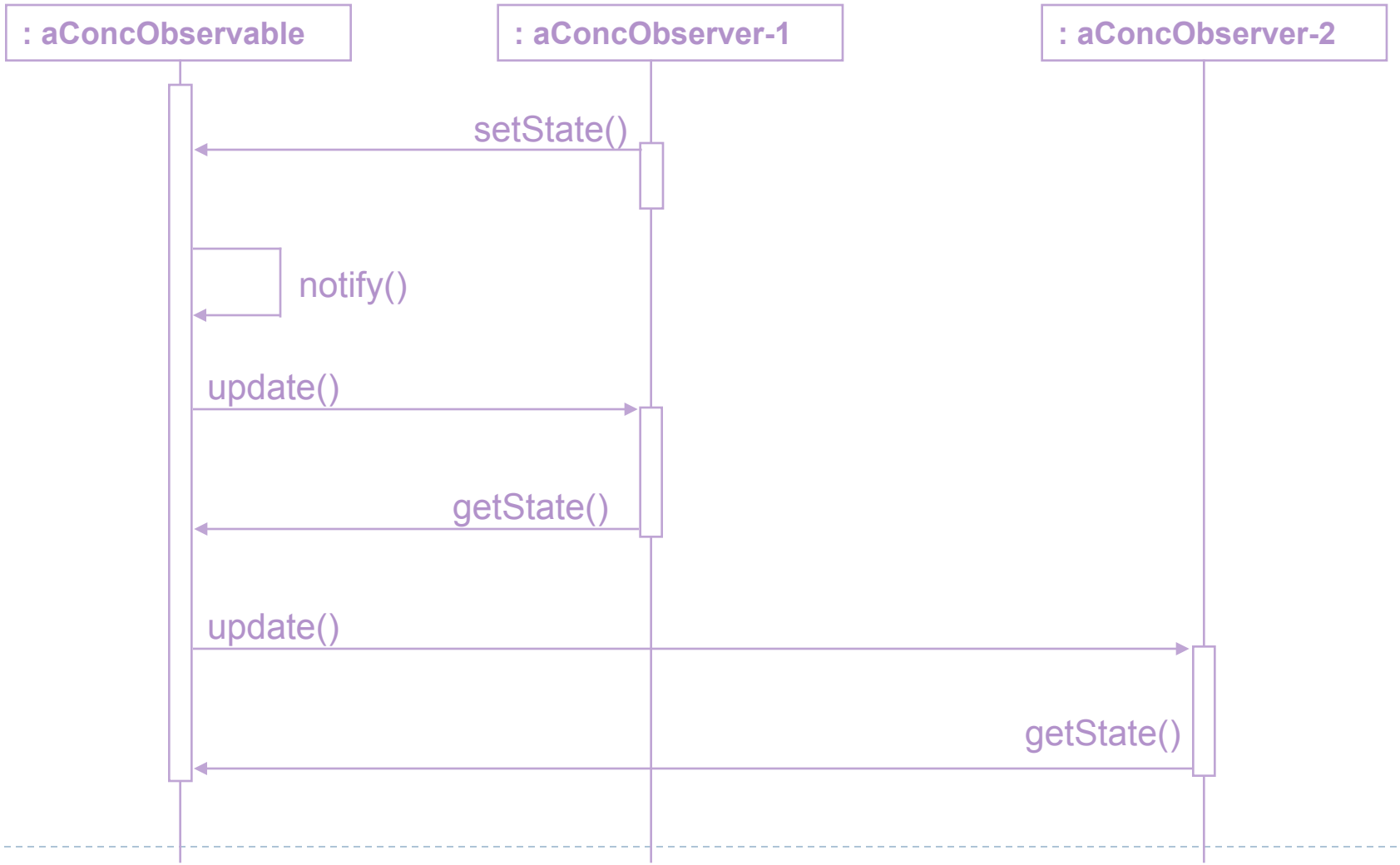
Approach Two – Observer/Subject
(AKA: publish/subscribe)

- Observables: objects with interesting state changes
- Observers: objects interested in state changes
- Observers *register* interest with observable objects.
- Observables *notify* all registered observers when state changes.

