Software Design Patterns

Jonathan I. Maletic, Ph.D.

<SDML>
Department of Computer Science
Kent State University
Background

• Search for recurring successful designs – emergent designs from practice (via trial and error)
• Supporting higher levels of reuse (i.e., reuse of designs) is quite challenging
• Described in Gama, Helm, Johnson, Vlissides 1995 (i.e., “gang of 4 book”)
• Based on work by Christopher Alexander (an Architect) on building homes, buildings and towns.
Background

• Design patterns represent solutions to problems that arise when developing software within a particular context, e.g., problem/solution pairs within a given context
• Describes recurring design structures
• Describes the context of usage
Background

- Patterns capture the static and dynamic structure and collaboration among key participants in software designs.
- Especially good for describing how and why to resolve nonfunctional issues.
- Patterns facilitate reuse of successful software architectures and designs.
Origins of Design Patterns

“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
Elements of Design Patterns

• Design patterns have four essential elements:
  – Pattern name
  – Problem
  – Solution
  – Consequences
Pattern Name

• A handle used to describe:
  – a design problem
  – its solutions
  – its consequences

• Increases design vocabulary
• Makes it possible to design at a higher level of abstraction
• Enhances communication

“The Hardest part of programming is coming up with good variable [function, and type] names.”

J. Maletic
Problem

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

• Describes the elements that make up the
  – design
  – relationships
  – responsibilities
  – 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:
  – evaluating design alternatives
  – understanding costs
  – understanding 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 They are Used

• Object-Oriented programming languages [and paradigm] are more amenable to implementing design patterns

• Procedural languages: need to define
  – Inheritance
  – Polymorphism
  – Encapsulation
Describing Design Patterns

• Graphical notation is generally not sufficient
• In order to reuse design decisions the alternatives and trade-offs that led to the decisions are critical knowledge
• Concrete examples are also important
• The history of the why, when, and how set the stage for the context of usage
Design Patterns

• Describe a recurring design structure
  – Defines a common vocabulary
  – Abstracts from concrete designs
  – Identifies classes, collaborations, and responsibilities
  – Describes applicability, trade-offs, and consequences
Example: Compiler
Façade Pattern

• Intent
  – Provide a unified interface to a set of interfaces in a subsystem.
  – Façade defines a higher-level interface that makes the subsystem easier to use
Structure

Facade

+interface()

Subsystem
Design Pattern Descriptions

- **Name and Classification**: Essence of pattern
- **Intent**: What it does, its rationale, its context
- **AKA**: Other well-known names
- **Motivation**: Scenario illustrates a design problem
- **Applicability**: Situations where pattern can be applied
- **Structure**: Class and interaction diagrams
- **Participants**: Objects/classes and their responsibilities
- **Collaborations**: How participants collaborate
- **Consequences**: Trade-offs and results
- **Implementation**: Pitfalls, hints, techniques, etc.
- **Sample Code**
- **Known Uses**: Examples of pattern in real systems
- **Related Patterns**: Closely related patterns
Example: Stock Quote Service

Real time Market Data Feed

Stock Quotes

Customer

Customer

Customer

Customer

Observers
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 to allow observers to change independently of the subject
for all observers obs
{ obs->update() }

return subjectState

observerState = subject->getState()
Collaborations in Observer Pattern

S1 : ConcreteSubject

obs1 : ConcreteObserver

obs2 : ConcreteObserver

setState() → notify() → update() → getState() → update() → getState()
Example: List and Iterator

- Abstract list (array or linked structure)
- Separate iterator that allows sequential access to the list structure without exposing the underlying representation
- Used in STL
- AKA: Cursor
Iterator Pattern

```java
{  
  ...  
  return new ConcreteIterator(this);  
  ...  
}
```
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
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

• **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
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
## Design Pattern Space

<table>
<thead>
<tr>
<th>Scope</th>
<th>Class</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Creational</td>
<td>Structural</td>
</tr>
<tr>
<td>Class</td>
<td>Factory method</td>
<td>Adapter (class)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Object</td>
<td>Abstract factory</td>
<td>Adapter (object)</td>
</tr>
<tr>
<td></td>
<td>Builder</td>
<td>Bridge</td>
</tr>
<tr>
<td></td>
<td>Prototype</td>
<td>Composite</td>
</tr>
<tr>
<td></td>
<td>Singleton</td>
<td>Decorator</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Façade</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Flyweight</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Proxy</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Categorization Terms

