Pattern-Based Design

Prof. Alex Bardas

Design Patterns

- ***What are developers doing with software?***
  - Develop
  - Understand
  - Maintain – fix bugs
  - Update – add new features
- ***Important questions for a designer***
  - *Has anyone developed a solution for this?*
  - *Is there a standard way of describing a problem so I can look it up?*
  - *Is there an organized method for representing the solution to the problem?*

Design Patterns

- A **design pattern** is
  - An abstraction that prescribes a design solution to a specific, well-bounded design problem.
  - A three-part rule which expresses a relation between a problem and a solution in a certain context
- Why?
  - Most problems have multiple solutions
  - Context helps to define an environment
  - How can the problem be interpreted within the environment?
  - What solution is appropriate within the environment? An environment is influenced by a “system of forces” (limitations and constraints)

Design Patterns

- ***Why use a design pattern?***
  - Allows the software engineering community to capture design knowledge in a way that enables it to be reused
  - *Efficient*: avoiding lengthy process of trials and errors
  - *Predictable*: the solution is known to work for a given problem
  - *Readable*: use pattern terminology

Design Patterns

- Popularity increased after the following book was published:
  - **Design Patterns: Elements of Reusable Object-Oriented Software**
  - Authors: E. Gamma; R. Helm, R. Johnson, and J. Vlissides (a.k.a the Gang of Four)
  - Catalogs 23 different patterns as solutions to different classes of problems, in C++ & Smalltalk
  - Provide solutions for common problems in micro-design
  - Broadly applicable, used by many people over many years

Design Patterns Structure

- **Name**
  - Important to know for easier communication between designers
- **Problem**
  - Intent including description and context
  - When to apply the pattern
- **Solution**
  - Usually a class diagram segment
  - Describe details of objects/classes/structure if needed
  - UML, abstract code
- **Consequences**
  - Results and tradeoffs

Design Patterns

- Creational Patterns
  - Abstracting the object-instantiation process (e.g., Factory method)
- Structural Patterns
  - How objects/classes can be combined (e.g., Proxy pattern)
- Behavioral Patterns
  - Communication between objects (e.g., Command pattern)

Creational Patterns

- Focus on "*creation, composition, and representation of objects*"
  - Deal with the form of object creation: initializing and configuring
  - **Abstract factory**: factory for building related objects
  - **Builder**: factory for building complex objects incrementally
  - **Factory method**: method in a derived class creates associates
  - **Prototype**: factory for cloning new instances from a prototype
  - **Singleton**: factory for a singular instance

Factory Method

- **Problem: how to support look-and-feel settings?**
  - Assume user sets the appearance of scrollbars, menus, windows, etc.
  - Results in different look-and-feel standards

    e.g., Two classes *MotifScrollBar* and *WindowsScrollBar*, both are subclasses of *ScrollBar*
  - *How to create a new scrollbar?*

Factory Method

- **Problem: how to support look-and-feel settings?**
  - But we don't know if it's a Motif or Windows type

```
ScrollBar sc = new WindowsScrollBar();
```

    - Not good!

```
if (style==Windows){
    sc = new WindowsScrollBar();
} else{
    sc = new MotifScrollBar();
}
```

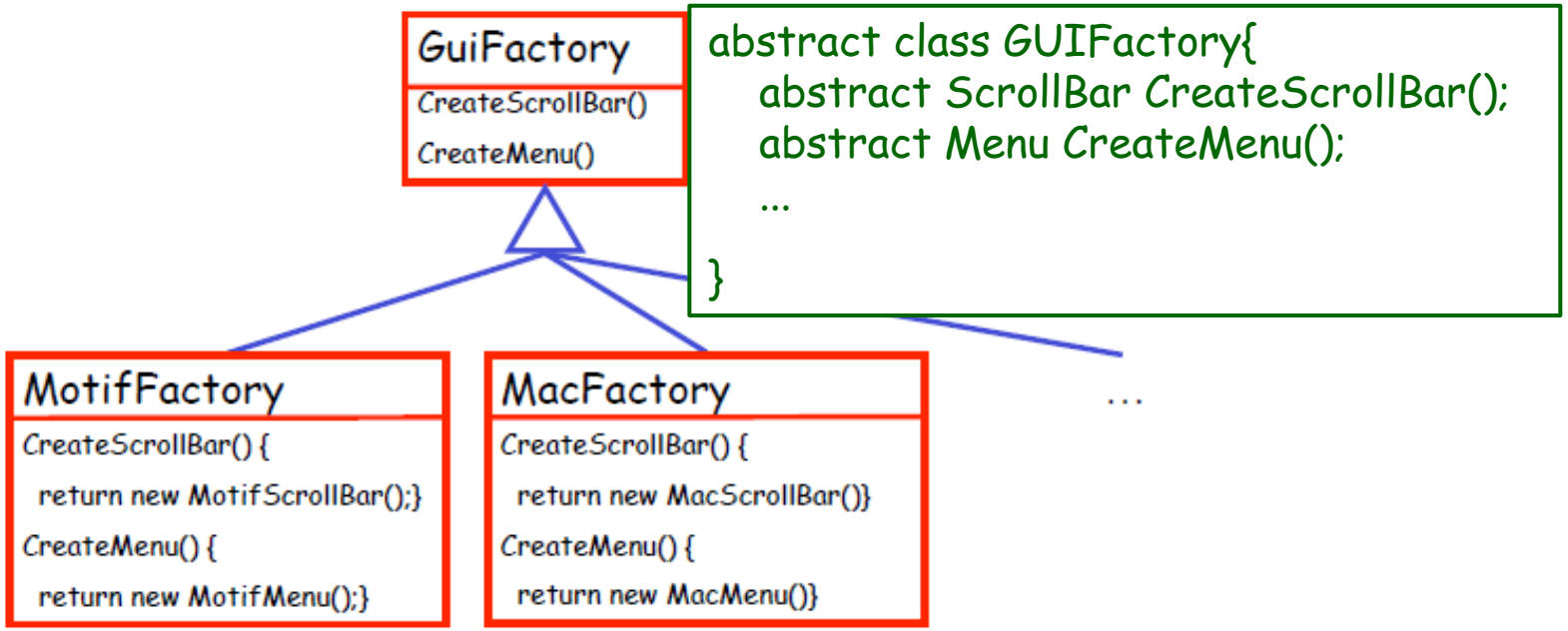
    - Still not good!
    - How to add new styles?

