Design patterns

Based on slides by Glenn D. Blank

Definitions

- A *pattern* is a recurring solution to a standard problem, in a context.
- Christopher Alexander, a professor of architecture…
  - *Why would what a prof of architecture says be relevant to software?*
  - "A pattern describes a problem which 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 this solution a million times over, without ever doing it the same way twice."
- Jim Coplein, a software engineer:
  "I like to relate this definition to dress patterns…"
  - *What are dress patterns?*
  - "... I could tell you how to make a dress by specifying the route of a scissors through a piece of cloth in terms of angles and lengths of cut. Or, I could give you a pattern. Reading the specification, you would have no idea what was being built or if you had built the right thing when you were finished. The pattern foreshadows the product: it is the rule for making the thing, but it is also, in many respects,
    the thing itself."

Patterns in engineering

- *How do other engineers find and use patterns?*
  - Mature engineering disciplines have handbooks describing successful solutions to known problems
  - Automobile designers don't design cars from scratch using the laws of physics
  - Instead, they reuse standard designs with successful track records, learning from experience
  - *Should software engineers make use of patterns? Why?*
  - "Be sure that you make everything according to the pattern I have shown you here on the mountain." Exodus 25:40.
- Developing software from scratch is also expensive
  - Patterns support reuse of software architecture and design

The "gang of four" (GoF)

- Erich Gamma, Richard Helm, Ralph Johnson & John Vlissides (Addison-Wesley, 1995)
  - *Design Patterns* book catalogs 23 different patterns as solutions to different classes of problems, in C++ & Smalltalk
  - The problems and solutions are broadly applicable, used by many people over many years
  - What design pattern did we discover with the Undo problem?
    - Why is it useful to learn about this pattern?
    - Patterns suggest opportunities for reuse in analysis, design and programming
  - GOF presents each pattern in a structured format
    - *What do you think of this format? Pros and cons?*

Elements of Design Patterns

- Design patterns have 4 essential elements:
  - Pattern name: increases vocabulary of designers
  - Problem: intent, context, when to apply
  - Solution: UML-like structure, abstract code
  - Consequences: results and tradeoffs

Model View Controller (MVC)

MVC slides by Rick Mercer with a wide variety of others

Model/View/Controller

- The intent of MVC is to keep neatly separate objects into one of tree categories
  - Model
    - The data, the business logic, rules, strategies, and so on
  - View
    - Displays the model and usually has components that allows user to edit change the model
  - Controller
    - Allows data to flow between the view and the model
    - The controller mediates between the view and model

Sun says

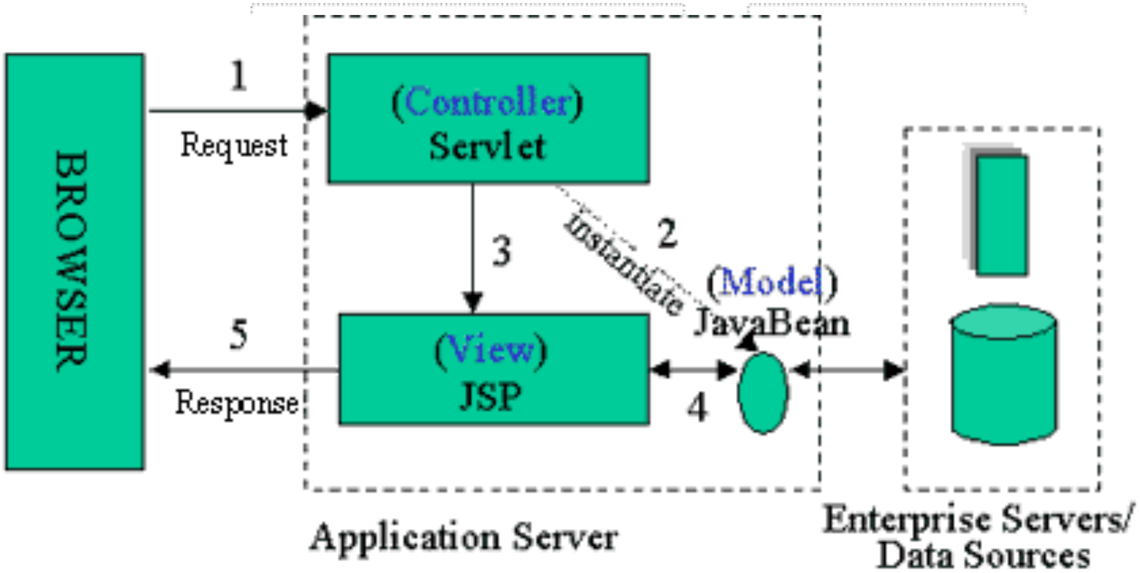
- Model-View-Controller ("MVC") is the recommended architectural design pattern for interactive applications
- MVC organizes an interactive application into three separate modules:
  - one for the application model with its data representation and business logic,
  - the second for views that provide data presentation and user input, and
  - the third for a controller to dispatch requests and control flow.

