

**SERVICES**

**COMPONENTS**

**OBJECTS**

**MODULES**

# **Code Smells & Refactoring**

**Reid Holmes**

# Program restructuring

- ▶ Software systems represent massive investments.
- ▶ To maintain their value, systems must evolve.
- ▶ The majority (>75%) of software development takes place on existing systems.
- ▶ Software maintenance / evolution comprises the largest proportion of a system's total budget.
- ▶ Systems are modified to:
  - ▶ ?
  - ▶ ?
  - ▶ ?

# Reasons for change

- ▶ Corrective: ? (21%)
- ▶ Adaptive: ? (25%)
- ▶ Perfective: ? (50%)
- ▶ Preventative: ? (4%)

(% from SE Matinenance, Hans van Vilet)

# Reducing change frequency

- ▶ Higher quality --> less (corrective) maintenance
- ▶ Predicting changes --> less (adaptive & perfective) maintenance
- ▶ Better requirements --> less (perfective) maintenance
- ▶ Less code --> less maintenance
- ▶ Regularly perform preventative maintenance

# Lehman's Laws

- ▶ Belady & Lehman proposed 8 laws of software evolution (beginning in 1974).
- ▶ #1 - Systems must evolve.
- ▶ #2 - Systems will become increasingly complex.
- ▶ #6 - Systems must gain new functionality.
- ▶ Lehman's advice:
  - ▶ Complexity must be managed.
  - ▶ Systems must be periodically redesigned and refined.
  - ▶ System and development process is a feedback loop.

# Why is maintenance hard?

- ▶ Unstructured and complex code
  - ▶ Low quality
  - ▶ Poor initial design
  - ▶ Lack of preventative maintenance
- ▶ Insufficient domain knowledge
  - ▶ Change requests push original design
- ▶ Insufficient / stale documentation

# Code smells

- ▶ Symptoms that hint at deeper problems.
- ▶ Can also be considered anti-patterns.
- ▶ Five core groups of smells:
  - ▶ **Bloaters: size becomes overwhelming.**
    - ▶ long method, large class, prim. obsession, long param list, data clump
  - ▶ **OO abusers: OO design not leveraged.**
    - ▶ switch statements, temp field, refused bequest, classes w/ similar interfaces
  - ▶ **Change preventers: Hinder further evolution.**
    - ▶ divergent change, shotgun surgery, parallel inheritance hierarchy
  - ▶ **Dispensables: Unnecessary complexity.**
    - ▶ lazy class, data class, duplicate code, dead code, speculative generality
  - ▶ **Couplers: Unnecessary coupling.**
    - ▶ feature envy, inappropriate intimacy, message chains, middle man

# Removing smells

- ▶ Using refactoring; 3 main steps:
  - ▶ ?
  - ▶ ?
  - ▶ ?
- ▶ Program behaviour should be unchanged.
  - ▶ Appropriate testing is crucial.
- ▶ Refactorings happen in small steps.
  - ▶ Test at each step to make sure everything works.

# Refactoring

- ▶ Should happen as you learn better techniques.
- ▶ Rule of threes:
  - ▶ 1) Code it up.
  - ▶ 2) Code it again (but wince).
  - ▶ 3) Refactor.

# OO abuser: switch

```
► class Animal {  
    final int MAMMAL = 0, BIRD = 1, REPTILE = 2;  
    int myKind; // set in constructor  
  
    String getSkin() {  
        switch (myKind) {  
            case MAMMAL: return "hair";  
            case BIRD: return "feathers";  
            case REPTILE: return "scales";  
            default: return "integument";  
        }  
    }  
}
```

**BEFORE**

```
► class Animal {  
    String getSkin() { return "integument"; }  
}  
class Mammal extends Animal {  
    String getSkin() { return "hair"; }  
}  
class Bird extends Animal {  
    String getSkin() { return "feathers"; }  
}  
class Reptile extends Animal {  
    String getSkin() { return "scales"; }  
}
```

**AFTER**

# Speculative generality example

```
interface MessageStrategy {
    public void sendMessage();
}

abstract class AbstractStrategyFactory {
    public abstract MessageStrategy createStrategy(MessageBody mb);
}
```

```
class MessageBody {
    Object payload;
    public Object getPayload() {
        return payload;
    }
    public void configure(Object obj) {
        payload = obj;
    }
    public void send(MessageStrategy ms) {
        ms.sendMessage();
    }
}
```

