Software Evolution and Refactoring

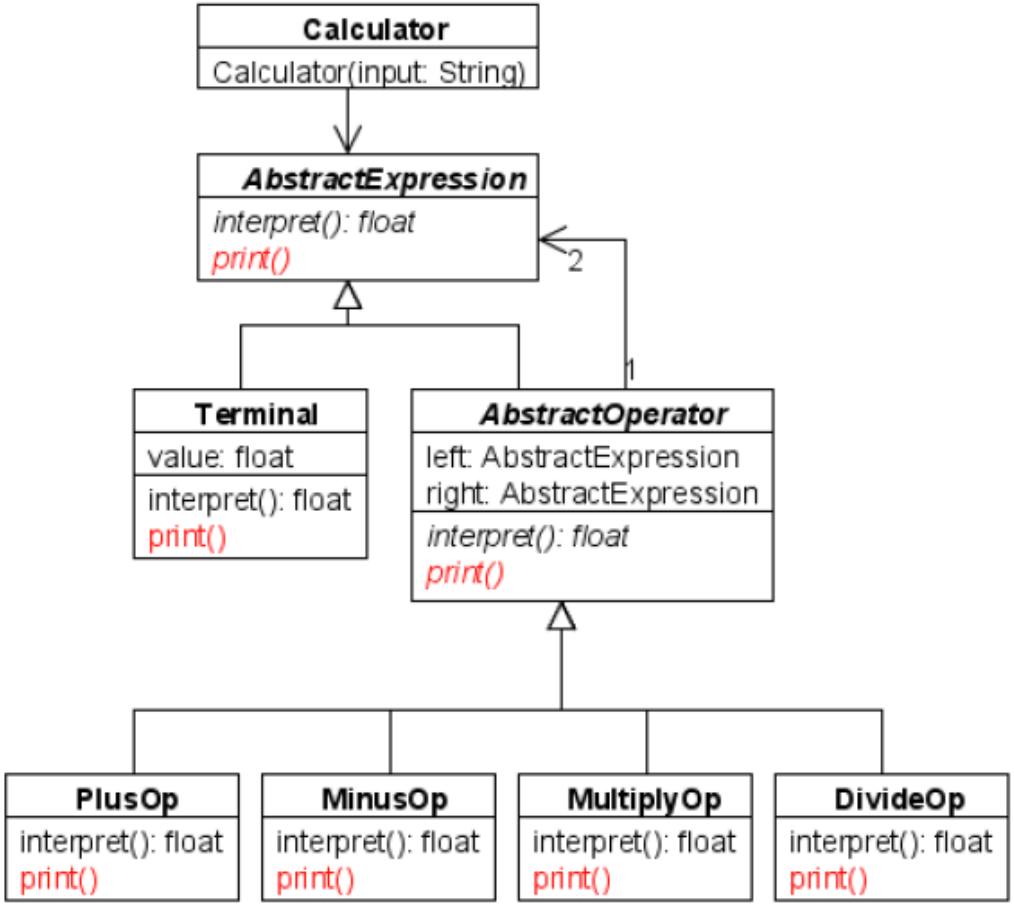
July 2011

Mehdi Amoui

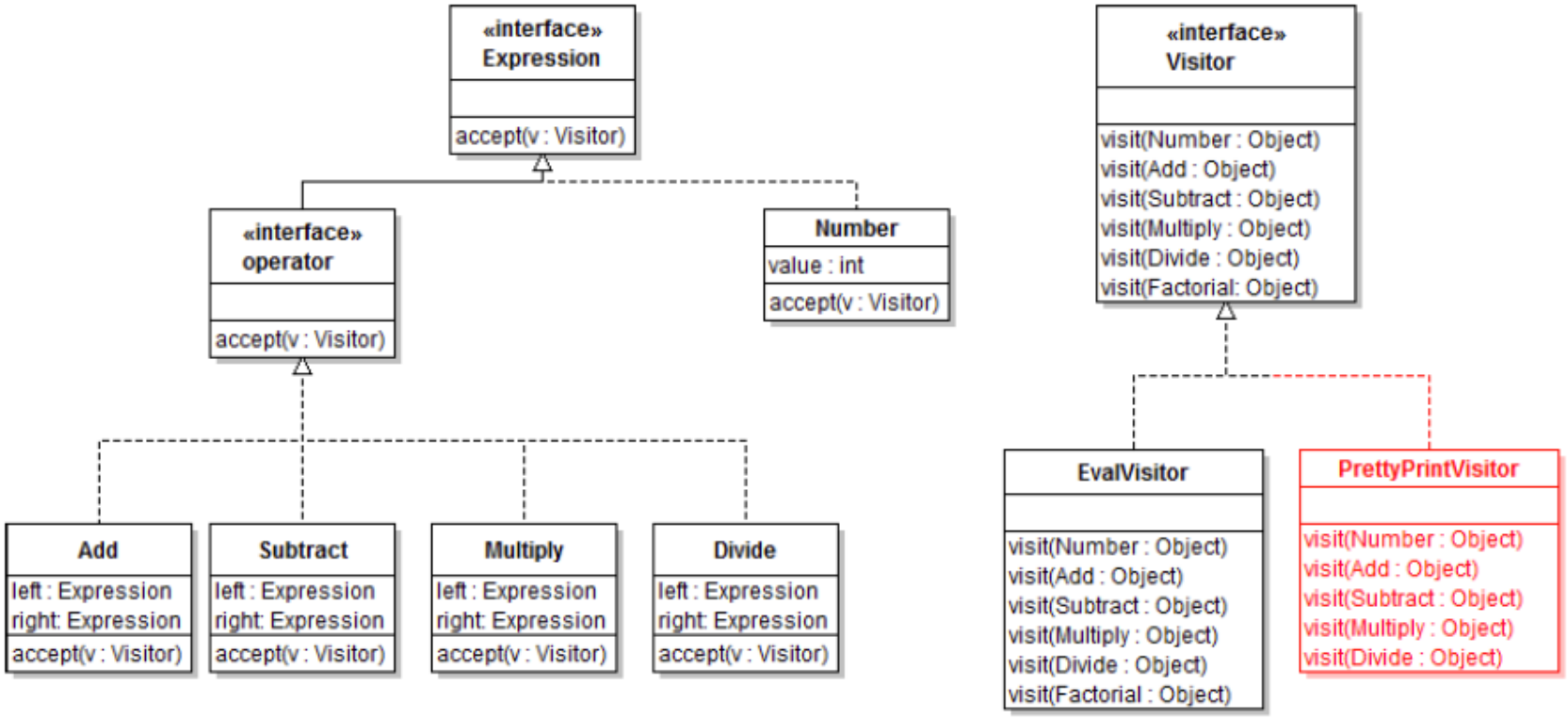
Software Change: Importance

- Developing new software is costly and time consuming
  - It is insane to build a new system for every required change in your software
- Reuse your software in another system
- Add some new features
- Fix some detected bugs
- …

Remember the Interpreter DP



and the Visitor DP



The Problem of Fitness for Future

- How can we design (re-design) and develop software to reduce the cost of future changes?
  - Reduce the total number of changes
  - Reduce the impact of each change
  - Reduce costs and efforts

Another Example (Form Last year Final Exam)

- Why is this implementation bad? How can you improve it? Provide a list of improvements…

```
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";
      }
   }
}
```

Bad/Misused switch statements

- switch statements are very rare in properly designed object-oriented code
  - Therefore, a switch statement is a simple and easily detected “bad smell”
  - Of course, not all uses of switch are bad