Observer Pattern - Class diagram



Interaction Diagram



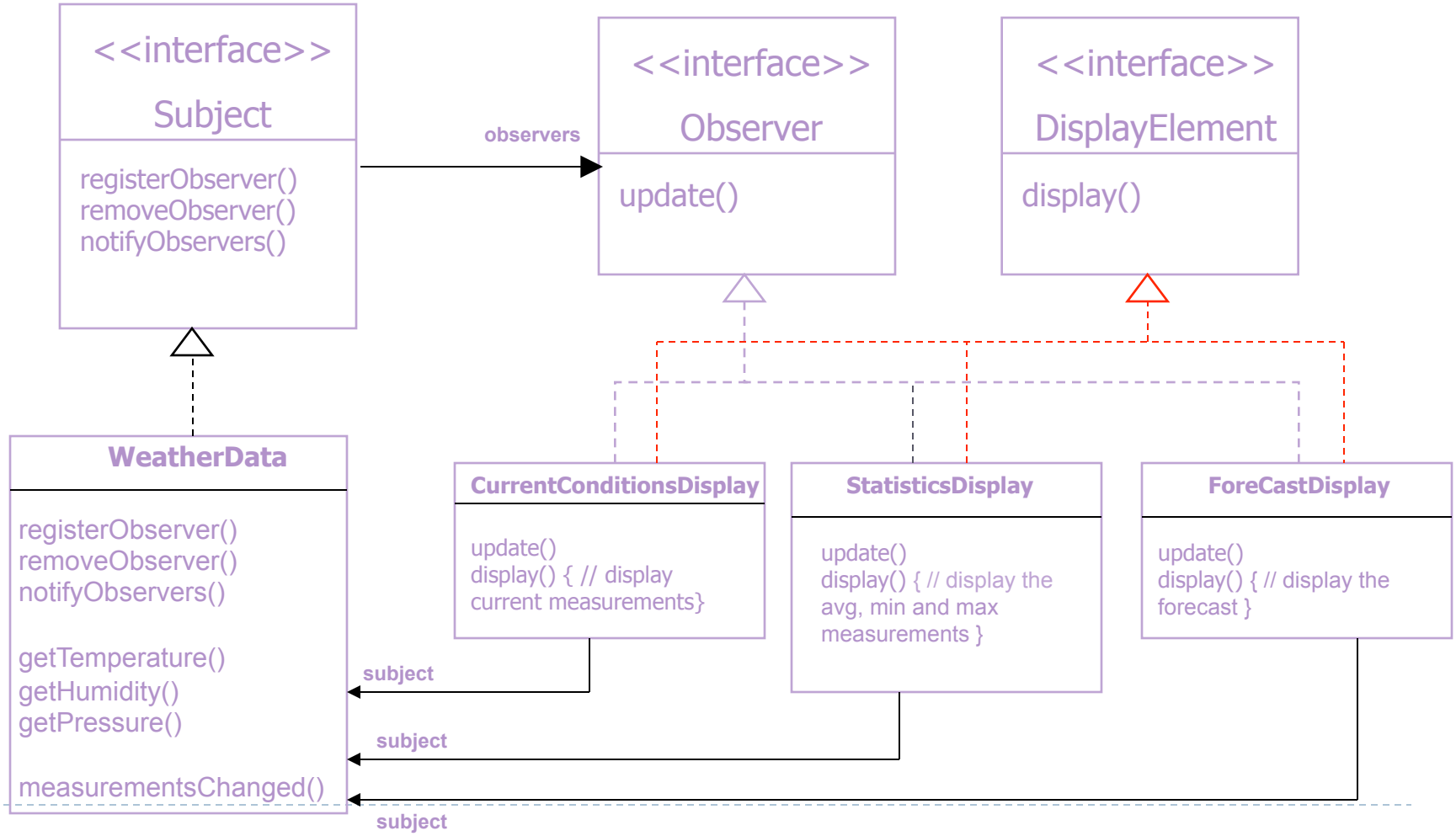
When to Use Observer

- Two subsystems evolve independently but must stay in synch.
- State change in an object requires changes in an unknown number of other objects (broadcast)
- Desire loose coupling between changeable object and those interested in the change.

