Lecturer: Andrea Corradini
andrea@di.unipi.it
http://pages.di.unipi.it/corradini/

AP-12: On Designing Software Frameworks
Software Framework Design

• Intellectual Challenging Task
• Requires a deep understanding of the application domain
• Requires mastering of software (design) patterns, OO methods and polymorphism in particular
• Impossible to address in the course, but we can play a bit...
Four levels for understanding frameworks

1. Frameworks are normally implemented in an object-oriented language such as Java. ➔ Understanding the applicable language concepts, which include inheritance, polymorphism, encapsulation, and delegation.

2. Understanding the framework concepts and techniques sufficiently well to use frameworks to build a custom applications

3. Being able to do detailed design and implementation of frameworks for which the common and variable aspects are already known.

4. Learning to analyze a potential software family, identifying its possible common and variable aspects, and evaluating alternative framework architectures.
A Framework for the family of Divide and Conquer algorithms

• Idea: start from a well-known generic algorithm
• Apply known techniques and patterns to define a framework for a software family
• Instances of the framework, obtained by standard extension mechanism, will be concrete algorithms of the family

```java
function solve (Problem p) returns Solution {
  if isSimple(p)
    return simplySolve(p);
  else
    sp[] = decompose(p);
    for (i = 0; i < sp.length; i = i+1)
      sol[i] = solve(sp[i]);
    return combine(sol);
}
```
Some terminology...

- **Frozen Spot**: common (shared) aspect of the software family
- **Hot Spot**: variable aspect of the family
- **Template method**: concrete method of base (abstract) class implementing behavior common to all members of the family
- A hot spot is represented by a group of abstract **hook methods**.
- A template method calls a hook method to invoke a function that is specific to one family member [*Inversion of Control*]
- A hot spot is realized in a framework as a **hot spot subsystem**:
  - An abstract base class + some concrete subclasses
Two Principles for Framework Construction

• The *unification principle*  [Template Method DP]
  – It uses *inheritance* to implement the *hot spot subsystem*
  – Both the *template methods* and *hook methods* are defined in the same abstract base class
  – The hook methods are implemented in subclasses of the base class

• The *separation principle*  [Strategy DP]
  – It uses *delegation* to implement the *hot spot subsystem*
  – The *template methods* are implemented in a *concrete context class*; the *hook methods* are defined in a *separate abstract class* and implemented in its subclasses
  – The template methods delegate work to an instance of the subclass that implements the hook methods
The **Template Method** design pattern

- One of the behavioural pattern of the Gang of Four
- **Intent**: Define the skeleton of an algorithm in an operation, deferring some steps to subclasses.
- A **template method** belongs to an abstract class and it defines an algorithm in terms of abstract operations that subclasses override to provide concrete behavior.
- Template methods call, among others, the following operations:
  - **concrete operations** of the abstract class (i.e., fixed parts of the algorithm);
  - **primitive operations**, i.e., abstract operations, that subclasses **have** to implement; and
  - **hook operations**, which provide default behavior that subclasses **may** override if necessary. A hook operation often does nothing by default.
```
Final templateMethod(): void {
    operation1();
    operation2();
}
```
Implementation of Template Methods

- **Using Java visibility modifiers**
  - The template method itself should not be overridden: it can be declared a `public final` method.
  - The **concrete operations** can be declared `private` ensuring that they are only called by the template method.
  - **Primitive operations** that must be overridden are declared `protected abstract`.
  - The hook operations that **may** be overridden are declared `protected`.

- **Using C++ access control**
  - The template method itself should not be overridden: it can be declared a `nonvirtual member function`.
  - The **concrete operations** can be declared `protected members` ensuring that they are only called by the template method.
  - **Primitive operations** that must be overridden are declared `pure virtual`.
  - The hook operations that **may** be overridden are declared `protected virtual`.
The **Strategy** design pattern

- One of the behavioural pattern of the Gang of Four
- **Intent**: Allows to select (part of) an algorithm at runtime
- The client uses an object implementing the interface and invokes methods of the interface for the hot spots of the algorithm
Applying the unification principle: UML diagram of the solution

```java
function solve (Problem p) returns Solution // template method
{ if isSimple(p)
    return simplySolve(p); // hot spots
else
    sp[] = decompose(p);
    for (i= 0; i < sp.length; i = i+1)
        sol[i] = solve(sp[i]);
    return combine(sol);
}"
Java code of the framework *(unification principle)*