  - A switch statement should *not* be used to distinguish between various kinds of object
- There are several well-defined solutions for this case
  - The simplest is the creation of subclasses

Example , improved

```
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"; }
}
```

How is this an improvement?

- Adding a new animal type, such as Insect, does not require revising and recompiling existing code
- Mammals, birds, and reptiles are likely to differ in other ways, and we've already separated them out (so we won't need more switch statements)
- We've gotten rid of the flags we needed to tell one kind of animal from another
- Basically, we're now using Objects the way they were meant to be used

Testing the Change

- As we improve the quality, we need to run JUnit tests to ensure that we haven't introduced errors

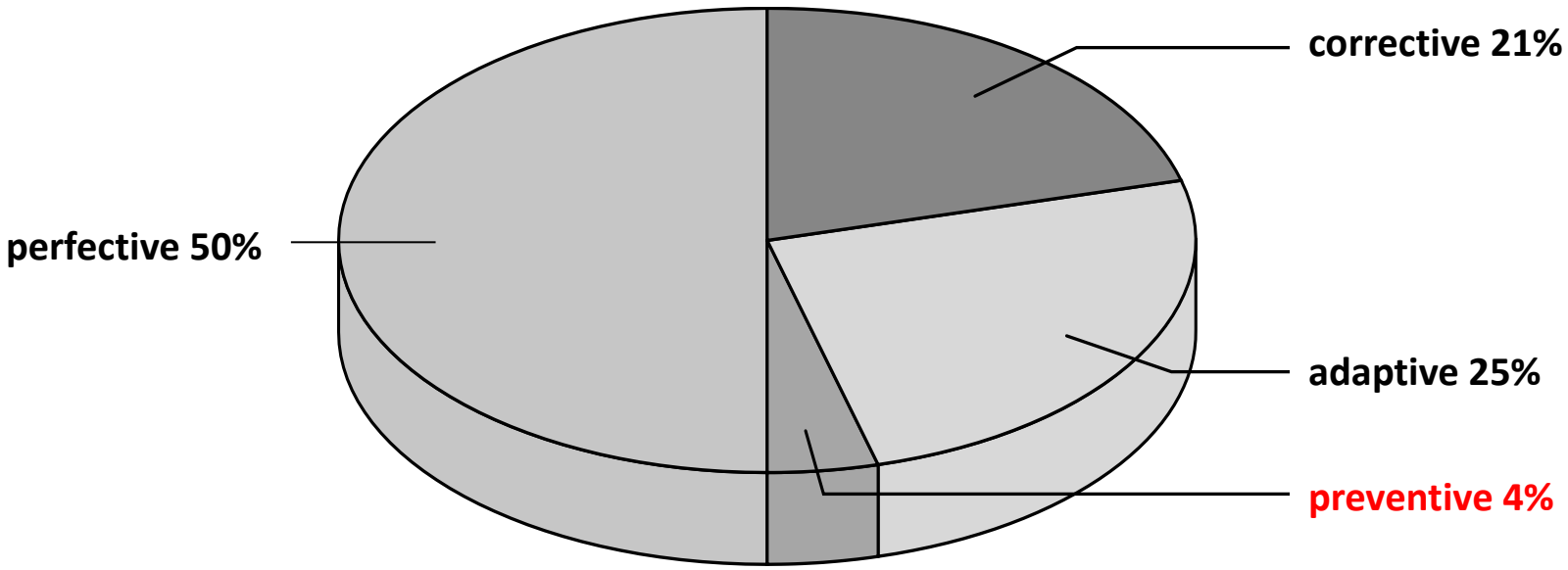
```
public void testGetSkin() {
      assertEquals("hair", myMammal.getSkin());
      assertEquals("feathers", myBird.getSkin());
      assertEquals("scales", myReptile.getSkin());
      assertEquals("integument", myAnimal.getSkin());
}
```