Factory Method

- Used when a method returns one of several possible classes that share a common super class
  - The class is chosen at run time – don't know ahead of time what class object to instantiate
  - Create a factory class
    - A superclass specifies all standard and generic behavior
    - Using virtual "placeholders" for creation steps
    - Delegate the creation details to subclasses that are supplied by the client

Factory Method

- **Problem: how to support look-and-feel settings?**
- **Solution: define a *GUIFactory* class**
  - Create objects without specifying the exact class of the object



Factory Method

- **Problem: how to support look-and-feel settings?**
- **Solution: define a *GUIFactory* class**
  - WindowsFactory implements the abstract GUIFactory class
  - Create a factory object with conditions set by the user

```
class WindowsFactory extends GUIFactory{
    ScrollBar createScrollBar(){
        return new WindowsScrollBar()
    }
    Menu createMenu(){
        return new WindowsMenu();
    }
    …
}
```

```
GUIFactory factory;
    if(style== WINDOW){
        factory = new WindowsFactory();
    } else

    if(style== MOTIF){
        factory = new MotifFactory();
    } else return null;
}
```

Factory Method

- Applies to the object creation of a family of classes
  - All potential classes are in the same subclass hierarchy
  - Can centralize class section code
  - Lift the conditional creation of objects to the creation of factories
  - The factory can be changed at runtime
- Pros and cons
  - Flexible for adding new types of objects
  - Hide subclasses from user

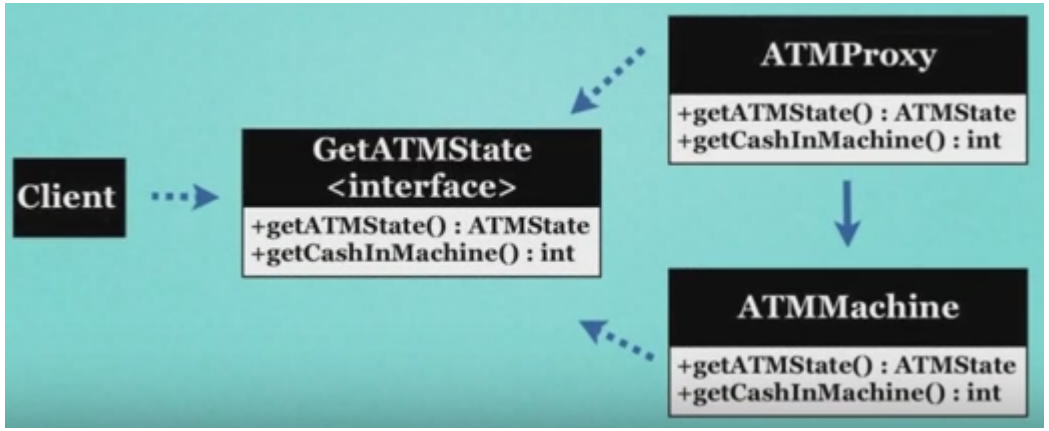
  - Not necessary if an instantiation of a class never changes
  - Sometimes it makes the code more difficult to understand

Structural Patterns

- Focus on "*how classes and objects are organized and integrated to build a larger structure*"
  - Deal with composition of classes and objects
  - Use inheritance to compose interfaces
  - Add flexibility inherent in object composition due to the ability to change composition at run-time

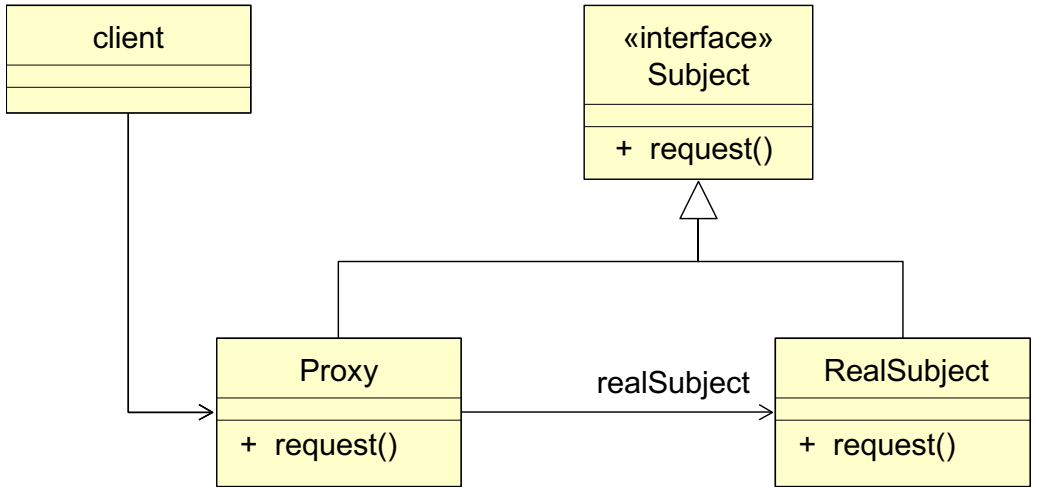
Proxy Pattern

- **Proxy** pattern acts as an interface to something else
  - Used to control access to an object
  - Functions as a placeholder for the server object – offers the same interface
  - Allows client objects to cross a “barrier” to the server object with limited access



