DESIGN PATTERNS I

Mary Eberlein
EE461L

Figures from Head First Design Patterns

Design Pattern, Defined

- “A solution to a problem in a context.”
- “A language for communication solutions with others.”

- Pattern languages exist for many problems, but we focus on design
- Best known: “Gang of Four” (Gamma, Helm, Johnson, Vlissides)
  - *Design Patterns: Elements of Reusable Object-Oriented Software*

Caveats

- Design patterns are not a substitute for thought
- Class names and directory structures do not equal good design
- Design patterns have tradeoffs
- Design patterns depend on the programming language
  - Certain language restrictions may necessitate certain patterns (e.g., patterns related to object creation and destruction)

An Early Pattern

EE 312: fencepost problem

- Print n numbers separated by commas
  - Only need n-1 commas
- Similar: building a fence with wire separated by posts

Common approach:

```
for(length of fence) {
    place a post
    place some wire
}
```

An Early Pattern

- That first solution was flawed
- In EE 312: you learned a general solution – the fencepost loop

Solution:

```
Place a post
for(length of fence - 1) {
   place some wire
   place a post
}
```

Motivation for Design Patterns

- They provide an abstraction of the design experience
  - Can often serve as a reusable base of experience
- They provide a common vocabulary for discussing complete system designs
- They reduce system complexity by naming abstractions
  - Thereby increasing program comprehension and reducing learning time with a new piece of code
- They provide a target for the reorganization or refactoring of class hierarchies

Pieces of a Design Pattern

- Pattern name and classification
- Intent
- Also known as
- Justification
- Applicability
- Structure
- Participants
- Collaborations
- Consequences
- Implementation
- Sample code
- Known uses
- Related patterns

THAT FIRST PATTERN

SimUDuck

The ducks should be able to swim!

We should have Mallard ducks!

The ducks should quack!

What about a redheaded duck?

Don't ducks fly, too?

https://commons.wikimedia.org/wiki/File:Redhead_duck_(Aythya_americana,_male).jpg

Liskov Substitution Principle

Let q(x) be a property provable about objects x of type T. Then q(y) should be provable for objects y of type S where S is a subtype of T.

SimUDuck, Take 2

Let's use an interface!

I need to control the quacking volume. Make the quack method take a volume.

Design Attempts

1. Inheritance: Didn't work well. Behavior varied across subclasses. Inappropriate for some subclasses to have inherited behaviors.
2. Interfaces: Not great either. No implementation → no reuse. Implement same methods repeatedly.

What next?

Design Principles

- **Identify the aspects of your application that vary and separate them from what stays the same.**
  - Encapsulate what varies
  - Program to an interface, not to an implementation
  - Favor composition over inheritance
- For our example:
  - Pull the duck *behavior* out of the duck *class*

Encapsulate what varies

… or in UML

Fly
Behavior
Has-A
Quack
Behavior
Has-A
Duck
extends
extends
extends
MallardDuck
PekinDuck
RubberDuck

Program to an interface

Extension and Reuse

- Ducks ***delegate*** the flying and quacking behaviors

- Now, other classes can use our quacking and flying behaviors since they're not specific to ducks
  - Who would want to do that?
- We can easily add new quacking and flying styles without impacting our ducks!

The Big Picture

And what does that really look like?

- Now it’s time for some code…

What's a Duck?

```
public abstract class Duck {
  QuackBehavior quackBehavior;
  // more

  public void performQuack() {
    quackBehavior.quack();
  }
}
```

A duck has a reference to something that implements the QuackBehavior interface

Instead of quacking all on its own, a Duck delegates that behavior to the quackBehavior object

It doesn't matter what **kind** of Duck it is; all that matters is a Duck knows how to quack

What's a MallardDuck?

- Exercise: Write the MallardDuck class now...

How do we make Ducks quack and fly?

```
public class MallardDuck extends Duck {
  public MallardDuck() {
    quackBehavior = new Quack();
    flyBehavior = new FlyWithWings();
  }
  public void display() {
    System.out.println("I'm a real Mallard duck!");
  }
}
```

This is not **quite** right yet because we're still programming to the implementation (i.e., we have to know about the specific Quack behavior and FlyWithWings behavior). We can fix this with another pattern … later …

But how do Ducks *learn* to quack and fly?

- How could you teach a Duck a new way to quack or a new way to fly?

- Do you know? Try to figure it out, then look at the sample code.

- Play with `MiniDuckSimulator` and `MiniDuckSimulator1`
  - https://github.com/bethrobson/Head-First-Design-Patterns/tree/master/src/headfirst/designpatterns/strategy

Favor Composition over Inheritance

- Stated another way… “has-a is better than is-a”

- Duck’s **have** quacking behaviors and flying behaviors
  - Instead of **being** Quackable and Flyable

- Composition is good because:
  - It allows you to encapsulate a family of algorithms into a set of classes (the **Strategy** pattern)
    - The what? Yup, that was your “first” pattern…
  - It allows you to easily change the behavior at **runtime**

An Exercise

- (This should be a quick one…)

- How would you make a duck call device without inheriting from the `Duck` class?

The Strategy Pattern

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

How to Use Design Patterns

QUESTIONS?