```java
public interface Problem {}
public interface Solution {};

abstract public class DivConqTemplate
{
  public final Solution solve(Problem p)
  {
    Problem[] pp;
    if (isSimple(p)) { return simplySolve(p); } 
    else { pp = decompose(p); }
    Solution[] ss = new Solution[pp.length];
    for(int i=0; i < pp.length; i++)
    {  ss[i] = solve(pp[i]);  }
    return combine(p,ss);
  }
  abstract protected boolean isSimple (Problem p);
  abstract protected Solution simplySolve (Problem p);
  abstract protected Problem[] decompose (Problem p);
  abstract protected Solution combine(Problem p,Solution[] ss);
}
```

function `solve` (Problem p) returns Solution  // **template method**  
   /  // **hot spots**
{
  if `isSimple`(p)
    return `simplySolve`(p);
  else
    sp[] = `decompose`(p);
    for (i= 0; i < sp.length; i = i+1)
      sol[i] = solve(sp[i]);
  return `combine`(sol);
}
An application of the framework: QuickSort

(unification principle)

- In-place sorting
- Both problem and solution described by the same structure: <array, first, last>

### QuickSortDesc.java

```java
public class QuickSortDesc implements Problem, Solution {
    public QuickSortDesc(int[] arr, int first, int last) {
        this.arr = arr; this.first = first; this.last = last;
    }
    public int getFirst() { return first; }
    public int getLast() { return last; }
    private int[] arr; // instance data
    private int first, last;
}
```

Fig. 5. Quicksort Problem and Solution implementation.

### QuickSort.java

```java
public class QuickSort extends DivConqTemplate {
    protected boolean isSimple (Problem p) {
        return ((QuickSortDesc)p).getFirst() >= ((QuickSortDesc)p).getLast();
    }
    protected Solution simplySolve (Problem p) {
        return (Solution) p; }
    protected Problem[] decompose (Problem p) {
        int first = ((QuickSortDesc)p).getFirst();
        int last = ((QuickSortDesc)p).getLast();
        int[] a = ((QuickSortDesc)p).getArr();
        int x = a[first]; // pivot value
        int sp = first;
        for (int i = first + 1; i <= last; i++) {
            if (a[i] < x) { swap (a, ++sp, i); }
        }
        swap (a, first, sp);
        Problem[] ps = new QuickSortDesc[2];
        ps[0] = new QuickSortDesc(a,first,sp-1);
        ps[1] = new QuickSortDesc(a,sp+1,last);
        return ps;
    }
    protected Solution combine (Problem p, Solution[] ss) {
        return (Solution) p; }
    private void swap (int[] a, int first, int last) {
        int temp = a[first];
        a[first] = a[last];
        a[last] = temp;
    }
}
```

### QuickSortDesc.java

```java
public class QuickSort extends DivConqTemplate {
    protected boolean isSimple (Problem p) {
        return ((QuickSortDesc)p).getFirst() >= ((QuickSortDesc)p).getLast();
    }
    protected Solution simplySolve (Problem p) {
        return (Solution) p; }
    protected Problem[] decompose (Problem p) {
        int first = ((QuickSortDesc)p).getFirst();
        int last = ((QuickSortDesc)p).getLast();
        int[] a = ((QuickSortDesc)p).getArr();
        int x = a[first]; // pivot value
        int sp = first;
        for (int i = first + 1; i <= last; i++) {
            if (a[i] < x) { swap (a, ++sp, i); }
        }
        swap (a, first, sp);
        Problem[] ps = new QuickSortDesc[2];
        ps[0] = new QuickSortDesc(a,first,sp-1);
        ps[1] = new QuickSortDesc(a,sp+1,last);
        return ps;
    }
    protected Solution combine (Problem p, Solution[] ss) {
        return (Solution) p; }
    private void swap (int[] a, int first, int last) {
        int temp = a[first];
        a[first] = a[last];
        a[last] = temp;
    }
}
```

Fig. 6. Quicksort application.

- **Merge-sort** can be defined similarly
- In that case, **combine** would do most of the work
Applying the separation principle: UML diagram of the solution

Fig. 7. Strategy pattern for divide and conquer framework.

function solve (Problem p) returns Solution // template method
{ if isSimple(p)
   return simplySolve(p); // hot spots
else
   sp[] = decompose(p);
   for (i= 0; i < sp.length; i = i+1)
      sol[i] = solve(sp[i]);
   return combine(sol); }
The client **delegates** the hot spots to an object implementing the strategy.

The implementations of `DivConqStrategy` are similar to the previous case.