Proxy Pattern

- **Proxy** pattern acts as an interface to something else
  - Proxy implements the same interface as the server object
  - Do not instantiate server objects unless and until it is actually requested by the client

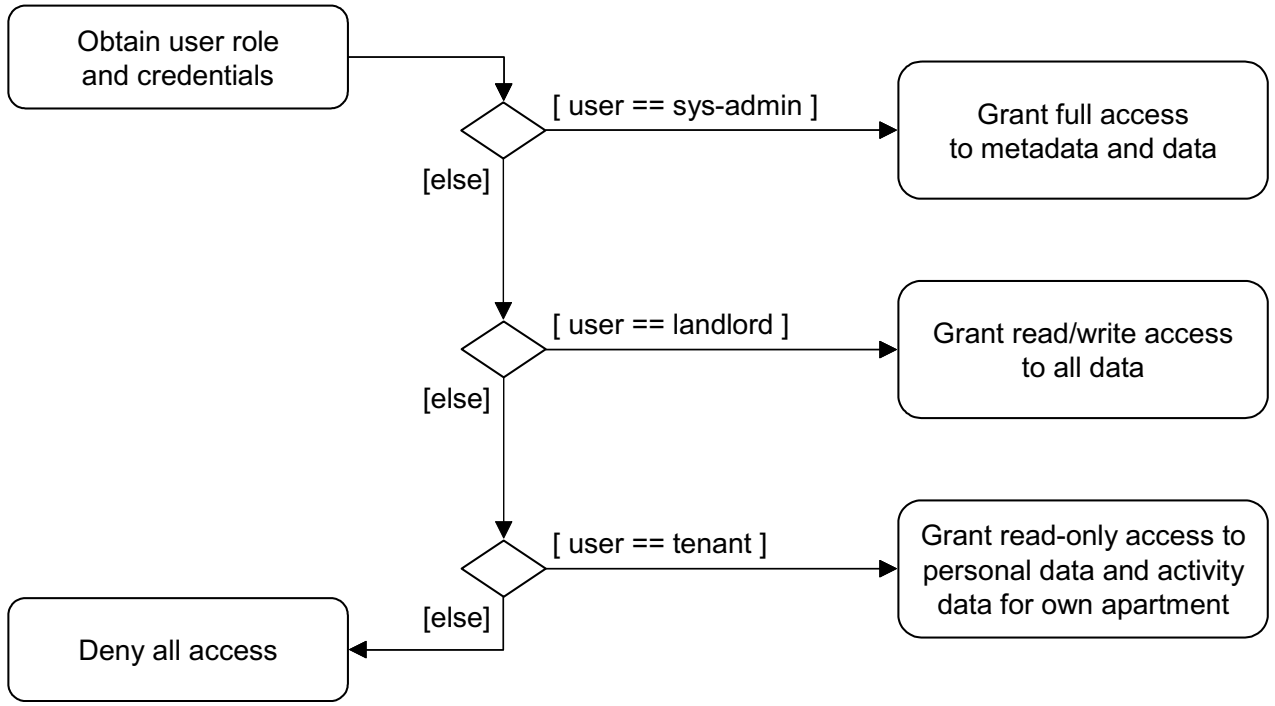


Proxy Pattern

- **Proxy** pattern acts as an interface to something else
  - Needed when the logistics of accessing the subject's service is overly complex – used as a helper object
  - **Protection Proxy**
    - Controls access to a sensitive master object
    - If different policies constrain the access to the subject
  - **Virtual Proxy**
    - A placeholder for "expensive to create" objects
    - If initiation of the subject is deferred to speed up the performance
  - **Remote Proxy**
    - If the subject is located in a remote address space
    - Provides a local representative for the remote object