Consequences

- Subject/observer coupling (loose coupling)
  - Subject only knows it has a list of observers
  - The only thing the Subject knows about an Observer is that it implements a certain interface
  - Observers can be added at any time
  - Does not know any Observer concrete class
  - Subjects don't need to be modified to add new types of Observers
  - Subjects and Observers can be reused independently
- Supports broadcast communication
  - Observables know little about notify receivers
  - Changing observers is trivial
- Unexpected & cascading updates
  - change/notify/update -> change/notify/update
  - May be hard to tell *what* changed

Designing the Weather Station



Observer Pattern – Key points

- Observer pattern defines a one-to-many relationship between objects
- Subjects/Observables update observers using a common interface
- Observers are loosely coupled in that the Observable knows nothing about them, other than that they implement observer interface
- You can PUSH or PULL data from the Observable when using the pattern (pull is considered more "correct")
- Don't depend on specific order of notification for your observers

Composite pattern

Acknowledgement: Freeman & Freeman

The Problem

- Problem
  - Have simple primitive component classes that collect into larger *composite* components
- Desire
  - Treat composites like primitives
  - Support composite sub-assemblies
  - Operations (usually) recurse to subassemblies
- Solution
  - Build composites from primitive elements

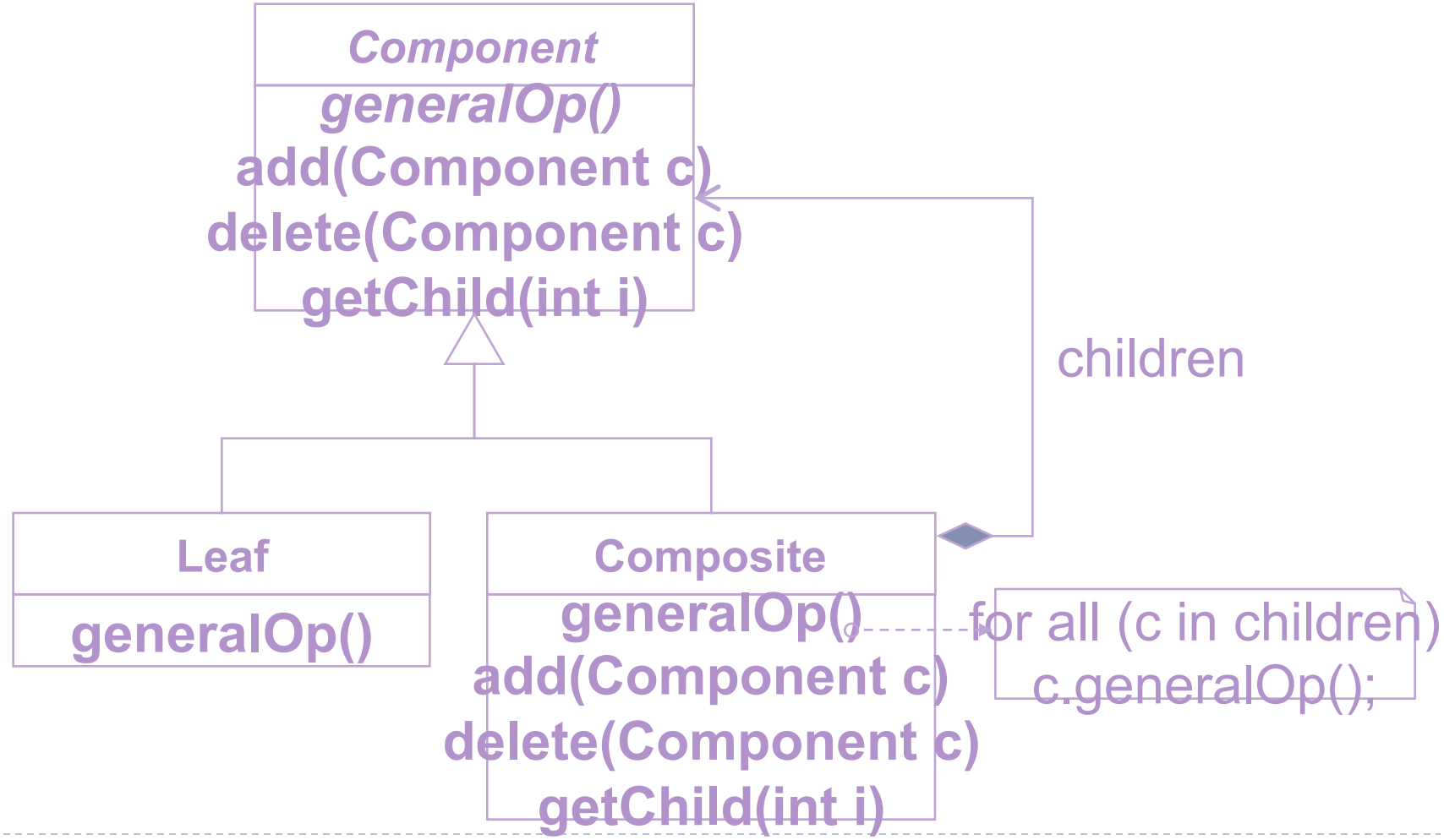
Examples - 1

- File systems
  - Primitives = text files, binary files, device files, etc.
  - Composites = directories (w/subdirectories)
- Make file dependencies
  - Primitives = leaf targets with no dependents
  - Composites = targets with dependents
- Menus
  - Primitives = menu entries
  - Composites = menus (w/submenus)