```java
public final class DivConqContext
{
    public DivConqContext (DivConqStrategy dc)
    {
        this.dc = dc;
    }
    public Solution solve (Problem p)
    {
        Problem[] pp;
        if (dc.isSimple(p)) { return dc.simplySolve(p); }
        else { pp = dc.decompose(p); }
        Solution[] ss = new Solution[pp.length];
        for (int i = 0; i < pp.length; i++)
        {
            ss[i] = solve(pp[i]);
        }
        return dc.combine(p, ss);
    }
    public void setAlgorithm (DivConqStrategy dc)
    {
        this.dc = dc;
    }
    private DivConqStrategy dc;
}
```

**Fig. 8.** Strategy context class implementation.

```java
abstract public class DivConqStrategy
{
    abstract public boolean isSimple (Problem p);
    abstract public Solution simplySolve (Problem p);
    abstract public Problem[] decompose (Problem p);
    abstract public Solution combine(Problem p, Solution[] ss);
}
```

**Fig. 9.** Strategy object abstract class.
• The two approaches differ in the coupling between client and chosen algorithm
• With Strategy, the coupling is determined by dependency (setter) injection, and could change at runtime
Framework development by generalization

• We address now level 4 of "framework understanding"
  – Learning to analyze a potential software family, identifying its possible common and variable aspects, and evaluating alternative framework architectures. Framework design involves incrementally evolving a design rather than discovering it in one single step.

• This evolution consists of
  – examining existing designs for family members
  – identifying the frozen spots and hot spots of the family
  – generalizing the program structure to enable
    • reuse of the code for frozen spots and
    • use of different implementations for each hot spot.

• We present an example based on binary trees traversals, starting from a concrete algorithm for printing a tree with preorder traversal
Binary trees and preorder traversal

Fig. 10. Binary tree using Composite design pattern.

Binary trees as instance of the **Composite** design pattern
- Provides uniform access to nodes and to leaves

Pseudo-code of generic **depth-first preorder left-to-right** traversal
(action not specified)

```plaintext
procedure preorder(t)
{    if t null, then return;
    perform visit action for root node of tree t;
    preorder(left subtree of t);
    preorder(right subtree of t);
}
```
Binary tree class hierarchy

Abstract class defining defaults and abstract methods

Implementation of the abstract class for Nodes
- The **action** simply prints

Implementation of the abstract class for leaves, using the **Singleton DP**
Identifying **Frozen and Hot Spots**

Possible choices, generalizing the concrete program to a family of tree-traversal algorithms

- **Frozen Spots** (fixed for the whole family)
  - The structure of the tree, as defined by the BinTree hierarchy
  - A traversal accesses every element of the tree once, but it can stop before completing
  - A traversal performs one or more visit actions accessing an element of the tree
Identifying Frozen and **Hot Spots**

- **Hot Spots** (to be fixed in each element of the family)
  1. Variability in the *visit operation’s action*: a function of the *current node’s value* and the *accumulated result*
  2. Variability in *ordering* of the visit action with respect to subtree traversals. Should support *preorder*, *postorder*, *in-order*, and their combination
  3. Variability in the *tree navigation technique*. Should support any access order (not only left-to-right, depth-first, total traversals)
Hot Spot #1: Generalizing the visit action

- Using the separation principle (Strategy pattern) we allow different visit actions on the same tree
- action is represented by the abstract method visitPre
- It takes an accumulator Object and a BinTree as arguments