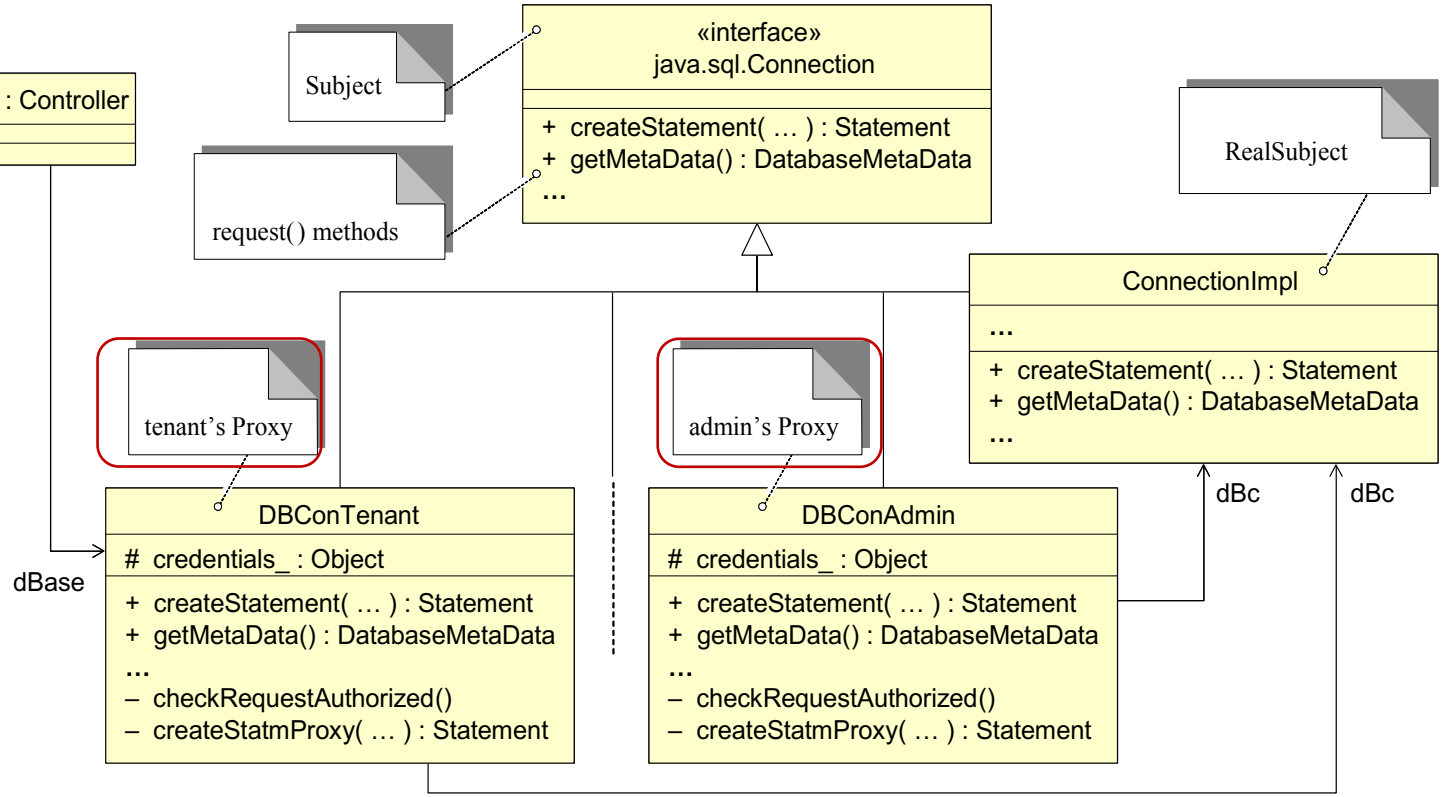
Protection Proxy

- Consider role-based access control



- Option 1: implement a “big/extensive” if-then-else statement at the client
- Not good! – leads to complex code that is difficult to extend

Protection Proxy



- Option 2: use a proxy for safe database access
  - Each specifies a set of authorized messages from client to subject
  - Unauthorized message will not pass through the proxy to the real subject (*ConnectionImpl*)

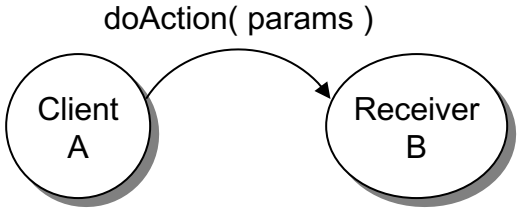
Behavior Patterns

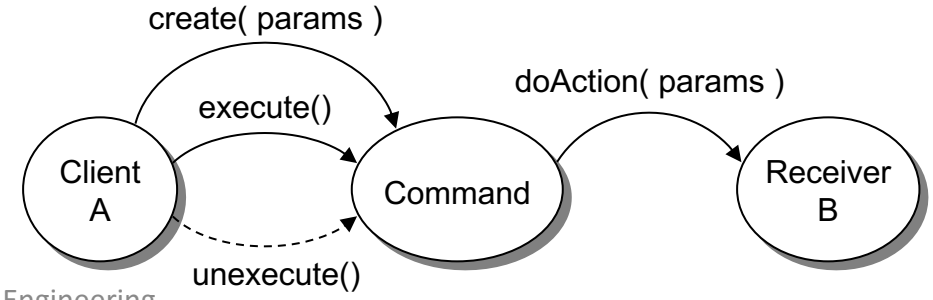
- Focus on "*assignment of responsibility between objects and the common communication patterns*"

- Separate functionality from the object to which the functionality applies

Command Pattern

- When objects invoke methods of other objects:
  - If the invoking object wants to reverse the effort of a previous invocation
  - If we want to track the course of the operations
- **A command pattern delegates the functionality from the client to the *Command* object**
  - e.g., rolling back B's state or logging operation history