## STRATEGY ABSTRACT FACTORY SINGLETON REIFICATION

```
class DefaultFactory extends AbstractStrategyFactory {
    private DefaultFactory() {
    }
    static DefaultFactory instance;
    public static AbstractStrategyFactory getInstance() {
        if (instance == null)
            instance = new DefaultFactory();
        return instance;
    }
    public MessageStrategy createStrategy(final MessageBody mb) {
        return new MessageStrategy() {
            MessageBody body = mb;
            public void sendMessage() {
                Object obj = body.getPayload();
                System.out.println(obj);
            }
        };
    }
}

public class HelloWorld {
    public static void main(String[] args) {
        MessageBody mb = new MessageBody();
        mb.configure("Hello World!");
        AbstractStrategyFactory asf = DefaultFactory.getInstance();
        MessageStrategy strategy = asf.createStrategy(mb);
        mb.send(strategy);
    }
}
```



# Bloater: long method

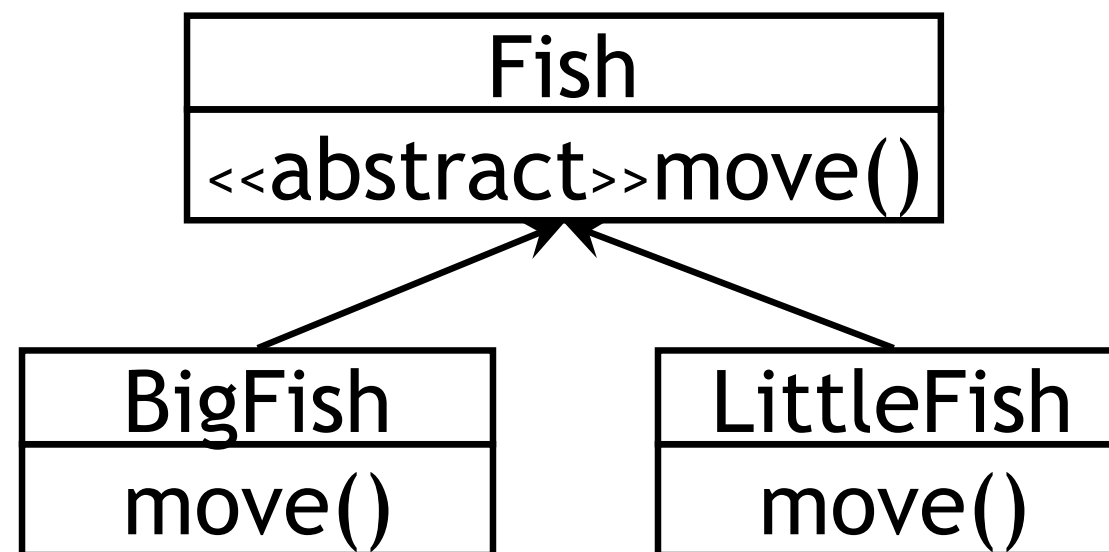
```
public void add(Object element) {  
    if (!readOnly) {  
        int newSize = size + 1;  
        if (newSize > elements.length) {  
            Object[] newElements =  
                new Object[elements.length + 10];  
            for (int i = 0; i < size; i++)  
                newElements[i] = elements[i];  
            elements = newElements;  
        }  
        elements[size++] = element;  
    }  
}
```



```
public void add(Object element) {  
    if (readOnly)  
        return;  
    if (atCapacity())  
        grow();  
    addElement(element);  
}
```

# Dispensables: duplicate code

- ▶ Template method can reduce duplicate code.
- ▶ Consider two fish:
  - ▶ Big fish randomly move anywhere
  - ▶ Little fish move anywhere except where big fish are.