- This should work equally well with either implementation

Reasons for Software Change

- Corrective: Repair software faults
  - Changing a system to correct deficiencies in the way meets its requirements.
- Adaptive: Adapt software to a different operating environment
  - Change for reuse in another system
  - Changing a system so that it operates in a different environment (computer, OS, etc.) from its initial implementation.
- Perfective: Add to or modify the system's functionality
  - Modifying the system to satisfy new requirements.
  - Performance tuning
- Preventive: Improve the program structure
  - Rewriting all or parts of the system to make it more efficient and maintainable.
  - Restructure code, "refactoring", legacy wrapping, build interfaces

Distribution of maintenance activities



How to Reduce Maintenance Changes

- Higher quality $\Rightarrow$ less (corrective) maintenance
- Anticipating changes $\Rightarrow$ less (adaptive and perfective) maintenance
- Better tuning to user needs $\Rightarrow$ less (perfective) maintenance
- Regularly perform preventive maintenance
- Less code $\Rightarrow$ less maintenance (true?)

Lemman's Laws of Software Evolution

- Total 8 Laws
  - Law 1: Software Change is inevitable
  - Law 2: As software changes, it becomes more complex
  - Law 3: Self-Regulation (Predict the change?)
  - Law 4: Conservation of Organizational Stability
  - Law 5: Conservation of Familiarity (Perceived Complexity)
  - Law 6: Continuing Growth (Size of Software)
  - Law 7: Declining Quality
  - Law 8: Evolution processes are feedback systems

- Read more on Original Lemman's paper (1974) and its revisions.

Lemman's Laws in a Nutshell

- Observations:
  - Code Decay: (Most) useful software must evolve or die.
  - Code Ageing: As a software system gets bigger, its resulting complexity tends to limit its ability to grow.

- Advice:
  - Need to manage complexity. (Sources of complexity?)
  - Do periodic redesigns, and refinements.
  - Treat software and its development process as a feedback system.

What Makes software hard to Maintain?

