CS 246	Software Design and Development
Clearness	
Design Patterns	*Exploration*

Goal

The goal of this exploration is for you as a team to research and give a clear presentation on a specific design pattern.

Requirements

Your task is to research an assigned Design Pattern and prepare a 15-minute presentation, for which Brother Neff will provide a model. In your presentation, explain the use of this specific pattern in the abstract, and also concretely in code that you will have written in conjunction with your Boldness (Threads of Glory) exploration. (This should be self-contained sample code demonstrating the pattern in actual use.) Budget up to 5 minutes for questions and answers, but these don't have to wait until the end of your presentation. The 15-minute maximum **MUST** be held to.

Assignments

Design Pattern	Researchers / Presenters
Adapter	Larson, Sam Clyde and Haru
Command	Travis, Brady and Jose
Composite	Dustin, Davis and Andrew
Decorator	Alex, McKay and Gabi
Model-View-Controller	Jordan Balls, Nick and David Mayfield
Null Object	Emmanuel, Brenton and Devin
Observer + Publisher-Subscriber	David Cheney, Jordan Olive and Tekara
Prototype	Sam Graham, Jason and Schuyler
Template Method	Jeff, Mackenzie and Kevin

Grading Criteria

The following grading rubric is meant to guide you to produce a presentation of exceptional quality.

	Exceptional **100%**	**Good** **90%**	**Acceptable** **70%**	**Developing** **50%**	**Missing** **0%**
Knowledge **40%**	I am convinced all members of the team are experts.	All questions answered correctly and without hesitation.	The presentation completely covers the topic, but a question was not answered correctly.	The presentation does not completely cover the topic, **or** more than one question was not answered correctly.	No grasp of information; no one can answer any question.
Slides **30%**	It is difficult to imagine how the slide deck could be improved.	Information on the slides greatly contributes to the audience's understanding.	All the required information is displayed on the slides.	One of the following: • Illogical order • Typos • Any slide is difficult to understand. • Deck does not contribute to the presentation.	Missing deck, or presentation would be better without it.
Presentation **30%**	I was spellbound. At no point did my mind wander. Audience participation was well planned and executed.	No hesitation, eye contact with the audience, speech easy to follow.	Some hesitation, eye contact most of the time, speech sometimes difficult to follow.	One of the following: • unclear speech (mumbling, slurring) • eye contact only seldom • frequent hesitation.	Read entire presentation from slides with no eye contact and with halting, monotone or timid (too quiet) speech.

Resources

- David Geary calls Design Patterns "best practices" of experienced object-oriented software developers, captured in some helpfully descriptive form. See http://www.javaworld.com/javaworld/jw-10-2001/jw-1012-designpatterns.html
- Building architect Christopher Alexander originally developed the concept of patterns in his book *The Timeless Way of Building* first published in 1979.
- A pattern is a design solution to a problem within the context of other problems.
- Problems are (usually) interdependent.
- This means that a single problem cannot be solved in isolation but rather must be solved in the context of other competing forces or obstacles.
- Thus, a pattern is a solution that strikes some balance between competing forces.
- The better we understand the problem, the better our chances of discovering or creating a solution.
- Alexander argues that if we find the right decomposition of the problem, then the form of the solution will reveal itself and the final solution is the synthesis of solutions to generate a form.
- Decomposing the problem is not only a matter of finding the problem constituents but also understanding their interdependencies.
- A pattern is *more* than a solution to a set of problems in a context.
- What makes a solution a pattern is the fact that it is *recurring*.
- If a design solution in its essential distilled form is found in multiple systems and is a sucessful design, then this solution is a pattern.
- The GoF (Gang of Four) is Gamma, Helm, Johnson and Vlissides. Their 1995 book *Design Patterns: Elements of Reusable Object-Oriented Software* is considered a classic.
- For a familiar example – Singleton
  - Problem: Provide global access to some service (Class)
  - Constraint: Only one instance of this service should be available.
  - Threading can be a problem (references from multiple threads to the same Singleton), but there are ways to alleviate that.
- Other Design Patterns (not all-inclusive):
  - Model-View-Controller
  - Template Method
  - Abstract Factory
  - Observer + Publisher-Subscriber
  - Mediator
  - Visitor
  - Composite
  - Decorator
  - Bridge
  - Adapter
  - Command

A Detailed Look at a Design Pattern

What follows is taken from Chapter 1 of *Head First Design Patterns* by Eric Freeman and Elisabeth Freeman, published by O'Reilly.