# Fish example

General outline of the method:

```
public void move() {  
    choose a random direction;           // same for both  
    find the location in that direction; // same for both  
    check if it's ok to move there;      // different  
    if it's ok, make the move;          // same for both  
}
```

Solution:

Extract the check on whether it's ok to move

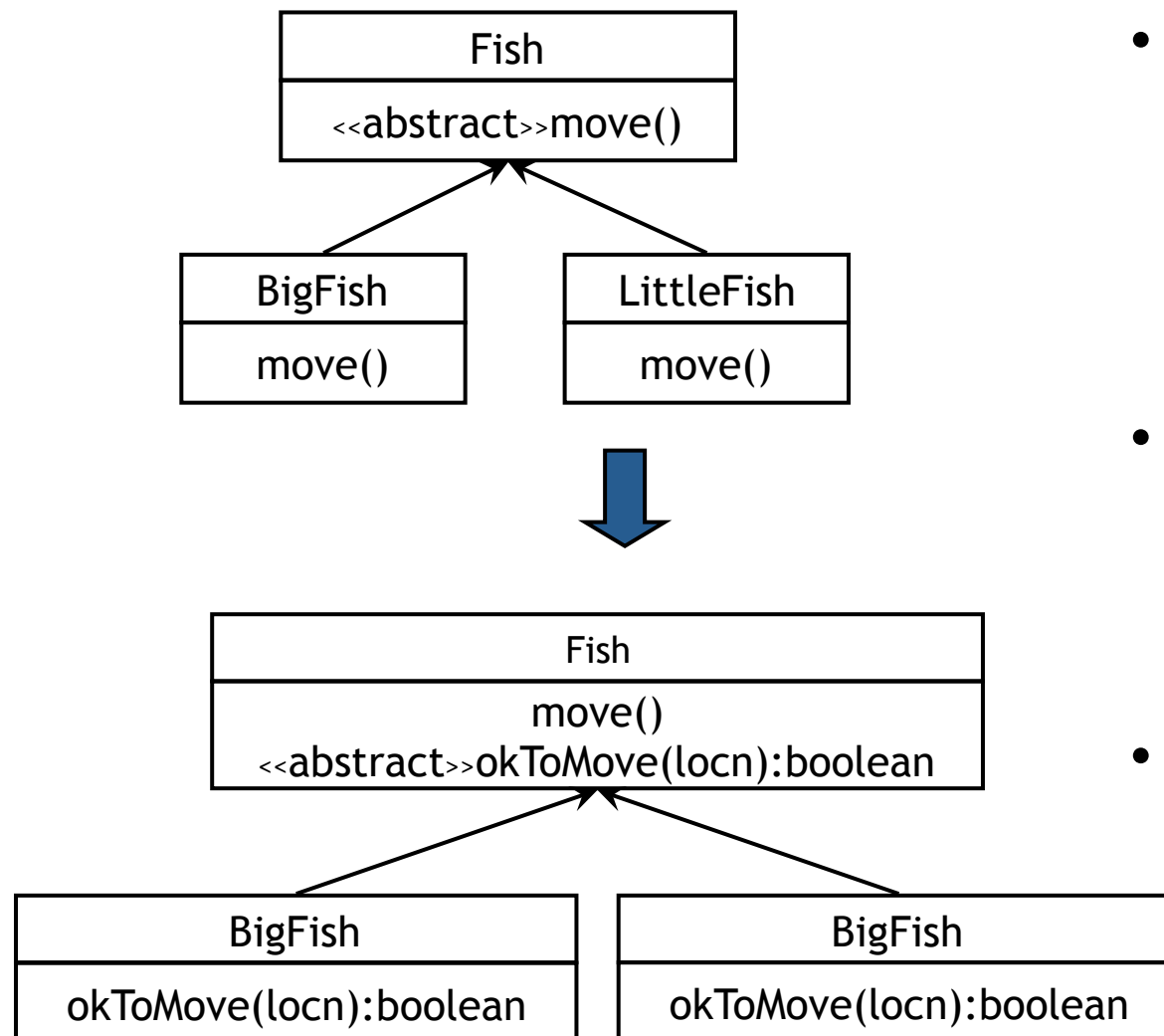
In the **Fish** class, put the actual (template) **move()** method

Create an abstract **okToMove()** method in the Fish class

Implement **okToMove()** in each subclass

# Fish example

- Use a template to vary specific detail without duplicating code.



- Note how this works: When a **BigFish** tries to move, it uses the `move()` method in **Fish**
- But the `move()` method in **Fish** uses the `okToMove(locn)` method in **BigFish**
- And similarly for **LittleFish**