Sun continued

- Most Web-tier application frameworks use some variation of the MVC design pattern
- The MVC (architectual) design pattern provides a host of design benefits

Java Server Pages

- Model 2 Architecture to serve dynamic content
  - Model: Enterprise Beans with data in the DBMS
    - JavaBean: a class that encapsulates objects and can be displayed graphically
  - Controller: Servlets create beans, decide which JSP to return, do the bulk of the processing
  - View: The JSPs generated in the presentation layer (the browser)



OO-tips Says

- The MVC paradigm is a way of breaking an application, or even just a piece of an application's interface, into three parts: the model, the view, and the controller.
- MVC was originally developed to map the traditional input, processing, output roles into the GUI realm:
  - Input --> Processing --> Output
  - Controller --> Model --> View

MVC Benefits

- Clarity of design
  - easier to implement and maintain
- Modularity
  - changes to one don't affect the others
  - can develop in parallel once you have the interfaces
- Multiple views
  - games, spreadsheets, powerpoint, Eclipse, UML reverse engineering, ….

Model

- The Model's responsibilities
  - Provide access to the state of the system
  - Provide access to the system's functionality
  - Can notify the view(s) that its state has changed

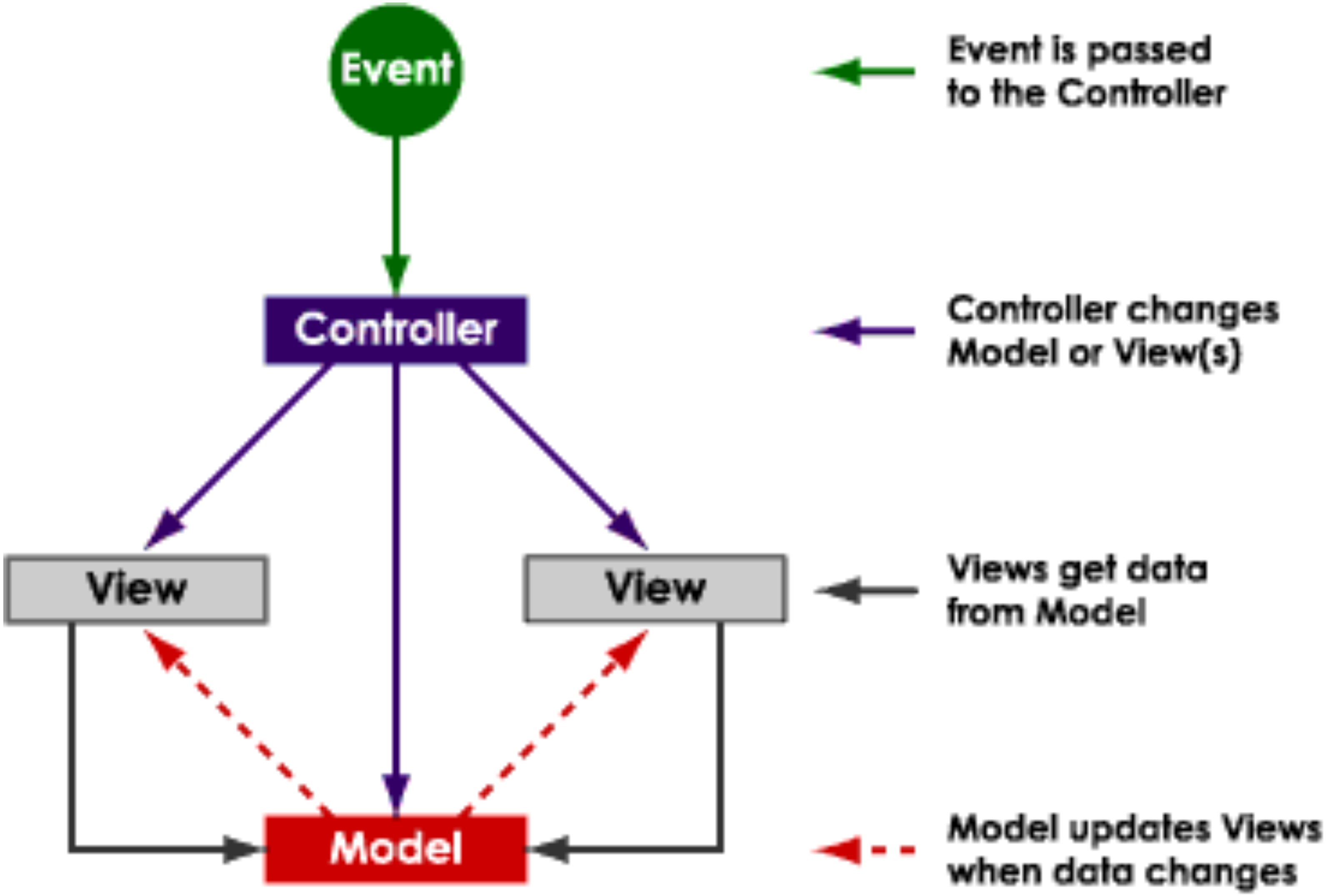
View

- The view's responsibilities
  - Display the state of the model to the user
- At some point, the model (a.k.a. the observable) must registers the views (a.k.a. observers) so the model can notify the observers that its state has changed

Controller

- The controller's responsibilities
  - Accept user input
    - Button clicks, key presses, mouse movements, slider bar changes
  - Send messages to the model, which may in turn notify it observers
  - Send appropriate messages to the view
- In Java, listeners are controllers

from http://www.enode.com/x/markup/tutorial/mvc.html)