It all started with a simple **SimUDuck** application.

You work for a company that makes a highly successful duck pond simulation game, **SimUDuck**. The game can show a large variety of duck species swimming and making quacking sounds. The initial designers of the system used standard OO techniques and created one Duck superclass from which all other duck types inherit.

```
class Duck
{
   void quack() {}
   void swim() {}
   void display() {}
   // other duck-like methods
}

class MallardDuck extends Duck
{
   void display()
   { // looks like a mallard
   }
}

class RedheadDuck extends Duck
{
   void display()
   { // looks like a redhead
   }
}
```

Lots of other types of ducks inherit from the Duck class.

But now we need the ducks to FLY!

Management decides that flying ducks is just what the simulator needs to blow away the competition. You are given the task. You decide to show your true OO genius and simply add a fly method in the Duck class so that all the ducks will inherit it.

But something goes horribly wrong!

You failed to notice that not all Duck subclasses should be able to fly! When you added new behavior to the Duck superclass, you were also (inadvertantly) adding behavior that was *not* appropriate for some Duck subclasses (RubberDuck). You now have flying inanimate objects in the SimUDuck program.

A **LOCALIZED** UPDATE TO THE CODE CAUSED A **NON-LOCAL** SIDE EFFECT (flying rubber ducks)!

You think about inheritance. . .

"I could always just override the fly() method in RubberDuck to do nothing, but then what happens when we add wooden decoy ducks to the program? They aren't supposed to fly *or* quack. . ."

```
class RubberDuck
{
   void quack()
   {
      // overridden to squeak
   }
   void display()
   { // looks like a rubber duck
   }
   void fly()
   {
      // overridden to do nothing
   }
}

class DecoyDuck
{
   void quack()
   {
      // overridden to do nothing
   }
   void display()
   { // looks like a decoy duck
   }
   void fly()
   {
      // overridden to do nothing
   }
}
```

How about an interface?

You realize that inheritance probably isn't the answer, because the spec will just keep changing and you'll be forced to look at and possibly override fly() and quack() for every new Duck subclass that's ever added to the program. . . *forever*.

So, you need a cleaner way to have only *some* (but not *all*) of the duck types fly or quack. What about THIS design?

```
class Duck
{
   void swim() {}
   void display() {}
   // other duck-like methods
}

interface Flyable { void fly(); }
interface Quackable { void quack(); }

class MallardDuck extends Duck implements Flyable, Quackable ...
class RedheadDuck extends Duck implements Flyable, Quackable ...
class RubberDuck extends Duck implements Quackable ...
class DecoyDuck extends Duck ...
```

Can you say DUPLICATE CODE?! If you thought having to override a few methods was bad, how are you gonna feel when you need to make a little change to the flying behavior. . .in all 48 of the flying Duck subclasses?!

Not *all* of the subclasses should have flying or quacking behavior, so inheritance isn't the right answer. But while having the subclasses implement Flyable and/or Quackable solves *part* of the problem (no inappropriately flying rubber ducks), it completely destroys code reuse for those behaviors, so it just creates a *different* maintenance nightmare. And of course there might be more than one kind of flying behavior even among the ducks that *do* fly. . .

Design Principle: Identify (and encapsulate) the aspects of your application that vary and separate them from what stays the same.

Later you can alter or extend the parts that vary WITHOUT affecting those that don't. This means fewer unintended consequences and more flexibility in your applications.

As simple as this concept is, it forms the basis for almost every design pattern. All patterns provide a way to let **SOME PART OF A SYSTEM VARY INDEPENDENTLY OF ALL OTHER PARTS**.

How to separate what changes from what stays the same?

