What is a Design Pattern?

- Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the problem.
- Novelists, playwrights, and other writers rarely invent new stories.
- Often ideas are reused, such as the “Tragic Hero” from Hamlet or Macbeth.
- Designers reuse solutions also, preferably the “good” ones.
  - Experience is what makes one an ‘expert’
- Problems are addressed without rediscovering solutions from scratch.
  - “My wheel is rounder”
Design Patterns

*Design Patterns* are the best solutions for the re-occurring problems in the application programming environment.

• Nearly a universal standard
• Responsible for design pattern analysis in other areas, including GUIs.
• Mainly used in Object Oriented programming
Design Pattern Elements

1. Pattern Name
Handle used to describe the design problem
Increases vocabulary
Eases design discussions
Evaluation without implementation details
Design Pattern Elements

1. Problem
Describes when to apply a pattern
May include conditions for the pattern to be applicable
Symptoms of an inflexible design or limitation
Design Pattern Elements

1. Solution
Describes elements for the design
Includes relationships, responsibilities, and collaborations
Does not describe concrete designs or implementations
A pattern is more of a template
Design Pattern Elements

1. Consequences
Results and Trade Offs
Critical for design pattern evaluation
Often space and time trade offs
Language strengths and limitations
(Broken into benefits and drawbacks for this discussion)
Design patterns can be subjective. One person’s pattern may be another person’s primitive building block.

The focus of the selected design patterns are:
Object and class communication
Customized to solve a general design problem
Solution is context specific
Design patterns in Smalltalk MVC

• The Model/View/Controller triad of classes is used to build user interfaces in Smalltalk-80

• MVC consists of three kinds of objects.
  • M-\(\rightarrow\)MODEL is the Application object.
  • V-\(\rightarrow\)View is the screen presentation.
  • C-\(\rightarrow\)Controller is the way the user interface reacts to user input.

MVC decouples to increase flexibility and reuse.
Design patterns in Smalltalk

MVC

• MVC decouples views and models by establishing a subscribe/notify protocol between them.

A view must ensure that its appearance must reflects the state of the model.

Whenever the model’s data changes, the model notifies views that depends on it.

You can also create new views for a model without Rewriting it.
Design Patterns in smalltalkMVC

• The below diagram shows a model and three views.
• The model contains some data values, and the views defining a spreadsheet, histogram, and pie chart display these data in various ways.
• The model communicates with it’s values change, and the views communicate with the model to access these values.
• Feature of MVC is that views can be nested.

Easy to maintain and enhancement.
Design Patterns in smalltalkMVC

A B C D
X 15 35 35 15
Y 10 40 30 20
Z 10 40 30 20

Relative Percentages
A = 10%
B = 40%
C = 30%
D = 20%

Change notification Requests, modifications
Application data
Describing Design Patterns:

- Graphical notations, while important and useful, aren’t sufficient. They capture the end product of the design process as relationships between classes and objects.

By using a consistent format we describe the design pattern.
Each pattern is divided into sections according to the following template.
Describing Design Patterns

Pattern Name and Classification:
• It conveys the essence of the pattern succinctly; a good name is vital, because it will become part of design vocabulary.

Intent: What does the design pattern do?
• What is it’s rational and intend?
• What particular design issue or problem does it address?

Also Known As: Other well-known names for the pattern, if any.

Motivation:
• A scenario that illustrates a design problem and how the class and object structures in the
Describing Design Patterns

Applicability:

• Applicability: What are the situations in which the design patterns can be applied?
• What are example of the poor designs that the pattern can address?
• How can recognize situations?
• Structure: Graphical representation of the classes in the pattern using a notation based on the object Modeling Technique (OMT).
• Participants: The classes and/or objects participating in the design pattern and their responsibilities.

Structure:

• Graphical representation of the classes in the pattern using a notation based on the object Modeling Technique (OMT).
Describing Design Patterns

Collaborations:
• How the participants collaborate to carry out their responsibilities.

Consequences:
• How does the pattern support its objectives?
• What are the trade-offs and result of using the pattern?
• What aspect of the system structure does it let vary independently?

Implementation:
• What pitfalls, hints, or techniques should be aware of when implementing the pattern?
Describing Design Patterns

Sample Code:
- Code fragments that illustrate how might implement the pattern in c++ or Smalltalk.

Known Uses:
Examples of the pattern found in real systems.

Related Patterns:
What design patterns are closely related to this one?
What are the imp differences?
With Which other patterns should this one be used?
Abstract Factory: Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Adaptor: Convert the interface of a class into another interface clients expect.