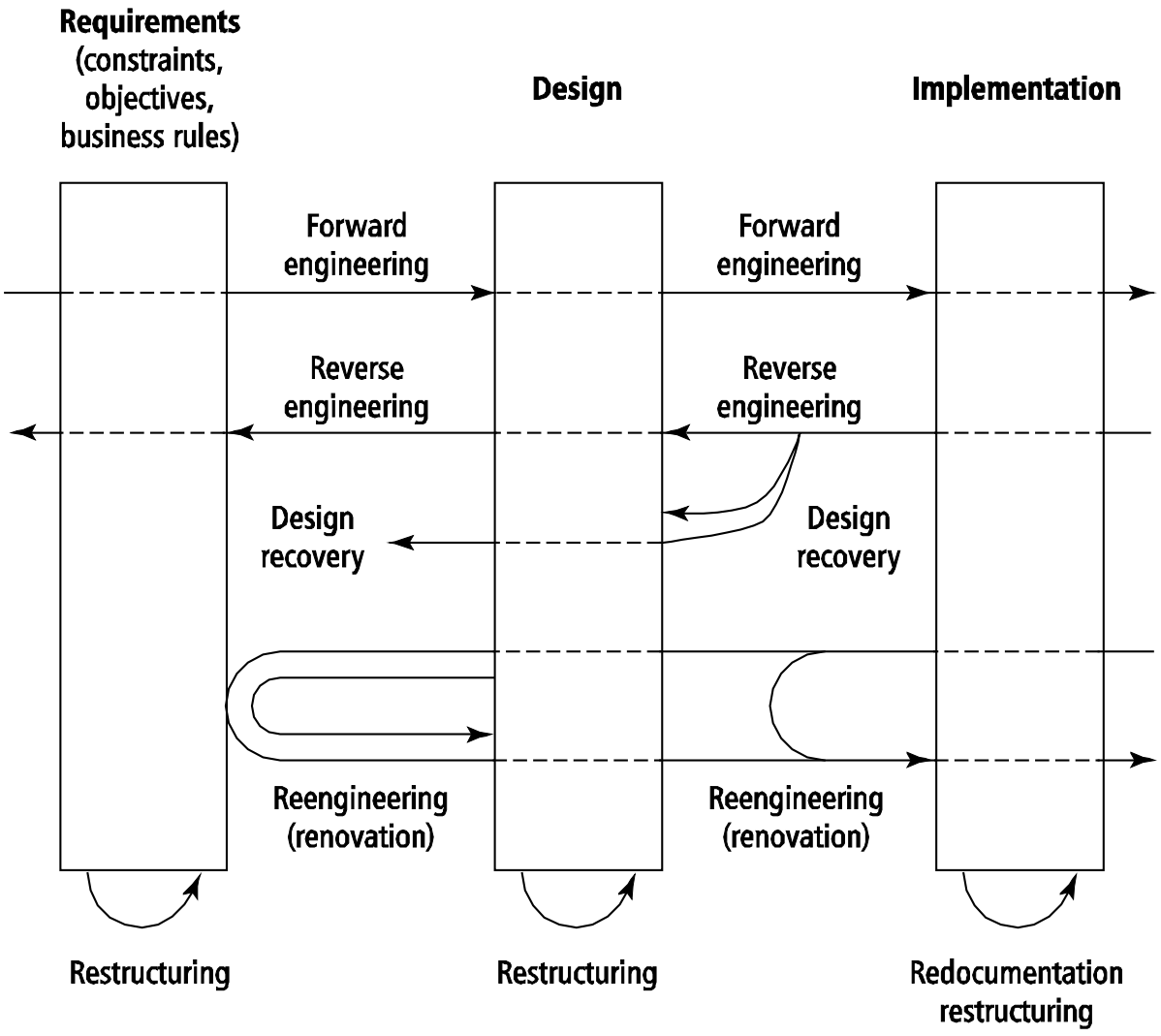
- Unstructured and complex code
  - Low quality and poor design
- Insufficient domain knowledge
  - Predict future demands, change requests, and …
- Insufficient and out of sync documentation
  - How to update design documents?
  - Naming Conversions
  - Commenting
- Lack of regular preventive maintenance
- Maintaining someone else's code

Engineering Vs. Reengineering

- Engineering is a process of designing and developing a new product.

- Reengineering is a process of understanding and changing an existing product for various reasons.
  - Usually, you reengineer a system that you did not developed initially.

Let's Visualize it!



Why do we care?!

- What should we learn in this (Architecture and Design) course to reduce maintenance problems?
  - Quality design
  - Common solutions are easier to maintain (Use design patterns and architectural styles)
  - Use CASE tools and common notations to keep the design in sync with code
  - Redesign (Preventive) to improve quality and remove bad code smells!

Bad Smells in Code

- You code is decaying… It is getting old and ugly… It stinks…

- The most common design problems result from code that:
  - Is duplicated
  - Is unclear
  - Is complicated

Some Code Smells

- Duplicate code
- Long method
- Conditional Complexity
- Data Class
- Solution Sprawl
- Switch statements
- Large class
- Lazy class

- Combinatorial Explosion
- Long Parameter List
- Shutgun Surgery
- Data Clumps

- Comments! (why?)

- And many more…

http://www.codinghorror.com/blog/2006/05/code-smells.html

Five Groups of Bad Code Smells

- **The Bloaters:** represents something that has grown so large that it cannot be effectively handled.
- **The Object-Orientation Abusers:** they represent cases where the solution does not fully exploit the possibilities of object-oriented design.
- **The Change Preventers:** are smells that hinder changing or further developing the software.
- **The Dispensables:** represent something unnecessary that should be removed from the source code.
- **The Couplers:** coupling-related smells.

Bad Smells are the Indicators of System Decay

- Frequent failures
- Overly complex structure
- Very large components
- Excessive resource requirements
- Deficient documentation
- High personnel turnover
- Different technologies in one system

How to remove bad smells?

- Via a reengineering process, three steps:
  - Understanding
  - Transforming
  - Refining
- But we don't want to change the program's behavior!
- So we need a set of transformations that are guaranteed to preserve the behavior while they can remove bad smells
  - We call them "Refactorings"

Refactoring

- Refactoring is:
  - restructuring (rearranging) code...
  - ...in a series of small, semantics-preserving transformations (i.e. the code keeps working)...
  - ...in order to make the code easier to maintain and modify
- Refactoring is *not* just any old restructuring
  - You need to keep the code working
  - You need small steps that preserve semantics
  - You need to have unit tests to prove the code works
- There are numerous well-known refactoring techniques
  - You should be at least somewhat familiar with these before inventing your own

Major Refactoring Catalogs

- *Refactoring: Improving the Design of Existing Code* by Martin Fowler with contributions by Kent Beck, John Brant, William Opdyke, and Don Roberts, Addison-Wesley 1999.

- *Refactoring to Patterns* by Joshua Kerievsky, Addison-Wesley 2004.