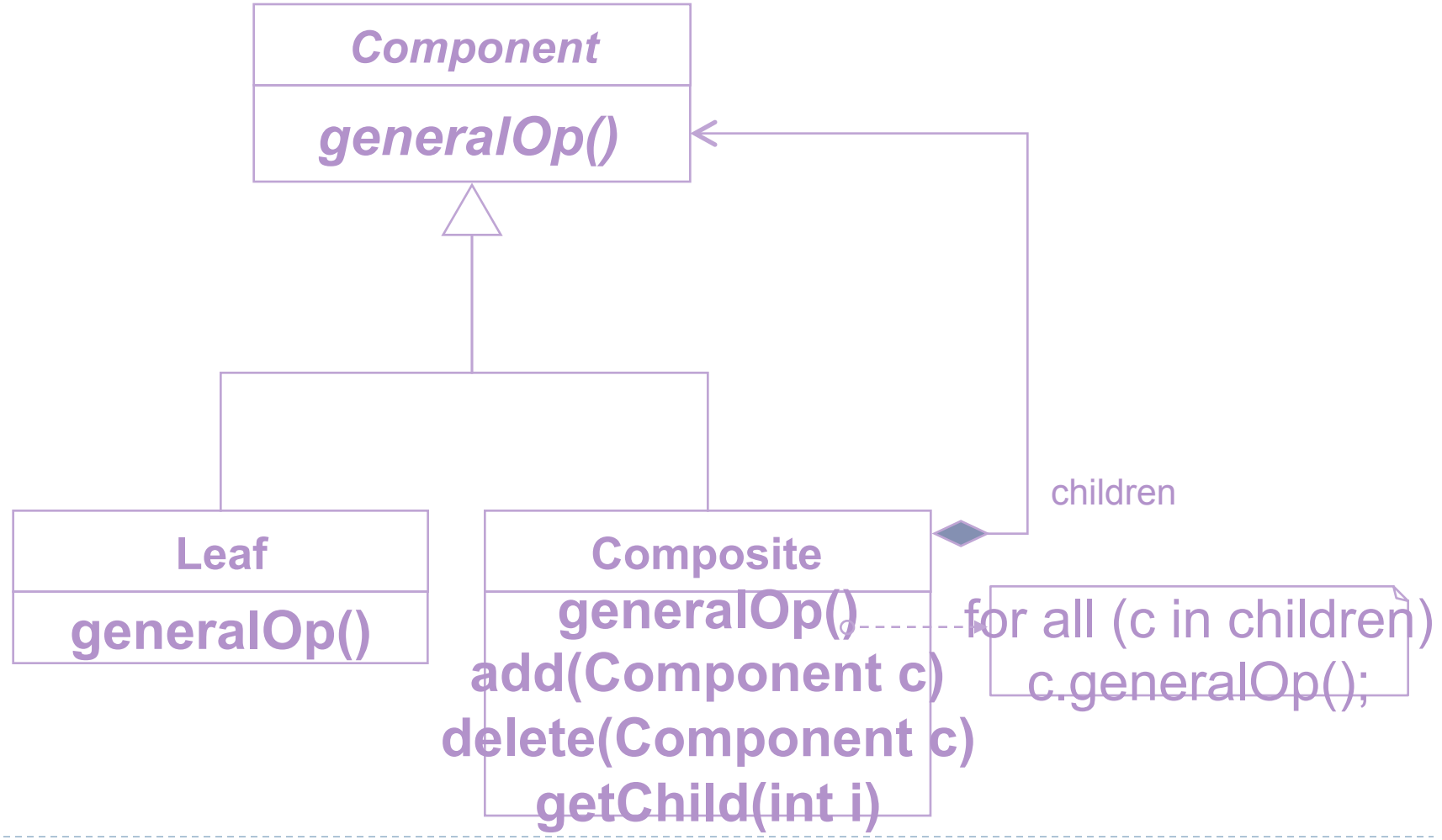
Examples - 2

- GUI Toolkits
  - Primitives = basic components (buttons, textareas, listboxes, etc).
  - Composites = frames, dialogs, panels.
- Drawing Applications
  - Primitives = lines, strings, polygons, etc.
  - Composites = groupings treated as unit.
- HTML/XML
  - Pages as composites of links (hypertext)
  - Pages as collections of paragraphs (with subparagraphs for lists, etc.)