Bridge: Decouple an abstraction from its implementation so that two can vary independently.
The Catalog of Design Patterns

- **Builder:**
  - Separates the construction of the complex object from its representation so that the same constriction process can create different representations.

- **Chain of Responsibility:** Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an objects handles it.
The Catalog of Design Pattern

- **Command:**
  - Encapsulate a request as an object, thereby letting parameterize clients with different requests, queue or log requests, and support undoable operations.

- **Composite:**
  - Compose objects into three objects to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
The Catalog of Design Pattern

• Decorator:
  • Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

• Façade: Provide a unified interface to a set of interfaces in a subsystem's Facade defines a higher-level interface that makes the subsystem easier to use.
The Catalog of Design Pattern

• Factory Method:
  • Defines an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

• Flyweight:
  • Use sharing to support large numbers of fine-grained objects efficiently.

• Interpreter:
  • Given a language, defining a representation of its grammar along with an interpreter that uses the representation to interpret sentences in the language.
The Catalog of Design Pattern

• Iterator:
• Provide a way to access the element of an aggregate object sequentially without exposing its underlying representation.

• Mediator:
• Define an object that encapsulate how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and let’s you very their interaction independently.
The Catalog of Design Pattern

• Memento: Without violating encapsulation, capture and externalize an object’s internal state so that object can be restored to this state later.

• Observer:

• Define a one-to-many dependency between objects so that when one object changes state, all it’s dependents are notified and updated automatically.
The Catalog of Design Pattern

• Prototype:
• Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

• Proxy: Provide a surrogate or placeholder for another object to control access to it.

• Singleton: Ensure a class has only one instance, and provide a point of access to it.

• State:
• Allow an object to alter its behavior when its internal state changes.
The Catalog of Design Pattern

- **Strategy:**
  - Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

- **Template Method:**
  - Define the Skeleton of an operation, deferring some steps to subclasses. Template method subclasses redefine certain steps of an algorithm without changing the algorithms structure.
The Catalog of Design Pattern

- Visitor:

- Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.
Organizing the Catalog

Two criteria

**Purpose**: what a pattern does

*Creational*: concern the process of object creation

*Structural*: the composition of classes or objects

*Behavioral*: characterize the ways in which classes or objects interact and distribute responsibility
## Organizing the Catalog

<table>
<thead>
<tr>
<th>Scope</th>
<th>Class</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td><strong>Creational</strong></td>
</tr>
<tr>
<td>Object</td>
<td>Object</td>
<td>Abstract Factory</td>
</tr>
<tr>
<td></td>
<td>Class</td>
<td>Factory Method</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Builder</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Prototype</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Singleton</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
How Design Patterns Solve Design Problems

• Finding Appropriate Objects
  o Decomposing a system into objects is the hard part
  o OO-designs often end up with classes with no counterparts in real world (low-level classes like arrays)
  o Strict modeling of the real world leads to a system that reflects today’s realities but not necessarily tomorrows
  o Design patterns identify less-obvious abstractions
How Design Patterns Solve Design Problems

• Determining Object Granularity
  o Objects can vary tremendously in size and number
  o **Facade pattern** describes how to represent subsystems as objects
  o **Flyweight pattern** describes how to support huge numbers of objects
Design Pattern relationship

• Mapping
Specifying Object Interfaces

• Interface:
  o Set of all signatures defined by an object’s operations
  o Any request matching a signature in the objects interface may be sent to the object
  o Interfaces may contain other interfaces as subsets

• Type:
  o Denotes a particular interfaces
  o An object may have many types
  o Widely different object may share a type
  o Objects of the same type need only share parts of their interfaces
  o A **subtype** contains the interface of its **super type**

• Dynamic binding, polymorphism
Specifying Object Implementations

• An object’s implementation is defined by its class
• The class specifies the object’s internal data and defines the operations the object can perform
• Objects is created by instantiating a class
  o an object = an instance of a class

• Class inheritance
  o parent class and subclass
Specifying Object Implementations (cont.)

• **Abstract class** versus **concrete class**
  - abstract operations

• **Override** an operation

• Class versus type
  - An object’s *class* defines how the object is implemented
  - An object’s *type* only refers to its interface
  - An object can have many types, and objects of different classes can have the same type
Specifying Object Implementations (cont.)

• Class versus Interface Inheritance
  - *class inheritance* defines an object’s implementation in terms of another object’s implementation (code and representation sharing)
  - *interface inheritance* (or *subtyping*) describes when an object can be used in place of another

• Many of the design patterns depend on this distinction
Specifying Object Implementations (cont.)