Primitive (Basic) Refactorings

- Behavior-preserving transformations:
  - Move Method
  - Rename Method
  - Add Class
  - Extract Method
  - Pull up Method
  - …

See:
www.refactoring.com

- Can be used as building blocks to create the so-called composite refactorings.
  - Example:
    - MoveMethodsToVisitor
    - Add Template Method

When Should We Refactor

- You should refactor:
  - Any time that you see a better way to do things
    - "Better" means making the code easier to understand and to modify in the future
  - You can do so without breaking the code
    - Unit tests are essential for this
  - You smell it! (If it stinks, change it)
- You should *not* refactor:
  - Stable code (code that won't ever need to change)
  - Someone else's code
    - Unless you've inherited it (and now it's yours)

What to refactor

Model (High-level)

- Not all code smells are design smells!
- We need reverse engineering
- More abstract, less detail
- Can be visualized
- Can be hard to reflect the changes to the code (refinement)
- Can be automated (more risky)

Code (Low-level)

- Hard to Understand
- Highly Detailed
- We can perform Unit testing after refactorings
- Can be automated

Refactoring to Patterns

- Argue that design patterns can remove code smells
- More high level
- Each refactoring to patterns is composed of a set of Primitive refactorings
- Don’t Forget Testing after refactoring!

- 21 refactorings are introduced in:
  - *Refactoring to Patterns* by Joshua Kerievsky, Addison-Wesley 2004.

Catalog of Refactorings to Patterns

1. Replace Constructors with Creation methods
2. Encapsulate Classes with Factory
3. Introduce Polymorphic Creation with Factory Method
4. Replace Conditional Logic with Strategy
5. Form Template Method
6. Compose Method
7. Replace Implicit Tree with Composite
8. Encapsulate Composite with Builder
9. Move Accumulation to Collecting Parameter
10. Extract Composite, Replace one/many with Composite.
11. Replace Conditional Dispatcher with Command
12. Extract Adapter, Unify Interfaces with Adapter
13. Replace Type Code with Class

Catalog of Refactorings to Patterns

14. Replace State-Altering Conditionals with State
15. Introduce Null Object
16. Inline Singleton, Limit Instantiation with Singleton
17. Replace Hard-Coded Notifications with Observer
18. Move Embellishment to Decorator, Unify Interfaces, Extract Parameter
19. Move Creation Knowledge to Factory
20. Move Accumulation to Visitor
21. Replace Implicit Language with Interpreter

Are these patterns themselves?

Each refactoring has

- Name and Intent
- Gives an application example
- Discusses motivation
- Benefits and Liabilities
- Mechanics
  - Specific things to do
- Presents detailed example

How to Use Refactorings

- Locate your Code/Design bad smells
- Look up for the list of refactorings to resolve that smell.

- ***Example:*** Conditional Complexity
  - Replace conditional logic with Strategy
  - Move embellishment to Decorator
  - Replace state-altering conditionals with State
  - (Introduce Null Object)

How to use the refactorings

- ***Example:*** Duplicated Code
  - Form Template Method
  - Introduce polymorphic creation with Factory Method
  - (Chain Constructors)
  - Extract composite
  - Unify interfaces with Adapter

Example 2: Duplicated code

- If the same code fragment occurs in more than one place within a single class, you can use *Extract Method*
  - Turn the fragment into a method whose name explains the purpose of the method
  - Any local variables that the method requires can be passed as parameters (if there aren't too many of them!)
  - If the method changes a local variable, see whether it makes sense to return that as the value of the method (possibly changing the name of the method to indicate this)
  - If the method changes two or more variables, you need other refactorings to fix this problem

Example 2: How to remove Duplicate Code

- If the same code fragment occurs in sibling classes, you can use *Extract Method* in both classes, then use *Pull Up Method*
  - Use *ExtractMethod* in each class
  - Be sure the code is identical
  - If necessary, adjust the method signatures to be identical
  - *Copy* the extracted method to the common superclass
  - Delete one subclass method
  - Compile and test
  - Delete the other subclass method
  - Compile and test

Example 2: How to remove Duplicate Code II

- If the same code fragment occurs in unrelated classes, you can use *Extract Method* in one class, then:
  - Use this class as a component in the other class, or
  - Invoke the method from the other class, or
  - Move the method to a third class and refer to it from both of the original classes
- In any case, you need to decide where the method makes most sense, and put it there

Example 2: A Hint

- If *almost* the same code fragment occurs in sibling classes, use *Extract Method* to separate the similar bits from the different bits, and use *Form Template Method*

Intent of Template Method

- Define the skeleton of an algorithm in an operation, deferring some steps to subclasses.