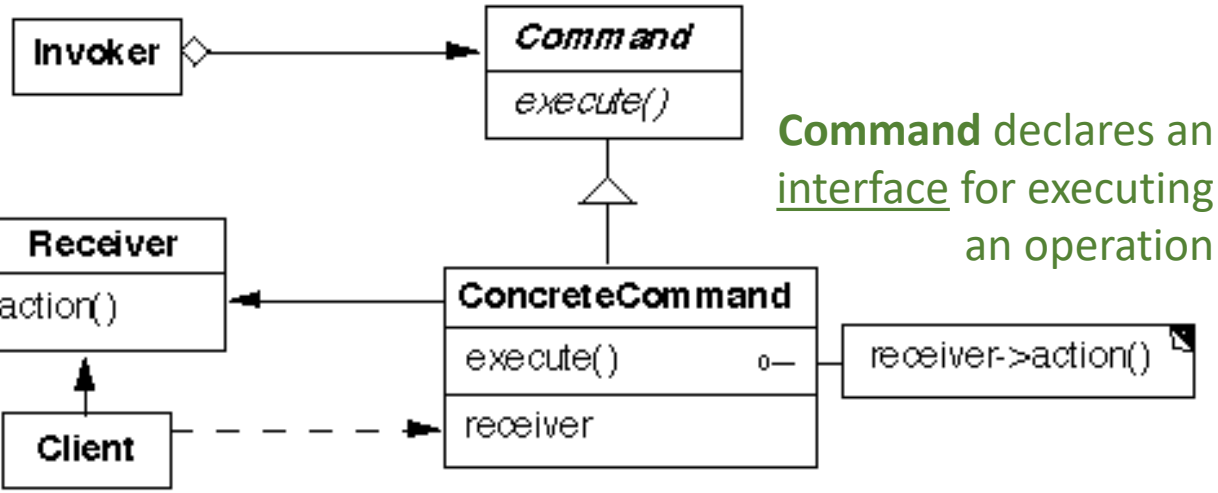


Command Pattern

- Command pattern encapsulates all the information needed to call a method into a “command” object

Structure:

Invoker asks the command to carry out the request



Command declares an interface for executing an operation

Receiver knows how to perform an operation

Client creates a ConcreteCommand object and sets its receiver

Example

- **Switch controls Light on/off**

```
public class Light {
        public Light() {
        }
        public void turnOn() {
                System.out.println("The light is on");
        }
        public void turnOff() {
                System.out.println("The light is off");
         }
}
```

Receiver

```
public interface Command {
        void execute();
}
```

Command interface

```
public class FlipUpCommand implements Command {
        private Light theLight;
        public FlipUpCommand(Light light) {
                this.theLight = light;
        }
        public void execute(){
                theLight.turnOn();
        }
 }
```

ConcreteCommand

```
public class Switch {
        private List<Command> history = new ArrayList<Command>();
        public Switch() {
        }
        public void storeAndExecute(Command cmd) {
                this.history.add(cmd);
                cmd.execute();
        }
 }
```

Invoker

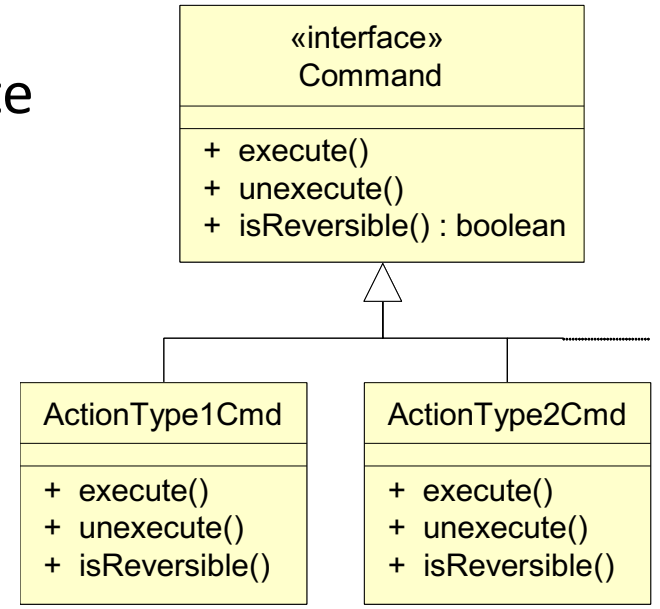
```
public class PressSwitch {
        public static void main(String[] args){
                Light lamp = new Light();
                Command switchUp = new FlipUpCommand(lamp);
                Switch mySwitch = new Switch();

                mySwitch.storeAndExecute(switchUp);
}
```

Client

- Think about how to flip down (turn-off the lights)
  - Need a separate *ConcreteCommand* invoked by the same invoker

Command Pattern

- Support *undo* (and *redo*)
  - Let each Command **store** what it needs to restore state
    - Store Commands in a stack or queue
  - Add more operations
    - *isReversible()*: allow the invoker to know if the command can be undone
    - *unexecute()*: undo the effect of a previous execute() operation
- Command pattern
  - Stores a set of commands in a class to use over and over
  - Easy to add new commands
  - Cons: create many small classes that store lists of commands

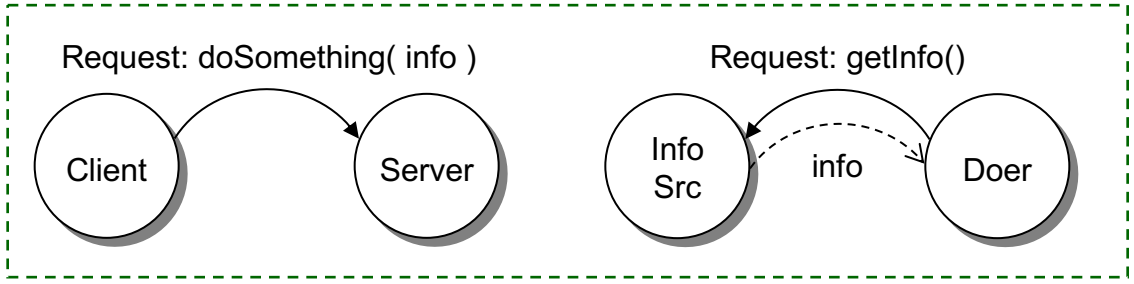


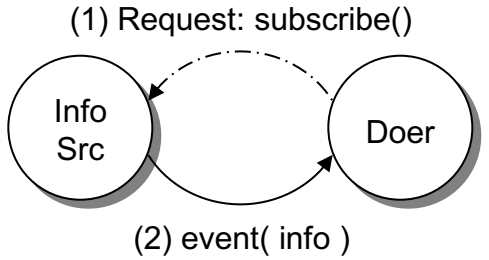
Other Useful Design Patterns

- Observer
- Façade
  - A façade is an object that provides a simplified interface to a larger body of code, such as a class library.
- Decorator
  - Allows behavior to be added to an individual object, either statically or dynamically, without affecting the behavior of other objects from the same class.
- Bridge
  - Decouples an abstraction from its implementation so that the two can vary independently