We know that fly() and quack() are the parts of the Duck class that vary across ducks.

To separate these behaviors from the Duck class, just pull both methods out of the Duck class and create a new set of classes to represent each behavior.

From now on, the Duck behaviors will live in a separate class—a class that implements a particular behavior interface. That way, the Duck classes won't need to know any of the implementation details for their own behaviors.

Design Principle: Program to an interface (a supertype), not an implementation.

```
interface FlyBehavior { public void fly(); }

class FlyWithWings implements FlyBehavior
{
   public void fly()
   { // implements duck flying
   }
}

class FlyNoWay implements FlyBehavior
{
   public void fly()
   { // do nothing -- can't fly!
   }
}
```

Making a set of classes whose entire reason for living is to represent some behavior (and that actually implements the behavior interface) is in contrast to the way you were doing things before, where a behavior either came from a concrete implementation in the superclass Duck, or by providing a specialized implementation—and there was no room for changing out the behavior (other than writing more code).

With the new design, the Duck subclass will use a behavior represented by an *interface* (FlyBehavior and QuackBehavior), so that the actual *implementation* of the behavior won't be "locked into" the Duck subclass.

What "program to an interface, not an implementation" means is that the declared type of the variables should be a supertype, usually an abstract class or interface, so that the objects assigned to those variables can be of

any concrete implementation of the supertype, which means the class declaring them doesn't have to know about the actual object types!

With this design, other types of objects can reuse our fly and quack behaviors because these behaviors are no longer hidden away in our Duck classes! And we can add new behaviors without modifying any of our existing behavior classes or touching any of the Duck classes that *use* flying behaviors.

So we get the benefit of REUSE without all the baggage that comes along with inheritance.

Now, you're probably saying: It feels a little weird to have a class that's just a behavior. Aren't classes supposed to represent *things*? Aren't classes supposed to have both state AND behavior?

In an OO system, yes, classes represent things that generally have both state (instance variables) and methods. And in this case, the *thing* happens to be a behavior. But even a behavior can still have state and methods; a flying behavior might have instance variables representing the attributes for the flying (wing beats per minute, max altitude and speed, etc.) behavior.

Integrating the Duck Behavior

The key is that a Duck will now *delegate* its flying and quacking behavior, instead of using quacking and flying methods defined in the Duck class (or subclass).

Take a moment and think about how you would implement a duck so that its behavior could change at runtime!

```
public abstract class Duck
{
   FlyBehavior mFlyBehavior;
   QuackBehavior mQuackBehavior;

   public Duck() {}
   public abstract void display();

   public void doFly() { mFlyBehavior.fly(); }
   public void doQuack() { mQuackBehavior.quack(); }

   public void swim() { System.out.println("All ducks float, even decoys!"); }
}

interface FlyBehavior { public void fly(); }
interface QuackBehavior { public void quack(); }

class FlyWithWings implements FlyBehavior
{
   public void fly()
   {
      System.out.println("I'm flying!!");
   }
}

class FlyNoWay implements FlyBehavior
{
   public void fly()
   {
      System.out.println("I can't fly.");
   }
}
```

```
class Quack implements QuackBehavior
{
   public void quack()
   {
      System.out.println("Quack");
   }
}

class MuteQuack implements QuackBehavior
{
   public void quack()
   {
      System.out.println("<< Silence >>");
   }
}

class Squeak implements QuackBehavior
{
   public void quack()
   {
      System.out.println("Squeak");
   }
}

public class MiniDuckSimulator
{
   public static void main(String[] args)
   {
      Duck mallard = new MallardDuck();
      mallard.doQuack();
      mallard.doFly();
   }
}
```

How do we change the MallardDuck's behavior "on the fly"?!

```
public void setFlyBehavior(FlyBehavior fb)
{
   mFlyBehavior = fb;
}
public void setQuackBehavior(QuackBehavior qb)
{
   mQuackBehavior = qb;
}
```

```
class ModelDuck extends Duck
{
   ModelDuck()
   {
      mFlyBehavior = new FlyNoWay(); // grounded!
      mQuackBehavior = new Quack();
   }

   public void display()
   {
      System.out.println("I'm a model duck.");
   }
}
```