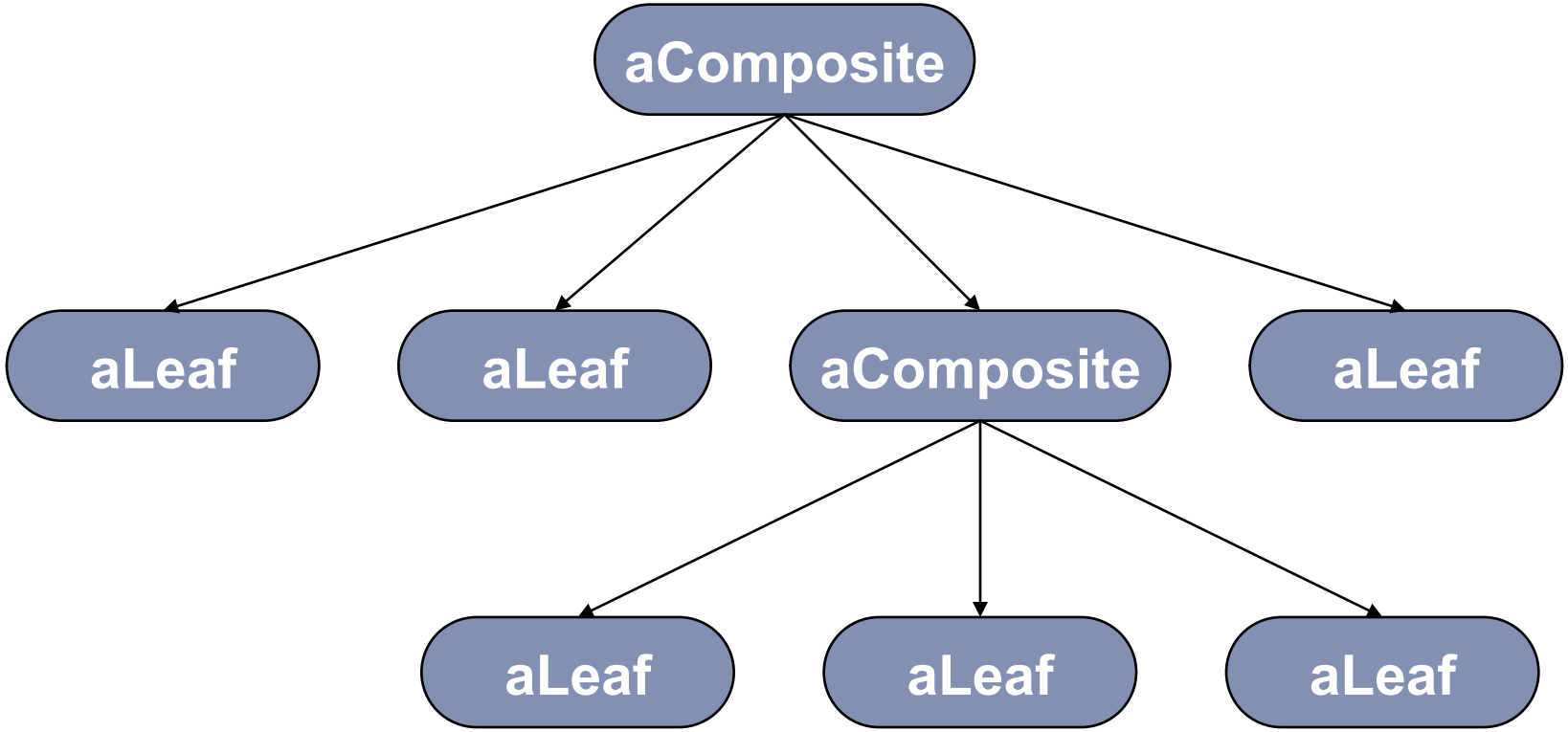
Class Diagram (Alternative 1)



