Design Patterns

Acknowledgements

Materials based on a number of sources
- D. Levine and D. Schmidt
- R. Helm
- Gamma et al
- S. Konrad
Motivation

• Developing software is hard
• Designing reusable software is more challenging
  – finding good objects and abstractions
  – flexibility, modularity, elegance reuse
  – takes time for them to emerge, trial and error
• Successful designs do exist
  – exhibit recurring class and object structures

Design Pattern

• Describes recurring design structure
  – names, abstracts from concrete designs
  – identifies classes, collaborations, responsibilities
  – applicability, trade-offs, consequences
Becoming a Chess Master

- *First learn rules and physical requirements*
  - e.g., names of pieces, legal movements, chess board geometry and orientation, etc.
- *Then learn principles*
  - e.g., relative value of certain pieces, strategic value of center squares, power of a threat, etc.
- *To become a Master of chess, one must study the games of other masters*
  - These games contain patterns that must be understood, memorized, and applied repeatedly.
- There are hundreds of these patterns.

Becoming a Software Design Master

- *First learn rules*
  - e.g., algorithms, data structures, and languages of software.
- *Then learn principles*
  - e.g., structured programming, modular programming, object-oriented programming, etc.
- *To become a Master of SW design, one must study the designs of other masters*
  - These designs contain patterns that must be understood, memorized, and applied repeatedly.
- There are hundreds of these patterns.
Design Patterns

• Design patterns represent solutions to problems that arise when developing software within a particular context
  – “Patterns == problem/solution pairs in a context”
• Patterns capture the static and dynamic structure and collaboration among key participants in software designs
  – Especially good for describing how and why to resolve non-functional issues

• Patterns facilitate reuse of successful software architectures and designs.

Design Patterns: Applications

• Wide variety of application domains:
  – drawing editors, banking, CAD, CAE, cellular network management, telecomm switches, program visualization

• Wide variety of technical areas:
  – user interface, communications, persistent objects, O/S kernels, distributed systems
What Is a Design Pattern (1)

“Each 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 in the same way twice”

Christopher Alexander, A Pattern Language, 1977

Context: City Planning and Building architectures

What Is a Design Pattern (2)

A pattern has 4 essential elements:

- Pattern name
- Problem
- Solution
- Consequences
Pattern Name

- A handle used to describe:
  - a design problem,
  - its solutions and
  - its consequences

- Increases design vocabulary

- Makes it possible to design at a higher level of abstraction

- Enhances communication

*But finding a good name is often difficult*

Problem

- Describes when to apply the pattern
- Explains the problem and its context
- Might describe specific design problems or class or object structures
- May contain a list of conditions
  - must be met
  - before it makes sense to apply the pattern
Solution

- Describes the elements that make up the design,
  - their relationships,
  - responsibilities and collaborations
- *Does not* describe specific concrete implementation
- Abstract description of design problems and
  - how the pattern solves it

Consequences

- Results and trade-offs of applying the pattern
- Critical for:
  - evaluate design alternatives and
  - understand costs and
  - understand benefits of applying the pattern
- Includes the impacts of a pattern on a system’s:
  - flexibility,
  - extensibility
  - portability
Design Patterns Are NOT

• Designs 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)

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

Where Design Patterns Are Used

• Object-Oriented Programming Languages:
  • more amenable to implementing design patterns

• Procedural languages: need to define
  • Inheritance,
  • Polymorphism and
  • Encapsulation
How to Describe Design Patterns

- Graphical notation is not sufficient
- In order to reuse design decisions,
  - alternatives and trade-offs that led to the decisions are important
- Concrete examples are also important

A Design Pattern

- Describes a recurring design structure
  - names, abstracts from concrete designs
  - identifies classes, collaborations, responsibilities
  - applicability, trade-offs, consequences
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.