Make a new FlyBehavior type:

```
class FlyRocketPowered implements FlyBehavior
{
   public void fly() { System.out.println("I'm flying with a rocket!"); }
}

public class MiniDuckSimulator
{
   public static void main(String[] args)
   {
      Duck mallard = new MallardDuck();
      mallard.doQuack();
      mallard.doFly();

      Duck model = new ModelDuck();
      model.doFly();
      model.setFlyBehavior(new FlyRocketPowered());
      model.doFly();
   }
}
```

Instead of thinking of the duck behaviors as a *set of behaviors*, start thinking of them as a *family of algorithms*. In the SimUDuck's design, the algorithms represent things a duck would do (different ways of quacking or flying), but you could just as easily use the same techniques for a set of classes that implement the ways to compute state sales tax by different states.

HAS-A can be better than IS-A

Each duck has a FlyBehavior and a QuackBehavior to which it delegates flying and quacking. When you put classes together like this you're using *composition*. Instead of *inheriting* their behavior, the ducks get their behavior by being *composed* with the right behavior objects.

Design Principle: Favor composition over inheritance.

You've seen that creating systems using composition gives you a lot more flexibility. Not only does it let you encapsulate a family of algorithms into their own set of classes, but it also lets you *change behavior at runtime* (as long as the object you're composing with implements the correct behavior interface).

Composition is used in MANY design patterns.

What this example has demonstrated is called the **Strategy Pattern**. Thanks to this pattern, the simulator is ready for any changes those execs might cook up on their next business trip to Vegas.

Here's the formal definition:

> The **Strategy Pattern** defines a family of algorithms, encapsulates each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

The Skeptical Developer!

Developer: Okay, hmm, but isn't this all just good object-oriented design; I mean as long as I follow encapsulation and I know about abstraction, inheritance, and polymorphism, do I really need to think about Design Patterns? Isn't it pretty straightforward? Isn't this why I took all those OO courses? I think Design Patterns are useful for people who don't know good OO design.

Patterns Guru: Ah, this is one of the true misunderstandings of object-oriented development: that by knowing the OO basics we are automatically going to be good at building flexible, reusable, and maintainable systems.

Developer: No?

Guru: No. As it turns out, constructing OO systems that have these properties is not always obvious and has been discovered only through hard work.

Developer: I think I'm starting to get it. These, sometimes non-obvious, ways of constructing object-oriented systems have been collected. . .

Guru: . . .yes, into a set of patterns called Design Patterns.

Developer: So, by knowing patterns, I can skip the hard work and jump straight to designs that always work?

Guru: Yes, to an extent, but remember, design is an art. There will always be tradeoffs. But, if you follow well thought-out and time-tested design patterns, you'll be way ahead.

Developer: What do I do if I can't find a pattern?

Guru: There are some object oriented principles that underlie the patterns, and knowing these will help you to cope when you can't find a pattern that matches your problem.

Developer: Principles? You mean beyond abstraction, encapsulation, and. . .

Guru: Yes, one of the secrets to creating maintainable OO systems is thinking about how they might change in the future and these principles address those issues.

The Power of a Shared Pattern Vocabulary

. . . overheard at the local diner. . .

Alice: I need a cream cheese with jelly on white bread, a chocolate soda with vanilla ice cream, a grilled cheese sandwich with bacon, a tuna fish salad on toast, a banana split with ice cream & sliced bananas and a coffee with cream and two sugars, . . . oh, and put a hamburger on the grill!

Flo: Give me a C.J. White, a black & white, a Jack Benny, a radio, a house boat, a coffee regular and burn one!

What's the difference between these two orders? Not a thing! They're both the same order, except Alice is using twice the number of words and trying the patience of a grumpy short order cook.

What's Flo got that Alice doesn't? A SHARED VOCABULARY with the short order cook. Not only is it easier to communicate with the cook, but it gives the cook less to remember because he's got all the diner patterns in his head.

When you communicate using patterns you are doing more than just sharing JARGON.