• Programming to an Interface, not an Implementation

• Benefits
  - clients remain unaware of the specific types of objects they use
  - clients remain unaware of the classes that implement these objects
Program to an interface, not an implementation

- Manipulate objects solely in terms of interfaces defined by abstract classes!

**Benefits:**

1. Clients remain unaware of the specific types of objects they use.
2. Clients remain unaware of the classes that implement the objects.
   Clients only know about abstract class(es) defining the interfaces

   - Do not declare variables to be instances of particular concrete classes
   - Use creational patterns to create actual objects.
Favor object composition over class inheritance

**White-box reuse:**
- Reuse by subclassing (class inheritance)
- Internals of parent classes are often visible to subclasses
- Works statically, compile-time approach
- Inheritance breaks encapsulation

**Black-box reuse:**
- Reuse by object composition
- Requires objects to have well-defined interfaces
- No internal details of objects are visible
Putting Reuse Mechanisms to Work

• Inheritance versus Composition
• Delegation
• Inheritance versus Parameterized Types
Inheritance versus Composition

• Two most common techniques for reuse
  ○ class inheritance
    ▪ white-box reuse
  ○ object composition
    ▪ black-box reuse

• Class inheritance
  ○ advantages
    ▪ static, straightforward to use
    ▪ make the implementations being reuse more easily
Inheritance versus Composition (cont.)

• Class inheritance (cont.)
  o disadvantages
    ▪ the implementations inherited can’t be changed at run time
    ▪ parent classes often define at least part of their subclasses’ physical representation
      ▪ breaks encapsulation
    ▪ implementation dependencies can cause problems when you’re trying to reuse a subclass
Inheritance versus Composition (cont.)

• Object composition
  o dynamic at run time
  o composition requires objects to respect each others’ interfaces
    ▪ but does not break encapsulation
  o any object can be replaced at run time
  o Favoring object composition over class inheritance helps you keep each class encapsulated and focused on one task
Inheritance versus Composition (cont.)

• Object composition (cont.)
  o class and class hierarchies will remain small
  o but will have more objects

• Favor object composition over class inheritance

• Inheritance and object composition should work together
Delegation

• Two objects are involved in handling a request: a receiving object delegates operations to its delegate.
Delegation (cont.)

• Makes it easy to compose behaviors at run-time and to change the way they’re composed.
• Disadvantage: dynamic, highly parameterized software is harder to understand than more static software.
• Delegation is a good design choice only when it simplifies more than it complicates.
• Delegation is an extreme example of object composition.
Inheritance versus Parameterized Types

• Let you define a type without specifying all the other types it uses, the unspecified types are supplied as parameters at the point of use
• Parameterized types, generics, or templates
• Parameterized types give us a third way to compose behavior in object-oriented systems
Inheritance versus Parameterized Types (cont.)

• Three ways to compose
  o *object composition* lets you change the behavior being composed at run-time, but it requires indirection and can be less efficient
  o *inheritance* lets you provide default implementations for operations and lets subclasses override them
  o *parameterized types* let you change the types that a class can use
Relating Run-Time and Compile-Time Structures

• An object-oriented program’s *run-time structure* often bears little resemblance to its *code structure*
• The code structure is frozen at compile-time
• A program’s run-time structure consists of rapidly changing networks of communicating objects
• *aggregation* versus *acquaintance* (association)
  o *part-of* versus *knows of*
The distinction between acquaintance and aggregation is determined more by intent than by explicit language mechanisms.

The system’s run-time structure must be imposed more by the designer than the language.
Designing for Change

• A design that doesn’t take change into account risks major redesign in the future
• Design patterns help you avoid this by ensuring that a system can change in specific ways
  ○ each design pattern lets some aspect of system structure vary independently of other aspects
Common Causes of Redesign

• Creating an object by specifying a class explicitly
• Dependence on specific operations
• Dependence on hardware and software platform
• Dependence on object representations or implementations
• Algorithmic dependencies
Common Causes of Redesign (cont.)

• Tight coupling
• Extending functionality by subclassing
• Inability to alter classes conveniently
Design for Change (cont.)

• Design patterns in application programs
  • Design patterns that reduce dependencies can increase internal reuse
  • Design patterns also make an application more maintainable when they’re used to limit platform dependencies and to layer a system
Design for Change (cont.)

• Design patterns in toolkits
  o A *toolkit* is a set of related and reusable classes designed to provide useful, general-purpose functionality
  o Toolkits emphasize code reuse. They are the object-oriented equivalent of subroutine libraries
  o Toolkit design is arguably harder than application design
Design for Change (cont.)