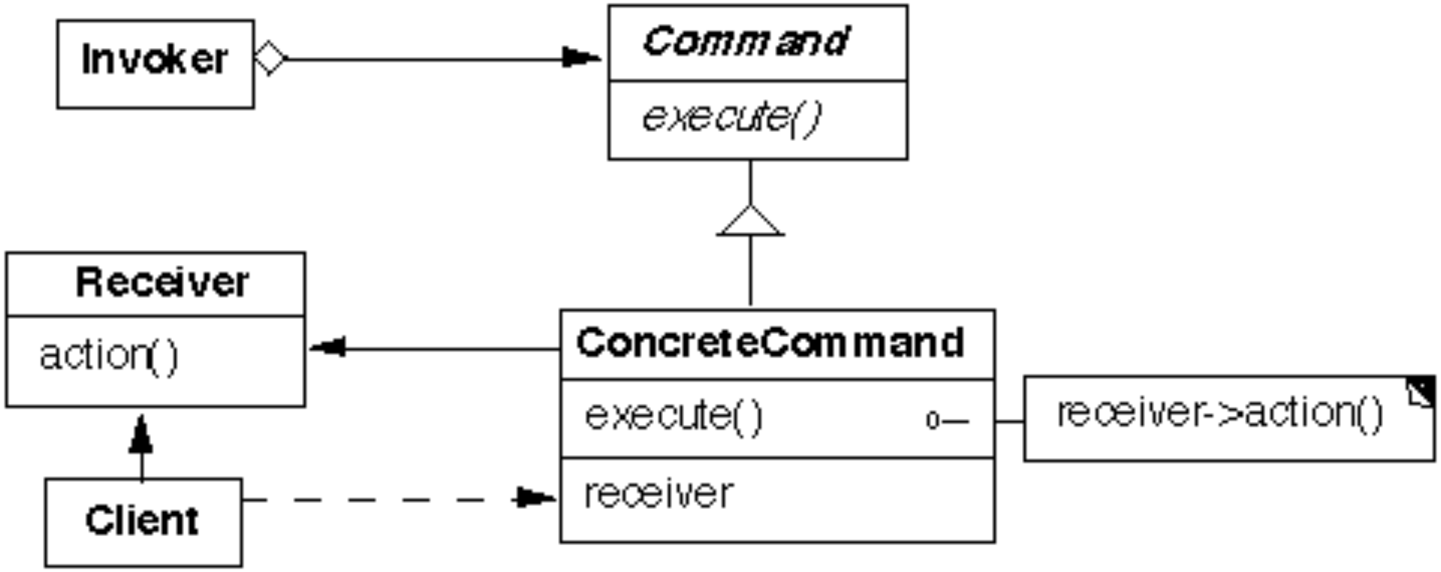

Event
Controller
View
View
Model
Event is passed to the Controller
Controller changes Model or View(s)
Views get data from Model
Model updates Views when data changes

Command pattern

- **Synopsis** or **Intent**: Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations
- **Context**: You want to model the time evolution of a program:
  - What needs to be done, e.g. queued requests, alarms, conditions for action
  - What is being done, e.g. which parts of a composite or distributed action have been completed
  - What has been done, e.g. a log of undoable operations
- *What are some applications that need to support undo?*
  - Editor, calculator, database with transactions
  - Perform an execute at one time, undo at a different time
- **Solution**: represent units of work as Command objects
  - Interface of a Command object can be a simple execute() method
  - Extra methods can support undo and redo
  - Commands can be persistent and globally accessible, just like normal objects

Command pattern, continued

- **Structure**:



Participants (the classes and/or objects participating in this pattern):

Command (Command) declares an interface for executing an operation

ConcreteCommand defines a binding between a Receiver object and an action
implements Execute by invoking the corresponding operation(s) on Receiver

Invoker asks the command to carry out the request

Receiver knows how to perform operations associated with carrying out the request

Client creates a ConcreteCommand object and sets its receiver

Command pattern, continued

- **Consequences:**
  - You can undo/redo any Command
    - Each Command stores what it needs to restore state
  - You can store Commands in a stack or queue
    - Command processor pattern maintains a history
  - It is easy to add new Commands, because you do not have to change existing classes
    - Command is an abstract class, from which you derive new classes
    - execute(), undo() and redo() are polymorphic functions