Example: Observer Pattern

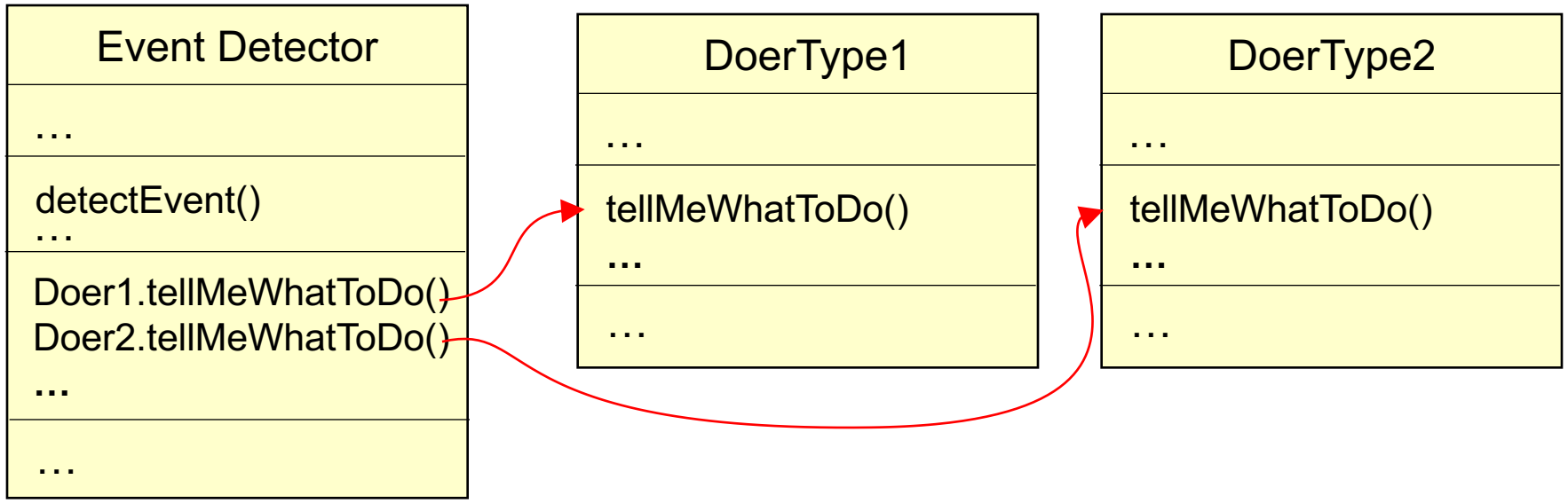
- A.k.a. ***Publish-Subscribe*** pattern
  - Defines one-to-many dependency between objects
    - The subject (i.e., publisher) maintains a list of dependents, observers (i.e., subscriber) and notifies them automatically of any state changes, generally by calling one of their methods
  - Indirect communication





Example: Observer Pattern

- Why use the ***Publish-Subscribe*** pattern?
  - When the subject doesn't know the identity of observers
  - Or, when the subject doesn't need/want to know the observers
    - The subject updates its state changes to observers and calls methods of the observers
- Disassociating unrelated responsibilities increases reusability



Responsibilities of Event Detector:

Doing:

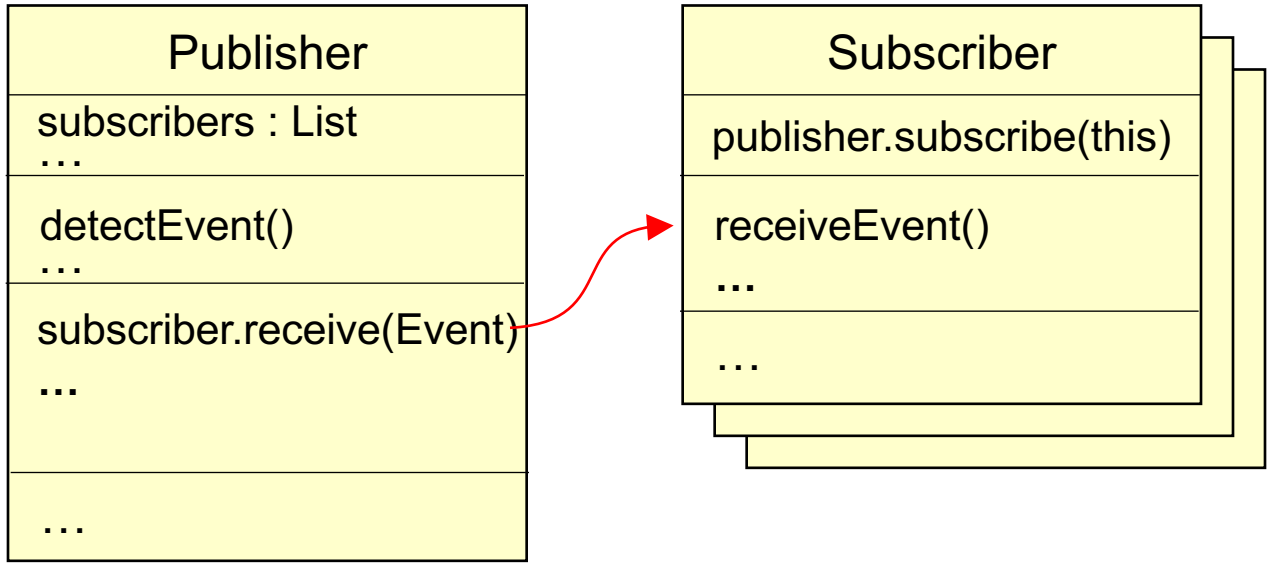
- Detect events

Calling:

- Tell Doer-1 what to do
- Tell Doer-2 what to do

unrelated! ⇒ change the Event Detector

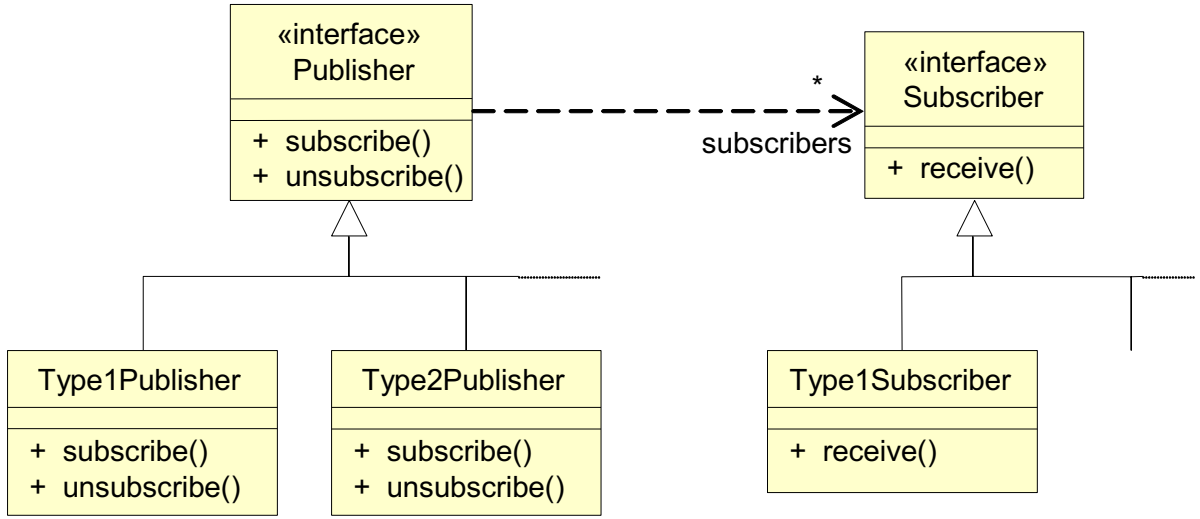
- When event detection needs to change
- When new doer types need to be told what to do



Implement the Event Detector as Publisher

Dissociate unrelated responsibilities:

- When event detection needs to change → change Publisher
- When new doer types need to be added → add an new Subscriber type



Other Categorization of Patterns

- **Architectural patterns** describe broad-based design problems that are solved using a structural approach
- **Data patterns** describe recurring data-oriented problems and the data modeling solutions
- **Component patterns** (a.k.a. design patterns) address problems associated with the development of subsystems and components
- **Interface design patterns** describe common user interface problems and their solution with a system of forces
- **WebApp patterns** address a problem set that is encountered when building WebApps.

Frameworks

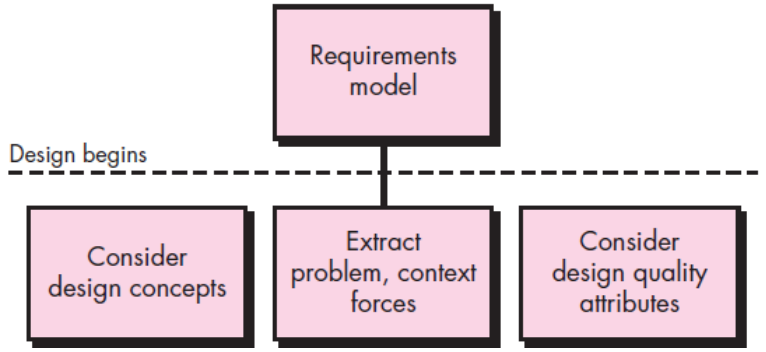
- In some cases, patterns may not be enough
- **A framework is an implementation-specific skeletal infrastructure**
- A framework contains a collection of:
  - *Hooks*: some functionality is *optional*, user may add it if needed
  - *Slots*: some components (classes/methods) are intentionally incomplete, but must be implemented by the developer

Design Patterns vs. Frameworks

- Design patterns are more abstract than frameworks
- Design patterns are smaller architectural elements than frameworks
- Design patterns are less specialized than frameworks