- **Key forces:**
  - There may be many observers.
  - Each observer may react differently to the same notification.
  - The subject should be as decoupled as possible from the observers.
  - Allow observers to change independently of the subject.
Structure of the Observer Pattern

Subject

- notify()
- attach(observer)
- detach(observer)

Concrete Subject

- subject state
- get_state()
- set_state()

Concrete Observer

- update()

Observer

- update()

Collaboration in the Observer Pattern

Concrete Subject

- attach(o1)
- attach(o2)
- set_state()
- notify()
- update()
- get_state()

Concrete Observer

- update()
- get_state()
Design Pattern Descriptions

- **Main Parts:**
  - **Name and Classification** (see table in two more slides)
  - **Intent**: Problem and Context
  - **Also known as** (other well-known names)
  - **Motivation**: scenario illustrates a design problem
  - **Applicability**: situations where pattern can be applied
  - **Structure**: graphical representation of classes (class diagram, interaction diagram)
  - **Participants**: objects/classes and their responsibilities
  - **Collaborations**: how participants collaborate
  - **Consequences**: trade-offs and results
  - **Implementation**: pitfalls, hints, techniques for coding; language-specific issues
  - **Sample Code**
  - **Known Uses**: examples of pattern in real systems
  - **Related Patterns**: closely related; what are diffs.

- Pattern descriptions are often independent of programming language or implementation details

Design Pattern Space

- **Creational patterns**: Deal with initializing and configuring classes and objects
- **Structural patterns**: Deal with decoupling interface and implementation of classes and objects
- **Behavioral patterns**: Deal with dynamic interactions among societies of classes and objects

### Categorize Design Patterns

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Creational</th>
<th>Structural</th>
<th>Behavioral</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scope</td>
<td>Factory Method</td>
<td>Adapter (class)</td>
<td>Interpreter</td>
</tr>
<tr>
<td>Class</td>
<td>Abstract Factory</td>
<td>Adapter (object)</td>
<td>Chain of Responsibility</td>
</tr>
<tr>
<td>Object</td>
<td>Builder</td>
<td>Bridge</td>
<td>Command</td>
</tr>
<tr>
<td></td>
<td>Prototype</td>
<td>Composite</td>
<td>Iterator</td>
</tr>
<tr>
<td></td>
<td>Singleton</td>
<td>Decorator</td>
<td>Mediator</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Flyweight</td>
<td>Memento</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Facade</td>
<td>Observer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Proxy</td>
<td>State</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Strategy</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Visitor</td>
</tr>
</tbody>
</table>

### Categorization Terms

- **Scope**: domain over which a pattern applies
  - **Class Scope**:
    - relationships between base classes and their subclasses
    - Static semantics
  - **Object Scope**:
    - relationships between peer objects
    - Can be changed at runtime
    - More dynamic
Purpose of Patterns

- Creational:
  - Class: defer some part of object creation to subclasses
  - Object: Defer object creation to another object
- Structural:
  - Class: use inheritance to compose classes
  - Object: describe ways to assemble classes
- Behavioral:
  - Class: use inheritance to describe algs and flow of control
  - Object: describes how a group of objects cooperate to perform task that no single object can complete

Terminology

- Signature: \textit{(of an operation)}
  - operation name,
  - objects taken as parameters, and
  - operation’s return value
- Interface: \textit{(of an object)}
  - Set of all signatures defined by an object’s operations
  - Characterizes the complete set of requests that can be sent to object.
  - Key to OO technology
Creational Patterns

- **Factory Method**: method in a derived class creates associations
- **Abstract Factory**: Factory for building related objects
- **Builder**: Factory for building complex objects incrementally
- **Prototype**: Factory for cloning new instances from a prototype
- **Singleton**: Factory for a singular (sole) instance

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 is first-class object
- **Iterator**: Aggregate elements are accessed sequentially
- **Interpreter**: language interpreter for a small grammar
- **Mediator**: coordinates interactions between its associates
- **Memento**: snapshot captures and restores object states privately
- **Observer**: dependents update automatically when subject changes
- **State**: object whose behavior depends on its state

### Behavior Patterns (more)

- **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
When to Use Patterns

- Solutions to problems that recur with variations
  - No need for reuse if problem only arises in one context
- Solutions that require several steps:
  - Not all problems need all steps
  - Patterns can be overkill if solution is a simple linear set of instructions
- Solutions where the solver is more interested in the existence of the solution than its complete derivation
  - Patterns leave out too much to be useful to someone who really wants to understand
    - They can be a temporary bridge

What Makes it a Pattern

A Pattern must:

- Solve a problem
  - must be useful

- Have a context
  - describe where the solution can be used

- Recur
  - relevant in other situations

- Teach
  - provide sufficient understanding to tailor the solution

- have a name
  - referenced consistently
Class Scope

- Class Creational: abstract how objects are instantiated
  - hide specifics of creation process
  - may want to delay specifying a class name explicitly when instantiating an object
  - just want a specific protocol
Example Class Creational

