

# **CS 417**

## **Design Patterns**

**Design patterns:**  
**Designing generic components**

Dr. Chad Williams  
Central Connecticut State University

# Generic components

- Also known as reusable components
- Extended or reused in many different contexts without having to change the code
- Two basic techniques
  - Refactoring
  - Generalization

# Mechanisms

- To build generic components use
  - Inheritance
  - Delegation
- Abstract classes and interfaces also play key role

# Refactoring

- Identify recurring code - same logic in multiple different places
- Capture the logic in a single place on a generic component
- Restructure program so all existing code uses generic implementation (ie. Remove duplication)

# Why refactor

- Faster implementation of new components that use the method
- Bug fixes must be made in all duplicate methods making it easy to fix in some places but not all
- Duplicate code can lead to drifting of implementations making understanding of code more difficult even though same task done
- **Be careful not all code that looks alike is alike!**

# Function/method refactoring

```
class Computation{  
    void method1() {  
        computeStep1();  
        computeStep1();  
        computeStep1();  
    }  
    void method2() {  
        computeStep1();  
        computeStep1();  
        computeStep1();  
    }  
    ...  
}
```

```
class Computation{  
    void computeAll() {  
        computeStep1();  
        computeStep2();  
        computeStep3();  
    }  
    void method1() {  
        computeAll();  
    }  
    void method2() {  
        computeAll();  
    }  
    ...  
}
```

# Method refactoring limitations

- Method refactoring is the easiest but has several limitations
  - Only effective when common code is contained in a single method, can become much more difficult when spread across multiple functions
  - Only possible when duplicate code is all within the same class

# Refactoring across classes

- Commonalities in code across classes has several complications
  - Similar/duplicate code may refer to variables that are not accessible by other duplicate code
  - Strongly tying code together due to similarity of a function alone violates encapsulation
- Solution → refactoring by inheritance or delegation



# Refactoring by inheritance

- If duplicate code appears across two different classes may be able to refactor by inheritance
- Common code pulled into either shared parent class or new parent class created
- Any variables used also pulled to parent class

# inheritance refactoring

```
class Computation1 {  
    void method1 () {  
        computeStep1 ();  
        computeStep2 ();  
        computeStep3 ();  
    }  
}  
  
class Computation2 {  
    void method2 () {  
        computeStep1 ();  
        computeStep2 ();  
        computeStep3 ();  
    }  
}
```

```
class Computation {  
    void computeAll () {  
        computeStep1 ();  
        computeStep2 ();  
        computeStep3 ();  
    }  
}  
class Computation1 extends  
    Computation {  
    void method1 () {  
        computeAll ();  
    }  
}  
class Computation2 extends  
    Computation {  
    void method2 () {  
        computeAll ();  
    }  
    ...  
}
```

# Refactoring by inheritance limitations

- Creates problems if the two classes are not related enough to justify inheriting from same class
- Due to limitation of single inheritance, if either class has a different parent its not possible to move code to parent

# Refactoring by delegation

- Duplicate code pulled to different class and implementation of the common code is *delegated* to this class
- Requires original classes to have a reference to the delegated class
- Any non-visible variables need to be passed into helper class

# delegation refactoring

```
class Computation1{
    void method1() {
        computeStep1();
        computeStep2();
        computeStep3();
    }
}

class Computation2{
    void method2() {
        computeStep1();
        computeStep2();
        computeStep3();
    }
}

...

class Helper{
    void computeAll(int length){
        computeStep1(length);
        computeStep2();
        computeStep3();
    }
}

class Computation1{
    Helper myHelper = new Helper();
    void method1(){
        myHelper.computeAll(this.length);
    }
}

class Computation2 {
    Helper myHelper = new Helper();
    void method2(){
        myHelper.computeAll(this.length);
    }
}

...
```