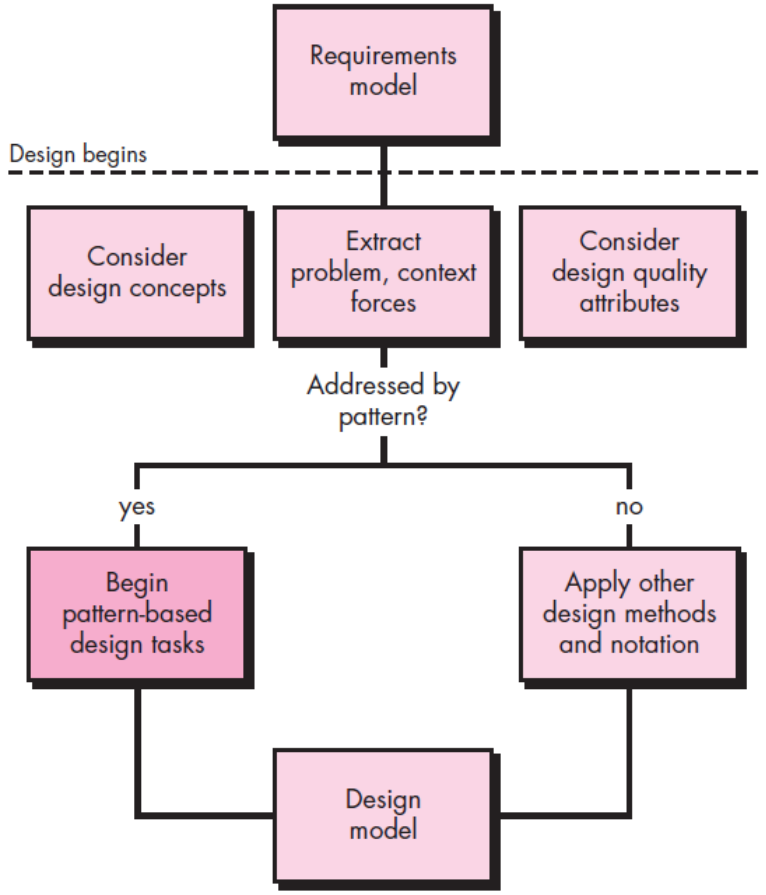
Pattern-Based Design

- Begins with a requirements model (either explicit or implied)
  - Presents an abstract representation of the system
  - Describes the *problem set*, establishes the *context*, and identifies the *system of forces*.



Pattern-Based Design

- Begins with a requirements model (either explicit or implied)
  - Presents an abstract representation of the system
  - Describes the *problem set*, establishes the *context*, and identifies the *system of forces*.
- Use methods and modeling tools only when you're faced with a problem, context, and system of forces that have not been solved before.



Thinking in Patterns

- Shalloway and Trott* suggested to ***think in patterns***:
  1. Be sure you understand the big picture – the ***context*** in which the software to be built resides.
  2. Examining the big picture – extract the patterns that are present at that ***level of abstraction***.
  3. Begin your design with "big picture" patterns that establish a context or ***skeleton*** for further design work.
  4. "Work inward from the context" – looking for ***patterns at lower levels of abstraction*** that contribute to the design solution.
  5. Repeat steps 1 to 4 until the complete design is fleshed out.
  6. ***Refine*** the design by adapting each pattern to the specifics of the software you're trying to build.

* Shalloway, A., and J. Trott, Design Patterns Explained , 2nd ed., Addison-Wesley, 2005.

When Thinking in Design Patterns …

Follow the design tasks for pattern-based design:

1. Examine the ***requirements model*** and develop a problem hierarchy.
2. Determine if a reliable ***pattern language*** has been developed for the problem domain.
3. Beginning with a broad problem, determine whether one or more ***architectural patterns*** are available for it.
4. Using the collaborations provided for the architectural pattern, examine subsystem- or component-level problems and search for appropriate ***patterns*** to address them.
5. Repeat steps 2 through 5 until all broad problems have been addressed.

When Thinking in Design Patterns …

6. If user interface design problems have been isolated (this is almost always the case), search the many ***user interface design pattern*** repositories for appropriate patterns.
7. Regardless of its level of abstraction, if a pattern language and/or patterns repository or individual pattern shows promise, ***compare*** the problem to be solved against the existing pattern(s) presented.
8. Be certain to ***refine*** the design as it is derived from patterns using design quality criteria as a guide.

Pattern-Organizing Table

- Microsoft suggests using a *pattern-organizing table* to organize your evaluation of candidate patterns:

	Database	Application	Implementation	Infrastructure
Data/Content				
Problem statement ...	PatternName(s)		PatternName(s)	
Problem statement ...		PatternName(s)		PatternName(s)
Problem statement ...	PatternName(s)			PatternName(s)
Architecture				
Problem statement ...		PatternName(s)		
Problem statement ...		PatternName(s)		PatternName(s)
Problem statement ...				
Component-level				
Problem statement ...		PatternName(s)	PatternName(s)	
Problem statement ...				PatternName(s)
Problem statement ...		PatternName(s)	PatternName(s)	
User interface				
Problem statement ...		PatternName(s)	PatternName(s)	
Problem statement ...		PatternName(s)	PatternName(s)	
Problem statement ...		PatternName(s)	PatternName(s)	

Common Design Mistakes

- Not enough time has been spent to understand the underlying problem, its context and forces, and as a consequence
  - Select a pattern that looks right, but is inappropriate for the solution required.
- A wrong pattern is selected
  - Refuse to see error and force fit the pattern.
  - Forces not considered by the chosen pattern result in a poor or erroneous fit.
- Sometimes a pattern is applied too literally and the required adaptations for your problem space are not implemented

Patterns Repositories

- There are many sources for design patterns available
- Some patterns can be obtained from individually published ***pattern languages***, while others are available as part of a ***patterns portal*** or ***patterns repository***.
  - Pattern Index - http://c2.com/cgi/wiki?PatternIndex
  - Portland Pattern Repository - http://c2.com/ppr/index.html

References

- Prof. Fengjun Li's EECS 448 Fall 2015 slides

- This slide set has been extracted and updated from the slides designed to accompany *Software Engineering: A Practitioner's Approach, 8/e* (McGraw-Hill 2014) by Roger Pressman