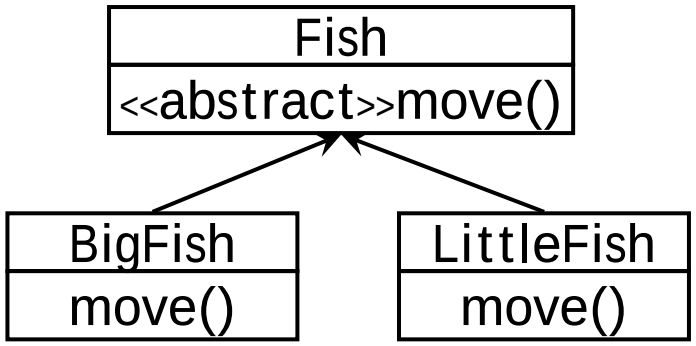
- Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

The Template Method

- Template Methods lead to an inverted control structure
  - A superclass calls methods in its subclass
- Template methods are so fundamental that they can be found in almost every abstract class
- Template Method uses inheritance
- A similar pattern, Strategy Pattern, uses delegation rather than inheritance

Example 2: Big fish and little fish

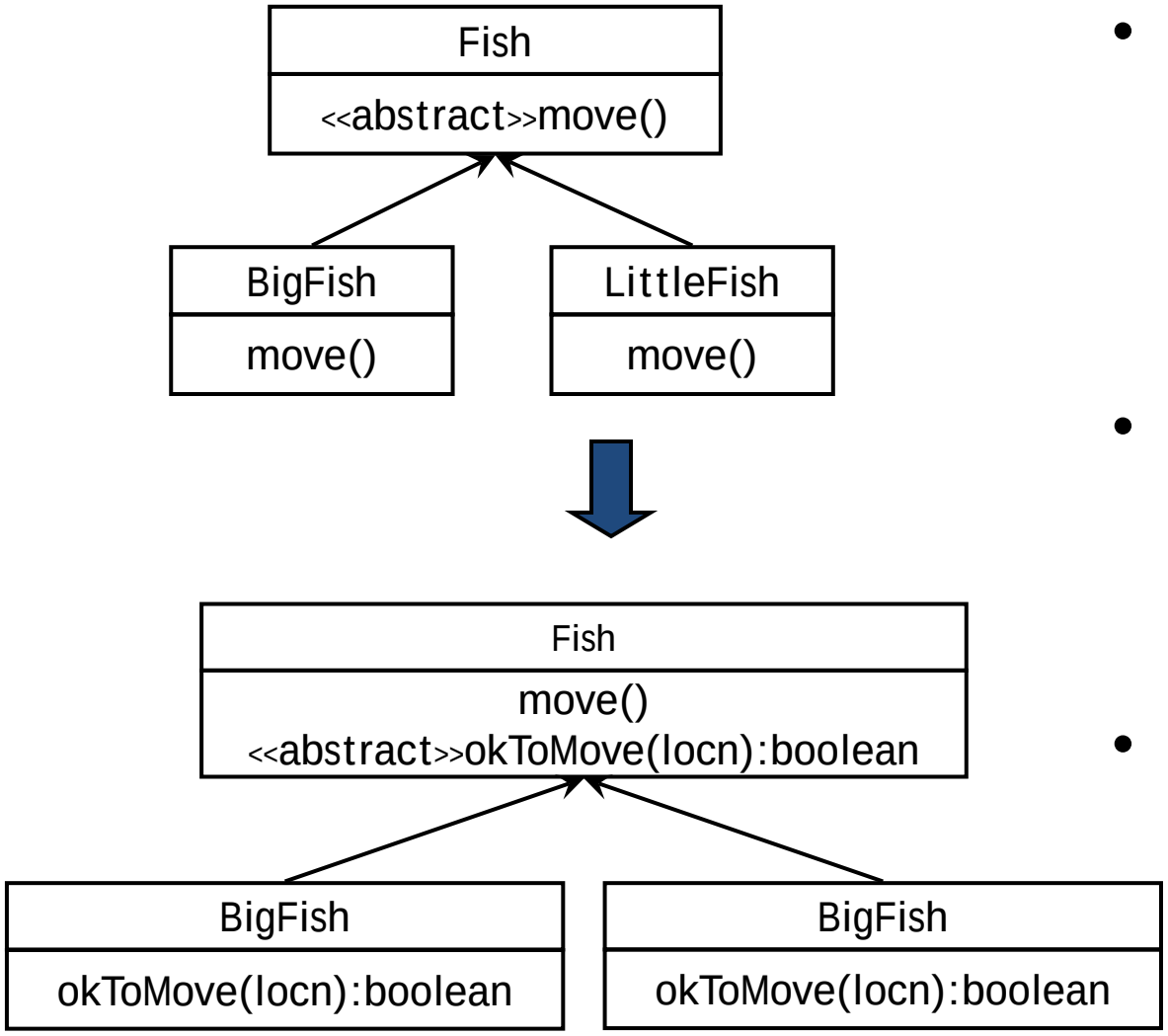
- The scenario: “big fish” and “little fish” move around in an “ocean”
  - Fish move about randomly
  - A big fish can move to where a little fish is (and eat it)
  - A little fish will *not* move to where a big fish is