# Refactoring decisions

- Refactoring by delegation can always be used and has the advantage of not putting inheritance restrictions (ie. Avoid problem with single inheritance limitation)
- Refactoring by delegation is most complex and most difficult to understand through reading the code. Therefore its often preferable to use refactoring by inheritance

# Maximizing extensibility

- When code is refactored the goal is to better the chance the code can be reused
- One way this is done is making the code extensible
  - Allow calling component to alter context the code runs in
  - Extend functionality through making changes to parameters of execution

# Extensibility

- Changing the parameters of execution
  - Allow calling component to set context of super class or helper class through accessors
    - Ex. `setNumberOfSidesShape`
  - Allow calling component to pass in parameters of execution
    - Ex. `drawShape(color)`



# Inheritance complications

- Complications arise when duplicate code but different detailed implementations

- Ex.

- ```
public class Computation1
// this portion of Computation1 and
// Computation 2 are identical
void compute() {
    computeA();
    computeB(); // implementation differs
    computeC();
}
```

# Extensibility of methods

- Duplicate code may make same function calls but be implemented differently
- Extensibility can also be applied to methods
  - Option 1: Provide a default implementation of method in super class, which can then be overridden
  - Option 2: If no default implementation make super class abstract and the method in question

# Extensible methods ex.

```
public abstract class Shape{
    public void computeAll(){
        calculatePerimeter();
        calculateArea();
    }
    public void calculatePerimeter() {
        // add all sides
    }
    abstract public void calculateArea();
}
public class square extends Shape{
    public void calculate(){
        ...
        computeAll();
    }
    public void calculateArea(){ area = side * side;}
}
public class circle extends Shape{
    public void calcValue(){
        ...
        computeAll();
    }
    public void calculatePerimeter(){ perim = 2*pi*r;}
    public void calculateArea(){ area = pi * r^2;}
}
```

# Preventing misuse

- When extracting duplicate code to super class there are times when some portion of the duplicate code should not be overridden while some portion could be
  - Ex. `initialization`
- Solution is to introduce methods aimed at extensibility while limiting misuse
- Code that should not be overridden would then be marked `final`

# Avoiding misuse example

```
public abstract class Shape{
    public void drawShape(){
        init();
        initShapeImplementation();
        getCenter();
        getPerimeter();
        drawShape();
    }
    final public void init(){
        // initialize drawing canvas
    }
    abstract public void initShapeImplementation(){}
}
public class Square extends Shape{
    public void changeSize(){
        ...
        drawShape();
    }
    public void initShapeImplementation(){
        // initialize shape specific drawing features
    }
}
```

# Complications with delegation

- Complications arise when duplicate code but different detailed implementations

- Ex.

- ```
public class Helper
```
- ```
    // this portion of Computation1 and  
    // Computation 2 are identical  
    void compute() {  
        computeA();  
        computeB();    // implementation differs  
        computeC();  
    }
```

# Complications with delegation

- With inheritance refactoring different implementations of the same method pose little problem
- Method with common code simply calls parent method or abstract method
- With delegation refactoring different implementations are more complex
- Implementation of individual methods is not extracted

# Delegation solution

- With delegation, in order to support different implementations the calling class needs to pass itself to the Helper class
- Helper class then calls the method on the passed in object
- Because different objects do not have the same parent class (otherwise inheritance refactoring would have been used) implementing a common interface is needed



# Delegation example

```
public class Helper{
    public void computeAll(ShapeComputer passedInstance) {
        computeA();
        passedInstance.computeB();
        computeC();
    }
}

public interface ShapeComputer{
    public void computeB();
}

public class square implements ShapeComputer{
    public void calculate(){
        ...
        computeAll();
    }
    public void computeB(){ area = side * side;}
}

public class circle implements ShapeComputer{
    public void calcValue(){
        ...
        computeAll();
    }
    public void computeB(){ area = pi * r^2;}
}
```

- Result is delegation is more likely to become complex than inheritance refactoring