Design patterns

Glenn D. Blank

Definitions

- A *pattern* is a recurring solution to a standard problem, in a context.
- "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."

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?*
- 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
  - 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

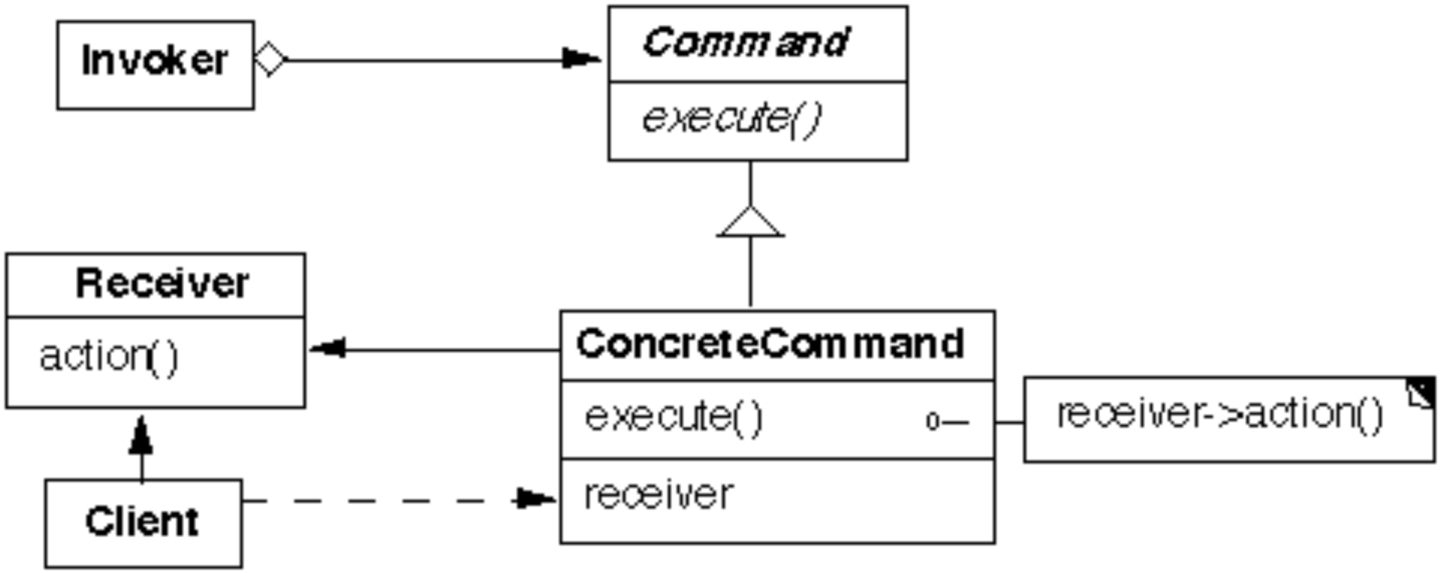
- Design patterns have 4 essential elements:
  - Pattern name: increases vocabulary of designers
  - Problem: intent, context, when to apply
  - Solution: UML-like structure, abstract code
  - Consequences: results and tradeoffs

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."

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

GoF Patterns

Creational	Structural	Behavioral
Abstract Factory	Adapter	Chain of Responsibility
Builder	Bridge	Command
Factory Method	Composite	Interpreter
Prototype	Decorator	Iterator
Singleton	Façade	Mediator
	Flyweight	Memento
	Proxy	Observer
		State
		Strategy
		Template Method
		Visitor

E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

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

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
- …

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