Example 2: The move() method

- 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

The Fish refactoring



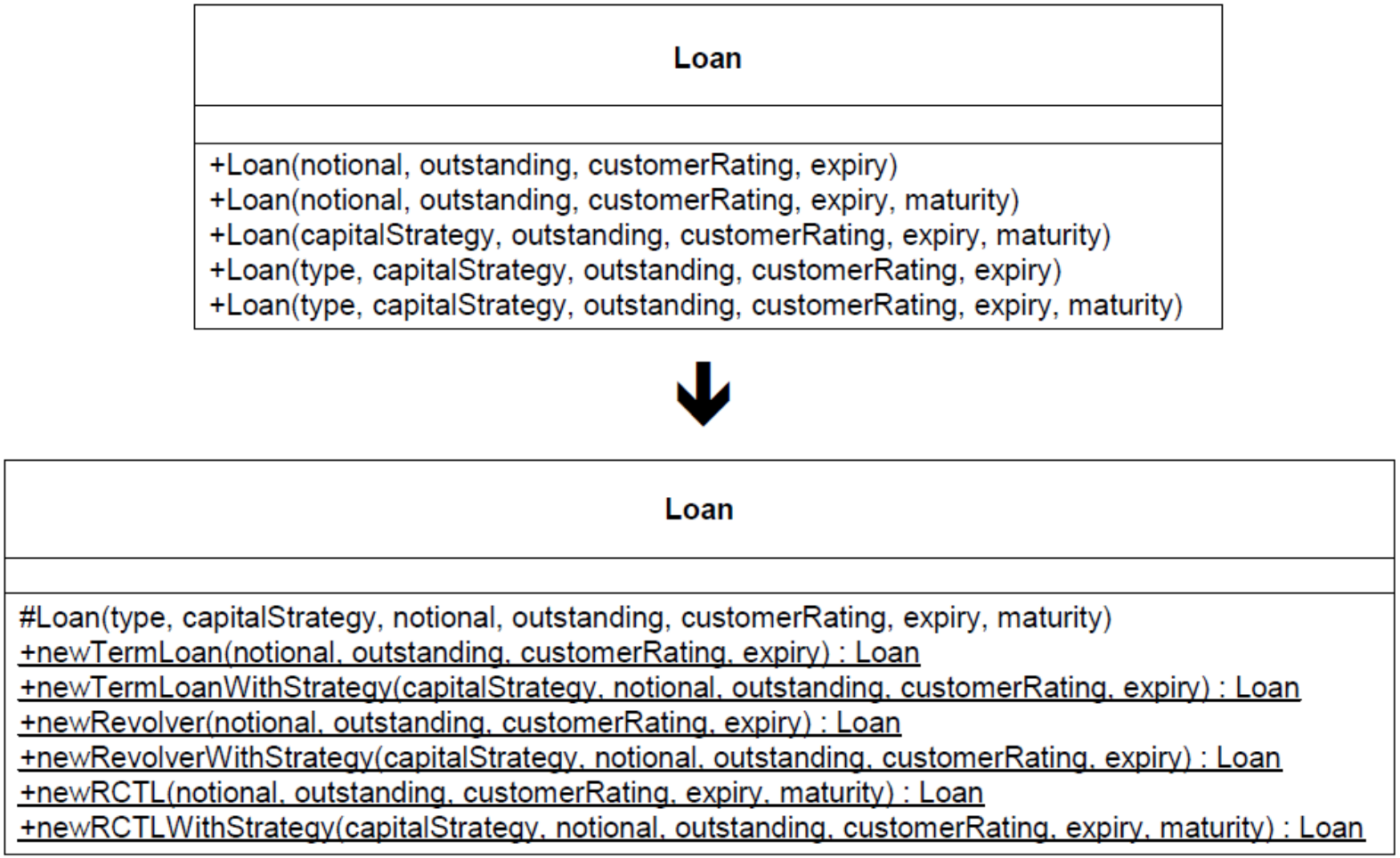
- Note how this works: When a BigFish tries to move, it uses the move() method in Fish
- But the move() method in Fish uses the okToMove(locn) method in BigFish
- And similarly for LittleFish

Example 3: Replace Constructor with Creation (Factory) Methods

- **Problem**: Constructors on a class make it hard to decide which constructor to call during development.

- **Solution**: Replace the constructors with intention-revealing Creation (Factory) Methods that return object instances .

Example 3: Class Diagram



Example 3: Mechanics (Step I)

- Find a client that calls a class's constructor in order to create a kind of instance.
- Apply *Extract Method* on the constructor call to produce a public, static method.
- This new method is a *creation method*.
- Apply *Move Method* to move the creation method to the class containing the chosen constructor.
- Compile and test.

Example 3: Mechanics (Step II)

- Find all callers of the chosen constructor that instantiate the same kind of instance as the *creation method*.
- Update them to call the *creation method*.
- Compile and test.

Example 3: Mechanics (Step III)

- If the chosen constructor is chained to another constructor:
  - Make the creation method call the chained constructor instead of the chosen constructor.
    - Inline the constructor (Apply *Inline method*).
- Compile and Test.