• Scope is the domain over which a pattern applies
  – Class Scope: relationships between base classes and their subclasses (static semantics)
  – Object Scope: relationships between peer objects

• Some patterns apply to both scopes.
Class:: Creational

- Abstracts how objects are instantiated
- Hides specifics of the creation process
- May want to delay specifying a class name explicitly when instantiating an object
- Just want a specific protocol
Example

• Use *Factory Method* to instantiate members in base classes with objects created by subclasses

• Abstract *Application* class: create application-specific documents conforming to a 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
Factory Method Pattern

```
{ product=FactoryMethod() }

{ Return new ConcreteProduct() }

ConcreteCreator

+FactoryMethod()

ConcreteProduct

«instance»

+FactoryMethod()
+Operation1()

Creator

Product

J. Maletic

34
Class:: Structural

• Use inheritance to compose protocols or code

• Example:
  – *Adapter Pattern*: makes one interface (Adaptee’s) conform to another
  – Gives a 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 Pattern

Client

Target
+request()
Class:: Behavioral

• Captures how classes cooperate with their subclasses to satisfy semantics.

Example:
• **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
Template Method Pattern

AbstractClass

+TemplateMethod()
+operation1()
+operation2()

ConcreteClass

+operation1()
+operation2()

{ 
  operation1();
  ...
  operation2();
  ....
}
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 sub classing.
Object:: Creational

• Abstracts 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.
Abstract Factory Pattern

AbstractFactory

+CreateProductA()
+CreateProductB()

ConcreteFactory1
+CreateProductA()
+CreateProductB()

ConcreteFactory2
+CreateProductA()
+CreateProductB()

Client

AbstractProductA

ProductA1

AbstractProductB

ProductB1

ProductB2

ProductA2

«instance»
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
Proxy Pattern

```
{ realSubject->request();
  ...
}
```
Object:: Structural - example

• Implement ODBC

• Could be done with an adaptor unless you need to extend both the interface and implementation

• Or if you know the implementation will change often

• The implementation class defines what types of things need to be supported
Pattern Bridge

ODBC

1

ODBC Implementation

1 *

DB2 ODBC Driver

Oracle ODBC Driver

Informix ODBC Driver
Structure of Bridge

Client

Abstraction
  +Operation()

Implementor
  +OperationIMP()

Refined Abstraction

ConcreteImplementor1
  +OperationIMP()

ConcreteImplementor2
  +OperationIMP()

imp->OperationIMP();
Object:: Behavioral

Describes how a group of peer objects cooperate to perform a task that can be carried out by itself.

Example:
- **Strategy Pattern:** objectifies an algorithm (algorithm to first class object)
- Text Composition Object: support different line breaking algorithms
  - Don’t want to hard-code all algorithms into text composition class/subclasses
  - Simple, TeX, Array, Word, etc.
- Objectify each and provides them as Compositor subclasses
- Interface for Compositors defined by an 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.
Strategy Pattern

```cpp
{  
    ...  
    strat->algInterface()  
    ...  
}
```

```plaintext
Strategy
+AlgInterface()

Context
+ContextInterface()
+strat

ConcreteStrategyA
+AlgInterface()

ConcreteStrategyB
+AlgInterface()

ConcreteStrategyC
+AlgInterface()
```
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 and be useful
  – Have a context and can describe where the solution can be used
  – Recur in relevant situations
  – Provide sufficient understanding to tailor the solution
  – Have a name and be referenced consistently
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.
- 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 software development process is a human-intensive activity.
Suggestions for Effective 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.
References

• Gama, Helm, Johnson, Vlissides, *Design Patterns Elements of Reusable Object-Oriented Software*, Addison Wesley, 1995

• B. Cheng – Michigan State University
Web Resources

• [http://hillside.net/patterns/](http://hillside.net/patterns/)