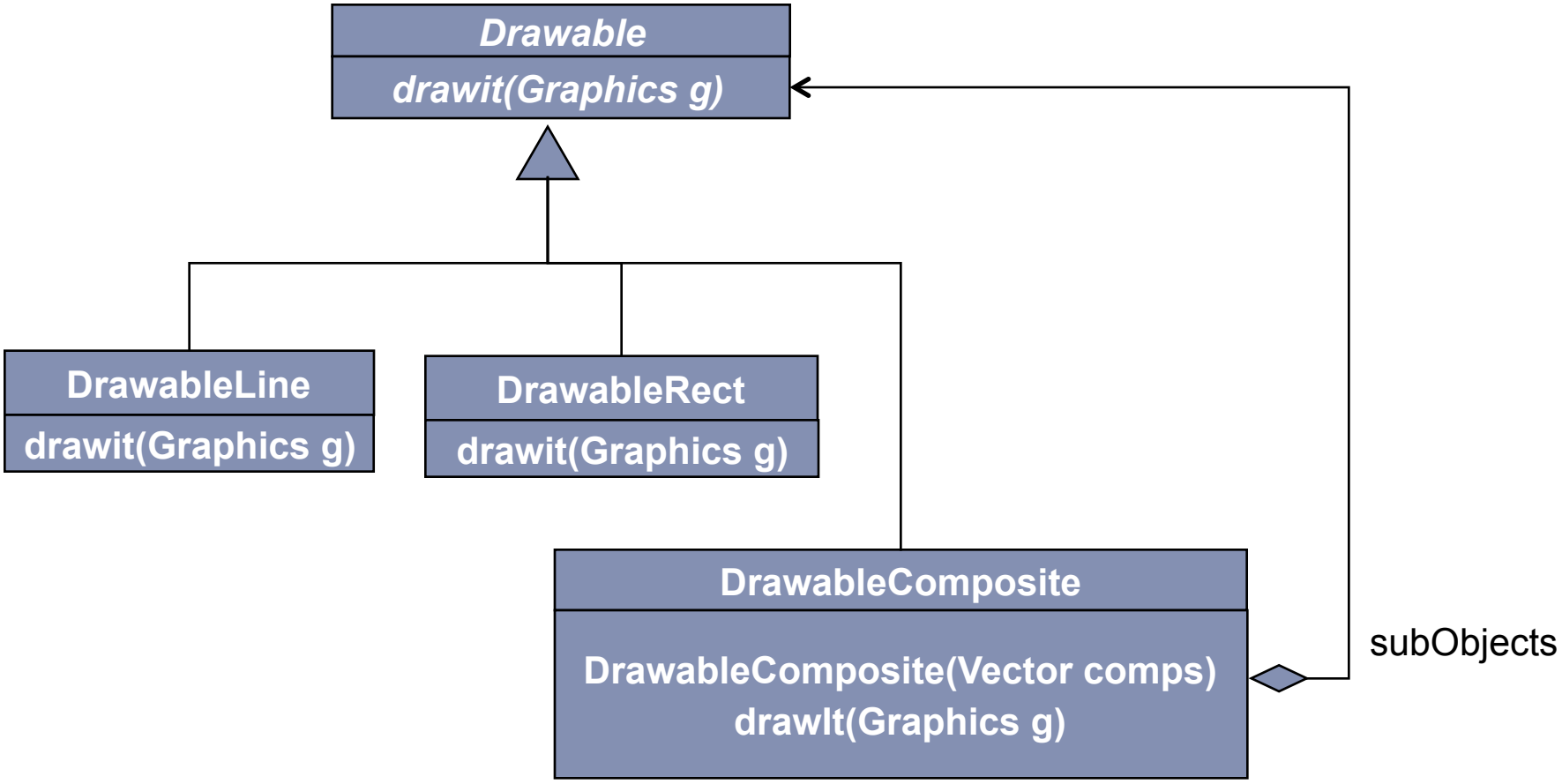
Class Diagram (Alternative 2)



Example Object Structure



Example: Drawable Figures



Discussion

- Clients (usually) ignore differences between primitives & composites
- Clients access (most) components via the generic interfaces
  - Primitives implement request directly
  - Composites can handle directly or forward
- Arbitrary composition to indefinite depth
  - Tree structure – no sharing of nodes
  - General digraph – supports sharing, multiple parents – be careful!
- Eases addition of new components
  - Almost automatic

Evaluating Designs

- The application of "well-known" design patterns that promote loosely coupled, highly cohesive designs.
- Conversely, identify the existence of recurring *negative* solutions – AntiPatterns
- AntiPattern : use of a pattern in an inappropriate context.
- Refactoring : changing, migrating an existing solution (antipattern) to another by improving the structure of the solution.