• Design patterns in framework
  o A **framework** is a set of cooperating classes that make up a reusable design for a specific class of software
  o You customize a framework to a particular application by creating application-specific subclasses of abstract classes from the framework
  o The framework dictates the *architecture* of your application
Design for Change (cont.)

- Design patterns in framework (cont.)
  - Frameworks emphasize *design reuse* over code reuse
  - When you use a *toolkit*, you write the main body of the application and call the code you want to reuse. When you use a *framework*, you reuse the main body and write the code it calls.
  - Advantages: build an application faster, easier to maintain, and more consistent to their users
• Design patterns in framework (cont.)
  - Mature frameworks usually incorporate several design patterns
  - People who know the patterns gain insight into the framework faster
  - Differences between framework and design pattern
    - Design patterns are more abstract than frameworks
    - Design patterns are smaller architectural elements than frameworks
    - Design patterns are less specialized than frameworks
How To Select a Design Pattern

- Consider how design patterns solve design problems.
- Scan Intent sections.
- Study how patterns interrelate.
- Study patterns of like purpose.
- Examine a Cause of redesign.
- Consider what should be variable in your design.
Figure 1.1: Design pattern relationships
How To Use a Design Pattern

• Read the pattern once through for an overview.
• Go Back and study the Structure, Participants, and Collaborations sections.
• Look At the Sample Code section to see a concrete Example of the pattern in code.

Choose names for pattern participants that are meaningful in the application context.

Define the classes.
<table>
<thead>
<tr>
<th>Purpose</th>
<th>Design Pattern</th>
<th>Aspect(s) That Can Vary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creational</td>
<td>Abstract Factory (87)</td>
<td>families of product objects</td>
</tr>
<tr>
<td></td>
<td>Builder (97)</td>
<td>how a composite object gets created</td>
</tr>
<tr>
<td></td>
<td>Factory Method (107)</td>
<td>subclass of object that is instantiated</td>
</tr>
<tr>
<td></td>
<td>Prototype (117)</td>
<td>class of object that is instantiated</td>
</tr>
<tr>
<td></td>
<td>Singleton (127)</td>
<td>the sole instance of a class</td>
</tr>
<tr>
<td>Structural</td>
<td>Adapter (139)</td>
<td>interface to an object</td>
</tr>
<tr>
<td></td>
<td>Bridge (151)</td>
<td>implementation of an object</td>
</tr>
<tr>
<td></td>
<td>Composite (163)</td>
<td>structure and composition of an object</td>
</tr>
<tr>
<td></td>
<td>Decorator (175)</td>
<td>responsibilities of an object without subclassing</td>
</tr>
<tr>
<td></td>
<td>Facade (185)</td>
<td>interface to a subsystem</td>
</tr>
<tr>
<td></td>
<td>Flyweight (195)</td>
<td>storage costs of objects</td>
</tr>
<tr>
<td></td>
<td>Proxy (207)</td>
<td>how an object is accessed; its location</td>
</tr>
<tr>
<td>Behavioral</td>
<td>Chain of Responsibility (223)</td>
<td>object that can fulfill a request</td>
</tr>
<tr>
<td></td>
<td>Command (233)</td>
<td>when and how a request is fulfilled</td>
</tr>
<tr>
<td></td>
<td>Interpreter (243)</td>
<td>grammar and interpretation of a language</td>
</tr>
<tr>
<td></td>
<td>Iterator (257)</td>
<td>how an aggregate’s elements are accessed, traversed</td>
</tr>
<tr>
<td></td>
<td>Mediator (273)</td>
<td>how and which objects interact with each other</td>
</tr>
<tr>
<td></td>
<td>Memento (283)</td>
<td>what private information is stored outside an object, and when</td>
</tr>
<tr>
<td></td>
<td>Observer (293)</td>
<td>number of objects that depend on another object; how the dependent objects stay up to date</td>
</tr>
<tr>
<td></td>
<td>State (305)</td>
<td>states of an object</td>
</tr>
<tr>
<td></td>
<td>Strategy (315)</td>
<td>an algorithm</td>
</tr>
<tr>
<td></td>
<td>Template Method (325)</td>
<td>steps of an algorithm</td>
</tr>
<tr>
<td></td>
<td>Visitor (331)</td>
<td>operations that can be applied to object(s) without changing their class(es)</td>
</tr>
</tbody>
</table>

Table 1.2: Design aspects that design patterns let you vary
Online resources

• Pattern FAQ
  ▪ http://g.oswego.edu/dl/pd-FAQ/pd-FAQ.html

• Basic patterns
  ▪ http://exciton.cs.oberlin.edu/javaresources/DesignPatterns/default.htm

• Patterns home page
  ▪ http://hillside.net/patterns/