Outline

- What are Design Patterns?
  - A design 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 the this solution a million times over, without ever doing it the same twice
- Design Patterns
  - Usefulness of design patterns
  - Design pattern notation
  - Design Pattern Categories
  - Object oriented design principles
- Patterns covered in this Lecture
  - Facade: Unifying the interface to a subsystem.
  - Adapter: Interfacing to existing systems (legacy systems)
  - Bridge: Interfacing to existing and future systems

Why Use Design Patterns

- **Reuse**: Once a design pattern has been verified, it can be used in any number of ways in a given context.
- **Common Vocabulary**: Design patterns give software designers a common vocabulary that concisely encapsulates solutions to design problems.
- **Easy to modify**: Designs patterns are easy to modify to apply to a particular problem. The solutions can also be modified to give flexibility with minimal risk.

Elements of a Pattern

- **The Pattern Name** encapsulates a well known solution to a design problem, and increases our design vocabulary
- **The Problem** describes when to apply the pattern. It gives the context of the pattern, and possibly some pre-conditions to ensure the pattern will be effective.
- **The Solution** describes the elements that make up the design, their relationships, responsibilities, and collaborations. The solution is a template, that can be modified to apply to range of situations.
- **The Consequences** are the results and trade-offs of applying the pattern. These are critical for evaluating the costs and benefits of applying a pattern.
  [Gamma et al 95]

Design Patterns Notation

- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley, 1995
- Based on OMT Notation
- Notational issues
  - Attributes come after the Operations
  - Associations are called acquaintances
  - Multiplicities are shown as solid circles
  - Inheritance shown as triangle
  - Dashed line : Instantiation Association (Class can instantiate objects of associated class. In UML it denotes a dependency)
  - UML Note is called Dogear box (connected by dashed line to class operation): Pseudo-code implementation of operation

Review: Modeling Typical Aggregations

Fixed Structure:

- Car
  - Doors
  - Wheels
  - Battery
  - Engine

Organization Chart (variable aggregate):

- University
  - School
  - Department

Dynamic tree (recursive aggregate):

- Program
  - Class

Compound Statement

Simple Statement
Ideal Structure of a Subsystem: Façade, Adapter, Bridge

- A subsystem consists of
  - an interface object
  - a set of application domain objects (entity objects) modeling real entities or existing systems
  - Some of the application domain objects are interfaces to existing systems
  - one or more control objects
- Realization of Interface Object:Facade
  - Provides the interface to the subsystem
- Interface to existing systems: Adapter or Bridge
  - Provides the interface to existing system (legacy system)
  - The existing system is not necessarily object-oriented!

Facade Pattern

- Provides a unified interface to a set of objects in a subsystem.
- A façade defines a higher-level interface that makes the subsystem easier to use (i.e. it abstracts out the gory details)
- Facades allow us to provide a closed architecture

Realization of a Closed Architecture with a Facade

- The subsystem decides exactly how it is accessed.
- No need to worry about misuse by callers
- The subsystem components can still be accessed directly.
- If a façade is used the subsystem can be used in an early integration test

Reuse

- Main goal:
  - Reuse knowledge from previous experience to current problem
  - Reuse functionality already available
- Composition (also called Black Box Reuse)
  - New functionality is obtained by aggregation
- Inheritance (also called White-box Reuse)
  - New functionality is obtained by inheritance.
- Three ways to get new functionality:
  - Implementation inheritance
  - Interface inheritance
  - Delegation

Implementation Inheritance vs Interface Inheritance

- Implementation inheritance
  - Also called class inheritance
  - Goal: Extend an applications' functionality by reusing functionality in parent class
  - Inherit from an existing class with some or all operations already implemented
- Interface inheritance
  - Also called subtyping
  - Inherit from an abstract class with all operations specified, but not yet implemented
**Implementation Inheritance**

- A very similar class is already implemented that does almost the same as the desired class implementation.
  - Example: I have a `List` class. I need a `Stack` class. How about subclassing the `Stack` class from the `List` class and providing three methods, `Push()` and `Pop()`, `Top()`?

