IMPORTANT NOTICE TO STUDENTS

These slides are NOT to be used as a replacement for student notes. These slides are sometimes vague and incomplete on purpose to spark class discussions.

Gang of Four (GoF)
OO Design Patterns

CS 446/646 ECE452
May 11th, 2011
Motivation

Object Oriented Analysis (OOA)

- domain problem **designed** as (domain) objects
  - addresses the **functional challenges**
  - what a system does
  - provides guidance for implementation

Object Oriented Design (OOD)

- domain problem **solved** as (implementation) objects
  - addresses the implementation challenges
  - how a system realizes OOA
Motivation

How can we improve OOD

- identify common characteristics
  - creation, structure, behaviour & interactions

- design patterns (design reuse)
  - generic blueprints (micro architecture)
  - language and implementation independent
  - two main catalogues
    - **GoF**: Gang of Four (Gamma, Helm, Johnson, Vlissides, 1995)
Motivation

What is a Design Pattern

- common solution to a reoccurring problem in design

Anatomy

- name
- problem/motivation
- solution
- consequences & tradeoffs
- which ones are important for us?
Design Patterns Classification

GoF Design Patterns

- Creational
- Structural
- Behavioral

class scope

object scope
Design Patterns Classification

“Purpose” based classification

- creational:
  - concerns with creation process of objects & classes
- structural
  - composition of classes & objects
- behavioral
  - characterizes interaction & responsibility of objects & classes
Design Patterns Classification

“Scope” based classification

- decided if the pattern applies to mainly classes or objects

Two categories

- class scope
  - relationship between classes & subclasses
  - statically defined at run-time

- object scope
  - object relationships (what type?)
  - can be manipulated at runtime (so what?)
Design Patterns Classification

Creational class
- defers object creation to sub-classes (*factory method*)

Structural class
- inheritance to compose classes (*adapter*)

Behavioral class
- uses inheritance to describe flow of control, algorithms (*template*)

Creational object
- defers object creation to other objects (*abstract factory*)

Structural object
- deals with object assembly (*adapter*)

Behavioral object
- group of objects working together to carry out a task (*iterator*)
Intent

- “ensure a class only has one instance, and provide a global point of access to it.”

Construction

```plaintext
Singleton
- singleton : Singleton
- Singleton()
+ getInstance() : Singleton
```
Singleton

Intent

• “ensure a class only has one instance, and provide a global point of access to it.”

Construction

```java
public class Singleton {
    private static final Singleton INSTANCE = new Singleton();

    // Private constructor prevents instantiation from other classes
    private Singleton() {}

    public static Singleton getInstance() {
        return INSTANCE;
    }
}
```

<table>
<thead>
<tr>
<th>Singleton</th>
</tr>
</thead>
<tbody>
<tr>
<td>- singleton : Singleton</td>
</tr>
<tr>
<td>- Singleton()</td>
</tr>
<tr>
<td>+ getInstance() : Singleton</td>
</tr>
</tbody>
</table>
Singleton

Advantages

- controlled access to the class instance(s)
  - can dictate who, and when a client can access
- refinement of functionality
  - via inheritance/subclass

```java
Worker
  - singleton: Worker
  - Worker()
  + getInstance(): Worker
  + perform()

getInstance() {
  for WINDOWS_OS singleton=WindowsWorker
  and for LINUX_OS singleton=LinuxWorker
  // normal singleton code
}
```

```java
WindowsWorker
  + perform()

LinuxWorker
  + perform()
```
Singleton

Advantages

• variable number of instances
  – the `getInstance()` method needs modification
  – *what else needs to change?*
A closer look at Singleton

- reuse
- separation of concerns
- global presence
- stateful vs. stateless
- multiple instances
- life cycle
Singleton – A Closer Look

Reuse

- coupling
  - results in tighter coupling
  - couples with the exact type of the singleton object
  - pass by reference to reduce coupling?
Singleton – A Closer Look

Reuse

- coupling
  - results in tighter coupling
  - couples with the exact type of the singleton object
  - pass by reference to reduce coupling?

```java
public void doSomething()
{
    Worker worker = Worker.getInstance();
    worker.perform();
}
```

```java
doSomething(Worker worker)
{
    worker.perform();
}
```
Singleton – A Closer Look

Reuse

• inheritance
  – easy to extend functionality in subclasses
  – not easy to override the object instance in subclasses
Singleton – A Closer Look

Separation of concerns

- singleton class responsible for creation
  - acts as a builder/factory
- what if we were to separate the two concerns
  - example
    - database connection as a singleton
    - system 1 uses a singleton to ensure only a single database connection
    - system 2 needs to connection pool of 10 databases connections
Singleton – A Closer Look