Shared pattern vocabularies are POWERFUL!

When you communicate with another developer or your team using patterns, you are communicating not just a pattern name but a whole set of qualities, characteristics and constraints that the pattern represents.

Patterns allow you to say more with less.

When you use a pattern in a description, other developers quickly know precisely the design you have in mind.

Talking at the pattern level allows you to stay "in the design" longer.

Talking about software systems using patterns allows you to keep the discussion at the design level, without having to dive down to the nitty gritty details of implementing objects and classes.

Shared vocabularies can turbocharge your development team.

A team well versed in design patterns can move more quickly with less room for misunderstanding.

Shared vocabularies encourage more junior developers to get up to speed.

Junior developers look up to experienced developers. When senior developers make use of design patterns, junior developers also become motivated to learn them. Build a community of pattern users and your organization will benefit!

In Summary

- Knowing the OO basics does not make you a good OO designer. These are just tools in your toolbox:
  - OO Basics
    - Abstraction
    - Encapsulation
    - Inheritance
    - Polymorphism
  - OO Principles
    - Encapsulate what varies.
    - Favor composition over inheritance.
    - Program to interfaces, not implementations.
  - OO Patterns
    - Strategy—defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from the clients that use it.
    - Many, many more. . .
- Good OO designs are reusable, extensible and maintainable. The testing framework **junit** is an example that is *thick* with Design Patterns.
- Patterns show you how to build systems with good OO design qualities.
- Patterns are proven object-oriented experience.
- Patterns don't give you code, they give you general solutions to design problems. You apply them to your specific application.
- Patterns aren't *invented*, they are *discovered*.
- Most patterns and principles address issues of *change* in software.
- Most patterns allow some part of a system to vary independently of all other parts.
- We often try to take what varies in a system and encapsulate it.
- Patterns provide a shared language that can maximize the value of your communication with other developers.

A Bit of Academese

What ***is*** a Design Pattern?

To put it simply, a pattern is a solution to a problem in a context, or more precisely, a description of a solution to a problem in a context.

The patterns "movement" was inspired by the work of Christopher Alexander, an architect with keen insights into the essential nature of building and urban planning. Alexander's patterns capture the essentials, not the incidentals. His *Light on Two Sides of Every Room* pattern, for example, says nothing about size, shape or precise placement of windows in a room. Presumably these details could be addressed by other patterns, but the above pattern expresses a concise and unpretentious invariant, and implies that a room is somehow incomplete or inferior without light coming in on two sides. Clearly, building buildings is different from building software, but there are points of similarity.

Patterns should facilitate the distillation, transmission and assimilation of design information. With carefully crafted patterns, an expert designer can communicate his or her knowledge in such a way that an inexpert designer can take full advantage of it, without having to rediscover it all through his or her own experience. Good pattern writers can present practical knowledge in a form that will "stick" in the reader's mind. The goal is to expedite the "expertizing" of inexperienced, inexpert designers, optimally stimulating and increasing both their confidence and their competence.

Overarching principles are usually discovered in retrospect, rarely in advance of design effort. Design as a creative activity is both retrospective and prospective, involving both analysis and synthesis, but perceiving and capturing "how to" design knowledge is almost always retrospective.

Design patterns should channel, but not otherwise constrain creativity. In the typical designer's mind, creative *leaps* occur occasionally, but creative *loops* are the norm. That is, experimentation, iteration and feedback (trial and error) are essential to creation, while unpredictable inspiration is accidental, albeit beneficial.

Patterns present acquired wisdom not in formal, mathematical language, but in well-structured prose. Here is a common structure:

Name: a short name and mental "hook" for the pattern. Alternate names for the same pattern may be given here.

Context: a digest of the situation, circumstances, or specific context where the pattern is applicable, noting any special endpoints or boundary conditions.

Problem: a brief statement of the problem the pattern addresses, often conveniently phrased as a question.

Forces: a list of the opposing forces, or tradeoffs that must be resolved.

Solution: a description of how to construct and fine-tune a solution, including at least one concrete example.

Result: a new context where other patterns may be applicable.