- Problem with implementation inheritance:
  - Some of the inherited operations might exhibit unwanted behavior. What happens if the Stack user calls `Remove()` instead of `Pop()`?

**Delegation**

- Delegation is a way of making composition (for example aggregation) as powerful for reuse as inheritance.
  - In Delegation two objects are involved in handling a request.
    - A receiving object delegates operations to its delegate.
    - The developer can make sure that the receiving object does not allow the client to misuse the delegate object.

**Delegation or Inheritance?**

- Delegation
  - Pros:
    - Flexibility: Any object can be replaced at run time by another one (as long as it has the same type)
  - Cons:
    - Inefficiency: Objects are encapsulated.

- Inheritance
  - Pros:
    - Straightforward to use
      - Supported by many programming languages
      - Easy to implement new functionality
  - Cons:
    - Inheritance exposes a subclass to the details of its parent class
      - Any change in the parent class implementation forces the subclass to change (which requires recompilation of both).

**Delegation instead of Inheritance**

- Delegation: Catching an operation and sending it to another object.

**Adapter Pattern**

- “Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn’t otherwise be because of incompatible interfaces
- Used to provide a new interface to existing legacy components (Interface engineering, reengineering).
- Also known as a wrapper
- Two adapter patterns:
  - Class adapter:
    - Uses multiple inheritance to adapt one interface to another
  - Object adapter:
    - Uses single inheritance and delegation
- We will mostly use object adapters and call them simply adapters

**Delegation pattern**

- Delegation is used to bind an Adapter and an Adaptee
  - Interface inheritance is used to specify the interface of the Adapter class.
  - `Target` and `Adaptee` (usually called legacy system) pre-exist the Adapter.
  - `Target` may be realized as an interface in Java.
Adapter pattern example

```java
public class ServicesEnumeration implements Enumeration {
    public boolean hasMoreElements() {
        return this.currentServiceIdx <= adaptee.numServices();
    }
    public Object nextElement() {
        if (!this.hasMoreElements()) {
            throw new NoSuchElementException();
        }
        return adaptee.getService(this.currentServiceIdx++);
    }
}
```

Client

```java
RegisteredServices
numServices()
getService(name)
```

Adapter pattern example 2

```java
Shape
draw()
```

```java
TextShape
draw()
```

```java
Triangle
draw()
```

```java
Circle
draw()
```

```java
DrawingEditor
* (Client)
```

```java
Abstract
Shape
```

```java
Concrete
Shape
```

```java
Adapter
```

```java
Bridge Pattern – reference B&D Appendix A.3
```

```java
Design Patterns encourage good Design Practice
```

- A facade pattern should be used by all subsystems in a software system. The façade defines all the services of the subsystem.
- The façade will delegate requests to the appropriate components within the subsystem.
- Adapters should be used to interface to any existing proprietary components.
- For example, a smart card software system should provide an adapter for a particular smart card reader or other hardware that it controls and queries.
- Bridges should be used to interface to a set of objects where the full set is not completely known at analysis or design time.
- Bridges should be used when the subsystem must be extended later (extensibility).

```java
Bridge Pattern – reference B&D Appendix A.3
```

```java
More Design Patterns!
```

- Modifiable Designs
  - Why are modifiable designs important, and what makes a design modifiable.
- Structural pattern
  - Proxy – creates a stand-in for an object that we don’t want to access.
- Behavioral pattern
  - Observer – coordinates several views of a single object.
- Creational Patterns
  - Abstract Factory – initializes objects independently from the client.
Why are modifiable designs important?

A modifiable design enables…

…an iterative and incremental development cycle
  * concurrent development
  * risk management
  * flexibility to change

…to minimize the introduction of new problems when fixing old ones

…to deliver more functionality after initial delivery

What makes a design modifiable?

- Low coupling and high coherence
- Clear dependencies
- Explicit assumptions

How do design patterns help?

- They are generalized from existing systems
- They provide a shared vocabulary to designers
- They provide examples of modifiable designs
  * Abstract classes
  * Delegation