Design Patterns are NOT

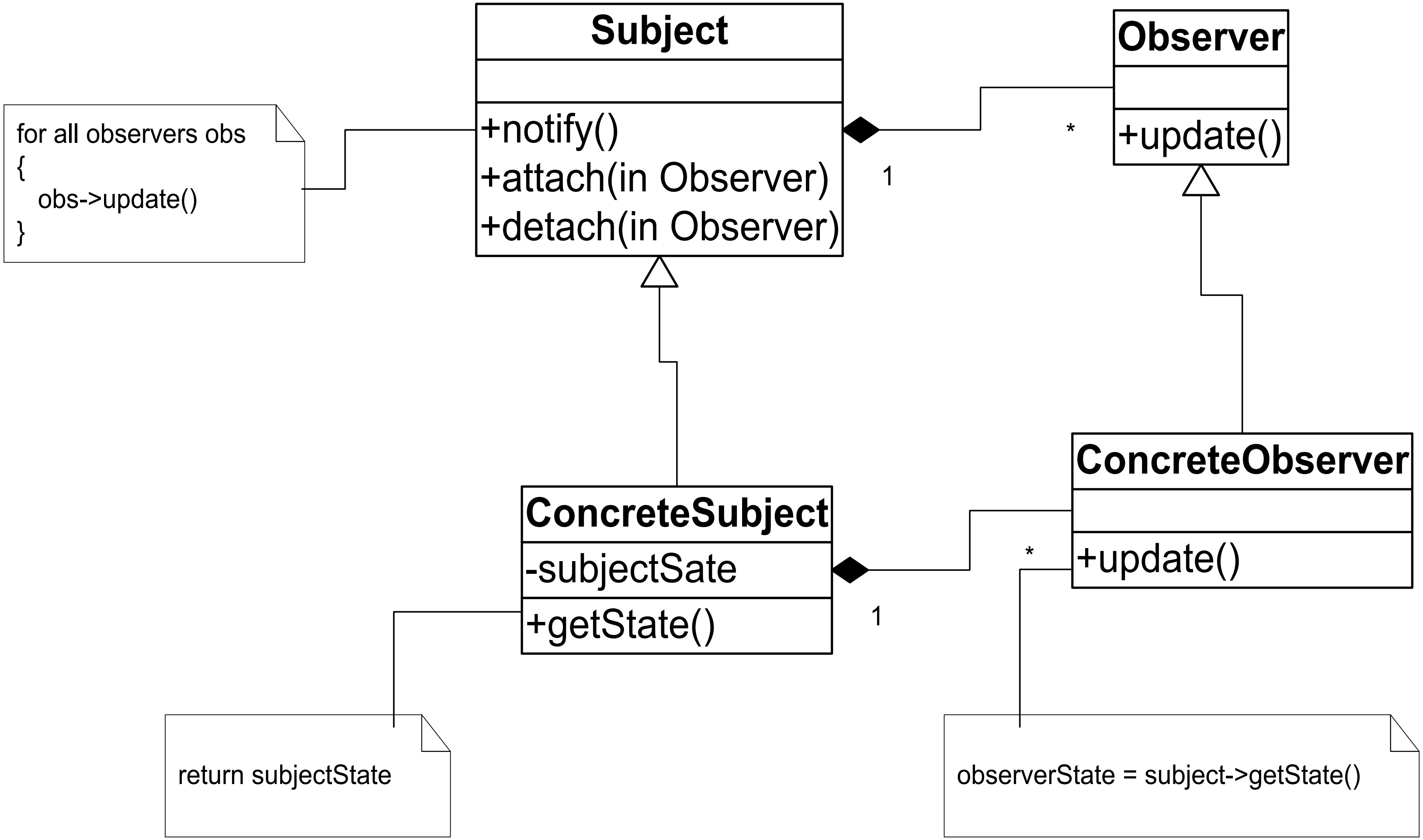
- Data structures that can be encoded in classes and reused *as is* (i.e., linked lists, hash tables)
- Complex domain-specific designs
  (for an entire application or subsystem)
- If they are not familiar data structures or complex domain-specific subsystems, *what are they*?

- They are:
  - "Descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context."

Observer pattern

- Intent:
  - Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically
- Used in Model-View-Controller framework
  - Model is problem domain
  - View is windowing system