Global presence

- provides a global access point to a service
  - aren't global variables bad?
  - can be accessed from anywhere
    - violation of layered access
- not part of method signature
  - dependency is not obvious
  - requires code inspection
- a large system may require many singletons
  - use a registry/repository
Singleton – A Closer Look

Stateful singleton

- same as a global variable in principle
  - aren't global variables bad?
- access concerns
  - synchronization
  - concurrency – multiple threaded using a singleton
- mutable vs. immutable state

Stateless singleton

- better then stateful
- can we have a stateless singleton?
Singleton – A Closer Look

Multiple instances

• distributed systems
  – is it possible to have a true singleton in a distributed system?
  – global registries/repositories

• language (Java) specific concerns
  – initialization – has to be thread safe
  – serialization
  – class loaders
Singleton – A Closer Look

Life-cycle & life span

- creation
  - lazy initialization
- singletons are long lived
  - as long as an application's life span
  - registries can outlive applications
  - unit testing requires short lived state
- language (Java) specific concern
  - reloading singleton class (servlets)
  - loss of state
Singleton

When can I use a singleton

- **considerations**[1]
  - will every user use this class exactly the same way?
  - will every applications ever need only one instance?
  - should the clients be unaware of the application

- **examples**
  - Java Math class (stateless – static class)
  - top level GUI (window/frame)
  - logging

Adapter

Intent

- “convert the interface of a class into another interface... Adapter lets classes work together that couldn't otherwise because of incompatible interface”

- also known as “wrapper”
- boolean values can be represented by
  - \{1,0\}, \{true, false\}, \{yes, no\}
  - does this qualify as an adapter?
Motivation

Need to add “Text” capability to our drawing editor.

Consider an off the shelf **TextView** component
Motivation

Observations

• can be done in two ways
  – object composition (shown above)
  – inheritance
    • Shape provides “interface” and TextView provides an implementation
    • Lets try to draw this?
Requirement

- requires multiple inheritance

what about implementations that do not support multiple inheritance (Java)?
Adapter – Object

Requirement

- **via object composition**

```
methodA(){
    ...
    adaptee.method1() //main implementation
    ...
}
```
Adapter – Class vs. Object

**Class**

- commitment to a **concrete** adaptee class
  - not to its subclasses (class hierarchy)
- allows for specialization
  - how?
- static in nature

**Object**

- can use many adaptees
  - including sub-classes
- harder to override the adaptee behavior
  - why?
Adapter – Class vs. Object

Client
- target: target
  perform()

Target
  + methodA()
  + method1()

Adaptee
  + method1()

Adaptor
- adaptee: Adaptee
  + methodA()

methodA()
  ...
  adaptee.method1() // main implementation
  ...

perform()
  ...
  target.methodA()
  ...

«interface»

«implementation»
Adapter & Dependency Inversion

Dependency Inversion (DI)

- decouple high level layer from lower level layer(s)
Dependency Inversion Example

Implications

- Button implementation relies on Lamp
- Any changes to Lamp will impact Button
- What if we want to reuse Button class with a different component
  - Such as a motor
Dependency Inversion Example

Dependency Inversion to Rescue

- looks good (?)
- still a dependency left
Dependency Inversion Example

Observation

- adapter enhanced the design
  - increased re-usability at the price of complexity
Adapter

How much adaptation is reasonable?
Bridge

Intent

- “decouples an abstraction from its implementation so the two can vary independently”

- does this not sounds like an adapter?
  - will take a closer look later
Bridge

Abstraction
- impl : Implementor
+ function()

Implementor
+ implementation()

RefinedAbstraction
+ refinedFunction()

ConcreteImplementor
+ implementation()
Bridge

Abstraction
- impl: Implementor
+ function()

Implementor
+ implementation()

RefinedAbstraction
+ refinedFunction()

ConcreteImplementor
+ implementation()
**Problem**

**Problem 1**: what if we have to support another platform?

**Problem 2**: client code is tied to an implementation. For portable code, the client should not refer to an implementation.
Bridge Example

Solution: Use bridge pattern to place abstraction and implementation in two different hierarchies.
Bridge Example

Solution: Use bridge pattern to place **abstraction** and **implementation** in two **different hierarchies**.
Bridge

Features

- flexible binding between abstraction & implementation
- two class hierarchies
- clients are decoupled
Adapter & Bridge

Common Elements

- flexibility via indirection
- request forwarding
Adapter & Bridge

Difference in intent

• adapter
  – resolves incompatibilities between two existing interfaces
  – two interfaces are independent and can evolve separately
  – coupling is unforeseen
  – adapts components after they have been designed

• bridge
  – connects an abstraction and its many implementations
  – evolution is in accordance with the base abstraction
  – coupling between the abstraction and the implementations are known