- Use of **Factory Method**: instantiate members in base classes with objects created by subclasses.
- Abstract **Application class**: create application-specific documents conforming to particular **Document type**
- **Application** instantiates these **Document objects** by calling the factory method **CreateDocument**
- Method is overridden in classes derived from Application
- Subclass **DrawApplication** overrides **CreateDocument** to return a **DrawDocument** object
Class Structural

- **Class Structural**: use inheritance to compose protocols or code

- Example:
  - *Adapter Pattern*: makes one interface (Adaptee’s) conform to another --> uniform abstraction of different interfaces.
  - Class Adapter inherits privately from an Adaptee class.
  - Adapter then expresses its interface in terms of the Adaptee’s.

Adapter Example (class)
Adapter Example (object)

Class Behavioral

- **Class Behavioral**: capture how classes cooperate with their subclasses to satisfy semantics.
  - **Template Method**: defines algorithms step by step.
  - Each step can invoke an abstract method (that must be defined by the subclass) or a base method.
  - Subclass must implement specific behavior to provide required services
Object Scope

- Object Patterns all apply various forms of non-recursive object composition.
- Object Composition: most powerful form of reuse
- Reuse of a collection of objects is better achieved through variations of their composition, rather than through subclassing.

Object Creational

- **Creational Object Patterns**: abstract how sets of objects are created
- **Example**:
  - *Abstract Factory*: create “product” objects through generic interface
    - Subclasses may manufacture specialized versions or compositions of objects as allowed by this generic interface
  - **User Interface Toolkit**: 2 types of scroll bars (Motif and Open Look)
    - Don’t want to hard-code specific one; an environment variable decides
  - **Class Kit**:
    - encapsulates scroll bar creation (and other UI entities);
    - an abstract factory that abstracts the specific type of scroll bar to instantiate
    - Subclasses of Kit refine operations in the protocol to return specialized types of scroll bars.
    - Subclasses *MotifKit* and *OpenLookKit* each have scroll bar operation.
**Object Structural**

- **Object Structural**: 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**: 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
**Object Behavioral**

- **Object Behavioral**: Describe how a group of peer objects cooperate to perform a task that can be carried out by itself.

- **Example**:
  - **Strategy Pattern**: objectifies an algorithm
  - **Text Composition Object**: support different line breaking algorithms
    - Don't want to hard-code all algs into `text composition` class/subclasses
  - Objectify different algs and provide them as `Compositor` subclasses (contains criteria for line breaking strategies)
  - Interface for `Compositors` defined by `Abstract Compositor` Class
    - Derived classes provide different layout strategies (simple line breaks, left/right justification, etc.)
  - Instances of Compositor subclasses couple with text composition at run-time to provide text layout
  - Whenever `text composition` has to find line breaks, forwards the responsibility to its current `Compositor object`. 
Object Behavioral Example

- **Iterator Pattern**: Iteration over a recursive structure
- Traversal strategies for a given structure:
  - Extract and implement each traversal strategy in an *Iterator* class.
  - *Iterators* objectify traversal algs over recursive structures
  - Different *iterators* can implement pre-order, in-order, post-order traversals
  - Require nodes in structure to provide services to enumerate their sub-structures
  - Don’t need to hard-code traversal algs throughout classes of objects in composite structure
  - *Iterators* may be replaced at run-time to provide alternate traversals.

Object Structural Example

- **Facade Pattern (Wrapper)**: describes how to flexibly attach additional properties and services to an object
  - Can be nested recursively; compose more complex object structures
- **User Interface Example**:  
  - A *Facade* containing a single UI component can add decorations such as border, shadows, scroll bars, or services (scrolling and zooming)
  - *Facade* must conform to interface of its wrapped component and forward messages to it
  - *Facade* can perform additional actions (e.g., drawing border around component) either before or after forwarding a message.
Benefits of Design Patterns

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

Drawbacks to Design Patterns

- Patterns do not lead to direct code reuse
- Patterns are deceptively simple
- Teams may suffer from pattern overload
- Patterns are validated by experience and discussion rather than by automated testing
- Integrating patterns into a SW development process is a human-intensive activity.
Suggestions for Effective Pattern Use

- Do not recast everything as a pattern
  - Instead, develop strategic domain patterns and reuse existing tactical patterns
- Institutionalize rewards for developing patterns
- Directly involve pattern authors with application developers and domain experts
- Clearly document when patterns apply and do not apply
- Manage expectations carefully.