  - Controller is mouse/keyboard control
- *How can Observer pattern be used in other applications?*
- JDK's Abstract Window Toolkit (listeners)
- Java's Thread monitors, notify(), etc.

Structure of Observer Pattern

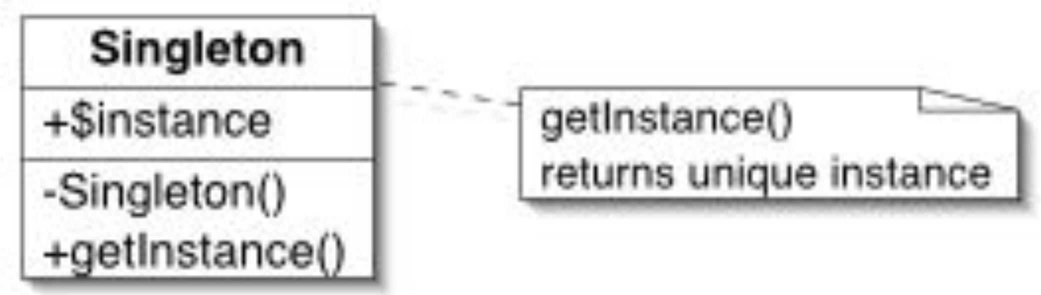


Three Types of Patterns

- **Creational patterns:**
  - Deal with initializing and configuring classes and objects
- **Structural patterns**:
  - Deal with decoupling interface and implementation of classes and objects
  - Composition of classes or objects
- **Behavioral patterns**:
  - Deal with dynamic interactions among societies of classes and objects
  - How they distribute responsibility

