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
  - It does not completely remove complexity in interactions but just provides a structure for centralizing it.
- Design patterns depend on the programming language
  - Certain language restrictions may necessitate certain patterns (e.g., patterns related to object creation and destruction)

设计模式是有取舍的

设计模式依赖于编程语言.

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

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*

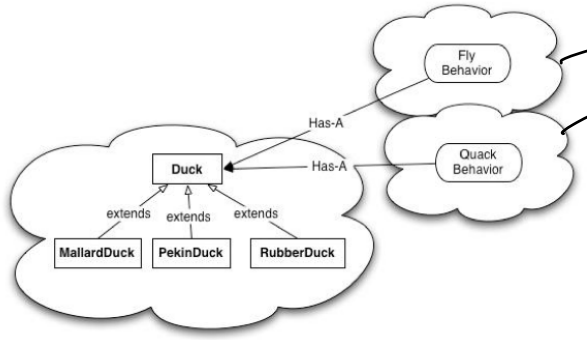
封装
面向接口编程
基于继承的组合

Liskov Subscitution 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.

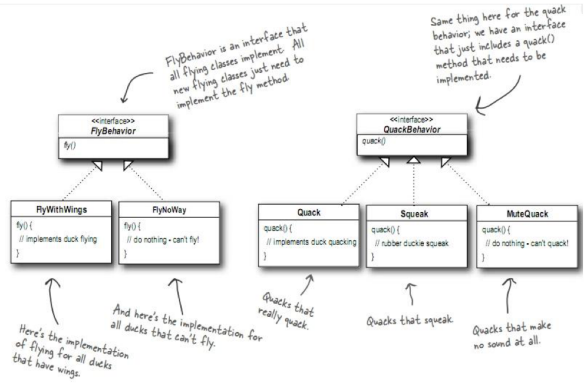
令 $x \in T$ 且有 $q(x)$ 可证，则当 $S \subseteq T$，$y \in S$ 时，$q(y)$ 可证.

Encapsulate what varies



将易于变化的部分取出

Program to an interface



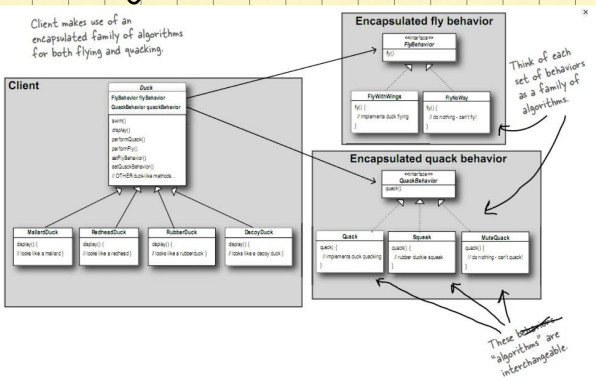
针对接口编程，而非实现

Extension and Reuse

将两个行为委托别人处理

- 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



What's a Duck?

```
public 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

```
public interface QuackBehavior {
  public void quack();
}

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

```
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!");
  }
}
```

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

How do Ducks learn to quack and fly?

```
public void setFlyBehavior (FlyBehavior fb) {
  flyBehavior = fb;
}

public void setQuackBehavior (QuackBehavior qb) {
  quackBehavior = qb;
}
```

```
public class ModelDuck extends Duck {
  public ModelDuck() {
    flyBehavior = new FlyNoWay();
    quackBehavior = new Quack();
  }

  public void display() {
    system.out.println("I am a model duck.");
  }
}
```

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

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

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

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**

组合好于继承（即使用接口会更好）
因此多用组合，少用继承

方便在运行时改变行为

The Strategy Pattern 策略模式

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.

定义了算法族

```java
public interface Strategy {
   public int doOperation(int num1, int num2);
}

public class OperationAdd implements Strategy{
   @Override
   public int doOperation(int num1, int num2) {
      return num1 + num2;
   }
}
```

```java
public class OperationSubstract implements Strategy{
   @Override
   public int doOperation(int num1, int num2) {
      return num1 - num2;
   }
}

public class OperationMultiply implements Strategy{
   @Override
   public int doOperation(int num1, int num2) {
      return num1 * num2;
   }
}
```

```java
public class StrategyPatternDemo {
   public static void main(String[] args) {
      Context context = new Context(new OperationAdd());
      System.out.println("10 + 5 = " + context.executeStrategy(10, 5));

      context = new Context(new OperationSubstract());
      System.out.println("10 - 5 = " + context.executeStrategy(10, 5));

      context = new Context(new OperationMultiply());
      System.out.println("10 * 5 = " + context.executeStrategy(10, 5));
   }
}
```

Context.java

```java
public class Context{
   public Strategy strategy;
   public Context(Strategy str){
      this.strategy = str;
   }
   public int executeStrategy(int num1, int num2){
      return this.strategy.doOperation(num1, num2);
   }
}
```