Example 3: Mechanics (Final Step)

- Repeat steps 1-3 for every constructor on the class that you'd like to turn into *Creation Method*.
- If a constructor on the class has no callers outside the class, make it non-public.
- Compile and test.

```
public class Loan {
    private static String TERM_LOAN = "TL";
    private static String REVOLVER = "RC";
    private static String RCTL = "RCTL";
    private String type;
    private CapitalStrategy strategy;
    private float notional;
    private float outstanding;
    private int customerRating;
    private Date maturity;
    private Date expiry;

    public Loan(float notional, float outstanding, int customerRating, Date expiry) {
        this(TERM_LOAN, new TermROC(), notional, outstanding,
        customerRating, expiry, null);
    }
    public Loan(float notional, float outstanding, int customerRating, Date expiry,
        Date maturity) {
        this(RCTL, new RevolvingTermROC(), notional, outstanding, customerRating,
        expiry, maturity);
     }
     public Loan(CapitalStrategy strategy, float notional, float outstanding,
        int customerRating, Date expiry, Date maturity) {
        this(RCTL, strategy, notional, outstanding, customerRating,
        expiry, maturity);
     }
     public Loan(String type, CapitalStrategy strategy, float notional,
        float outstanding, int customerRating, Date expiry) {
        this(type, strategy, notional, outstanding, customerRating, expiry, null);
     }
     public Loan(String type, CapitalStrategy strategy, float notional,
        float outstanding, int customerRating, Date expiry, Date maturity) {
        this.type = type;
        this.strategy = strategy;
        this.notional = notional;
        this.outstanding = outstanding;
        this.customerRating = customerRating;
        this.expiry = expiry;
        if (RCTL.equals(type))
            this.maturity = maturity;
     }
}
```

```
public class Loan {
    private static String TERM_LOAN = "TL";
    private static String REVOLVER = "RC";
    private static String RCTL = "RCTL";
    private String type;
    private CapitalStrategy strategy;
    private float notional;
    private float outstanding;
    private int customerRating;
    private Date maturity;
    private Date expiry;

    protected Loan(String type, CapitalStrategy strategy, float notional,
        float outstanding, int customerRating, Date expiry, Date maturity) {
        this.type = type;
        this.strategy = strategy;
        this notional = notional;
        this.outstanding = outstanding;
        this.customerRating = customerRating;
        this.expiry = expiry;
        if (RCTL.equals(type)
            this.maturity = maturity;
    }
    static Loan newTermLoan(float notional, float outstanding, int customerRating,
        Date expiry) {
        return new Loan(TERM_LOAN, new TermROC(), notional, outstanding, customerRating,
            expiry, null);
    }
    static Loan newTermWithStrategy(CapitalStrategy strategy, float notional,
        float outstanding, int customerRating, Date expiry) {
        return new Loan(TERM_LOAN, strategy, new TermROC(), notional, outstanding,
            customerRating, expiry, null);
    }
    static Loan newRevolver(float notional, float outstanding, int customerRating,
        Date expiry) {
        return new Loan(REVOLVER, new RevolverROC(), notional, outstanding,
            customerRating, expiry, null);
    }
    static Loan newRevolverWithStrategy(CapitalStrategy strategy, float notional,
        float outstanding, int customerRating, Date expiry) {
```

```
        return new Loan(REVOLVER, strategy, new RevolverROC(), notional, outstanding,
            customerRating, expiry, null);
    }
    static Loan newRCTL(float notional, float outstanding, int customerRating,
        Date expiry, Date maturity) {
        return new Loan(RCTL, new RCTLROC(), notional, outstanding,
            customerRating, expiry, maturity);
    }
    static Loan newRCTLWithStrategy(CapitalStrategy strategy, float notional,
        float outstanding, int customerRating, Date expiry, Date maturity) {
        return new Loan(RCTL, strategy, new RevolverROC(), notional, outstanding,
            customerRating, expiry, maturity);
    }
}
```

Example 3: Benefits and Liabilities

- Communicates what kinds of instances are available better than constructors.
- Bypasses constructor limitations, such as the inability to have two constructors with the same number and type of arguments.
- Makes it easier to find unused creation code.

- - Makes creation nonstandard: some classes are instantiated using *new*, while others use *Creation methods.*

Tool Support

- There are several CASE tools to automate the Code and Design Refactoring.
  - Eclipse: has a build in refactoring that supports a set of primitive refactorings.
  - RefactorIT: supports a large set of refactorings. Also detects some code smells by computing design metrics. Eclipse plugin.
  - Borland Together: Famous for its design level refactorings to patterns, but expensive!
  - Some tools exist for VisualStudio, too.

Terminology

- Evolution
- Maintenance
- Re-engineering
- Reverse Engineering
- Program Understanding
- Program Comprehension
- Forward Engineering
- Program Transformation
- Restructuring
- Design Recovery
- Modernization
- Retrofitting
- Renovation
- Migration
- Refactoring
- Refinement
- Replacement
- Redocumentation
- … and many more

Maintenance Vs. Evolution

- In many cases both terms are used interchangeably, or together.
- Some researchers consider evolution as a subset of maintenance, while some others define them vise versa
- Not very precise, but we can say that
  - Software maintenance usually (but not necessarily addresses bug fixes and minor enhancements (e.g. corrective, and preventive).
  - software evolution focuses on more extensive changes (perfective, and adaptive).