Singleton pattern (creational)

- Ensure that a class has only one instance and provide a global point of access to it
  - *Why not use a global variable?*



```
class Singleton
{ public:
     static Singleton* getInstance();
  protected: //Why are the following protected?
     Singleton();
     Singleton(const Singleton&);
     Singleton& operator= (const Singleton&);
 private: static Singleton* instance;
};
Singleton *p2 = p1->getInstance();
```

Creational Patterns

- **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 (sole) instance

Structural patterns

- Describe ways to assemble objects to realize new functionality
  - Added flexibility inherent in object composition due to ability to change composition at run-time
  - not possible with static class composition
- Example: Proxy
  - ***Proxy***: acts as convenient surrogate or placeholder for another object.
    - Remote Proxy: local representative for object in a different address space
    - Virtual Proxy: represent large object that should be loaded on demand
    - Protected Proxy: protect access to the original object

Structural Patterns

- **Adapter:**
  - Translator adapts a server interface for a client
- **Bridge**:
  - Abstraction for binding one of many implementations
- **Composite**:
  - Structure for building recursive aggregations
- **Decorator**:
  - Decorator extends an object transparently
- **Facade**:
  - Simplifies the interface for a subsystem
- **Flyweight**:
  - Many fine-grained objects shared efficiently.
- **Proxy**:
  - One object approximates another

Behavioral Patterns

- **Chain of Responsibility**:
  - Request delegated to the responsible service provider
- **Command**:
  - Request or Action is first-class object, hence re-storable
- **Iterator**:
  - Aggregate and access elements sequentially
- **Interpreter**:
  - Language interpreter for a small grammar
- **Mediator**:
  - Coordinates interactions between its associates
- **Memento**:
  - Snapshot captures and restores object states privately

Which ones do you think you have seen somewhere?

Behavioral Patterns (cont.)

- **Observer**:
  - Dependents update automatically when subject changes
- **State**:
  - Object whose behavior depends on its state
- **Strategy**:
  - Abstraction for selecting one of many algorithms
- **Template Method**:
  - Algorithm with some steps supplied by a derived class
- **Visitor**:
  - Operations applied to elements of a heterogeneous object structure

Patterns in software libraries

- AWT and Swing use Observer pattern
- Iterator pattern in C++ template library & JDK
- Façade pattern used in many student-oriented libraries to simplify more complicated libraries!
- Bridge and other patterns recurs in middleware for distributed computing frameworks
- …

More software patterns

- Design patterns
  - idioms **(low level, C++): Jim Coplein, Scott Meyers**
    - **I.e., when should you define a virtual destructor?**
  - design **(micro-architectures) [Gamma-GoF]**
  - architectural **(systems design): layers, reflection, broker**
    - Reflection makes classes self-aware, their structure and behavior accessible for adaptation and change:
      Meta-level provides self-representation, base level defines the application logic
- *Java Enterprise Design Patterns* (distributed transactions and databases)
  - E.g., ACID Transaction: *A*tomicity (restoring an object after a failed transaction), *C*onsistency, *I*solation, and *D*urability
- **Analysis patterns** (recurring & reusable analysis models, from various domains, i.e., accounting, financial trading, health care)
- **Process patterns** (software process & organization)

Benefits of Design Patterns

- Design patterns enable large-scale reuse of software architectures and also help document systems
- Patterns explicitly capture expert knowledge and design tradeoffs and make it more widely available
- Patterns help improve developer communication
- Pattern names form a common vocabulary

Web Resources

- http://home.earthlink.net/~huston2/dp/
- http://www.dofactory.com/
- http://hillside.net/patterns/
- *Java Enterprise Design Patterns*