```java
public interface PreorderStrategy
{
    abstract public Object visitPre(Object ts, BinTree t);
}

abstract public class BinTree
{
    ...
    abstract public Object preorder(Object ts, PreorderStrategy v);
    ...
}

public class Node extends BinTree
{
    ...
    public Object preorder(Object ts, PreorderStrategy v) //traversal
    {
        ts = v.visitPre(ts, this);
        ts = left.preorder(ts, v);
        ts = right.preorder(ts, v);
        return ts;
    }
    ...
}

public class Nil extends BinTree
{
    ...
    public Object preorder(Object ts, PreorderStrategy v)
    {
        return ts;
    }
    ...
}
```

New BinTree hierarchy.

The preorder method takes the action from the strategy and handles accumulation.

**Exercise:** define strategies for printing the values of the nodes, and for computing the sum/max of all node values.
Hot Spot #2: Generalizing the visit order

We generalize the previous hot spot subsystem

- The Euler Strategy visits each node three times (left = pre, right = post, bottom = in)

preorder is now traverse

Using the new abstract methods an Euler Strategy can implement any combination of pre-order, post-order or in-order traversal

Also visitNil method added, for the sake of generality
Hot Spot #3: Generalizing the tree navigation

• Support for breadth-first, depth-first, left-to-right, right-to-left, partial traversal, ...

• Remember the frozen spots:
  – The structure of the tree, as defined by the BinTree hierarchy: it cannot be modified
  – A traversal accesses every element of the tree once, but it can stop before completing

• Instead of generalizing the traverse method, we use the Visitor design pattern

• Visitor guarantees separation between algorithm and data structure
The **Visitor** design pattern

- The data structure can be made of different types of components (*ConcreteElements*)
- Each component implements an `accept(Visitor)` method
- The **Visitor** defines one `visit` method for each type
- The navigation logic is in the **Visitor**
- At each step, the correct `visit` method is selected by **overloading**
Hot Spot #3: Binary Tree Visitor framework

Fig. 14. Binary tree Visitor framework.
Binary Tree Visitor framework: the BinTree code

```java
public interface BinTreeVisitor {
    abstract void visit(Node t);
    abstract void visit(Nil t);
}

abstract public class BinTree {
    public void setValue(Object v) { } // mutators
    public void setLeft(BinTree l) { } // default
    public void setRight(BinTree r) { }
    abstract public void accept(BinTreeVisitor v); // accept Visitor
    public Object getValue() { return null; } // accessors
    public BinTree getLeft() { return null; } // default
    public BinTree getRight() { return null; }
}

public class Node extends BinTree {
    private BinTree left, right;
    private Object value; // instance data
    public BinTree getRight() { return null; }
    public BinTree getLeft() { return left; }
    public Object getValue() { return value; } // accessors
    public void accept(BinTreeVisitor v) { v.visit(this); }
    public void setRight(BinTree r) { } // default
    public void setLeft(BinTree l) { } // default
    public void setValue(Object v) { } // mutators
    public Node(Object v, BinTree l, BinTree r) {
        value = v; left = l; right = r;
    }
}

public class Nil extends BinTree {
    private Nil() { } // private to require use of getNil()
    public void accept(BinTreeVisitor v) { v.visit(this); }
    static public BinTree getNil() { return theNil; } // Singleton
    static public BinTree theNil = new Nil();
}
```

The BinTree code is almost unchanged, only the `traverse` method is changed to
- accept an instance of `Visitor`
- invoke `visit(this)` on it
Binary Tree Visitor framework: defining a visitor for Euler Traversal

• The Visitor framework has two levels
  – the Visitor pattern as described above
  – Possibly a second framework for the design of the Visitor objects.

• To implement an Euler tour traversal we
  – design a concrete class `EulerTourVisitor` that implements the `BinTreeVisitor` interface
  – this class delegates the specific visit actions to a Strategy object of type `EulerStrategy`.

Fig. 16. Euler tour traversal Visitor framework.
Visitor for Euler Traversal using Strategy

```java
public interface EulerStrategy {
    abstract public Object visitLeft(Object ts, BinTree t);
    abstract public Object visitBottom(Object ts, BinTree t);
    abstract public Object visitRight(Object ts, BinTree t);
    abstract public Object visitNil(Object ts, BinTree t);
}
```

```java
public class EulerTourVisitor implements BinTreeVisitor {
    private EulerStrategy es; // encapsulates state changing ops
    private Object ts; // traversal state

    public EulerTourVisitor(EulerStrategy es, Object ts) {
        this.es = es; this.ts = ts;
    }

    public void setVisitStrategy(EulerStrategy es) // mutators
    {
        this.es = es;
    }

    public void setResult(Object r) { ts = r; }

    public void visit(Node t) // Visitor hook implementations
    { t.s = es.visitLeft(ts,t); // upon first arrival from above
        t.getLeft().accept(this);
        ts = es.visitBottom(ts,t); // upon return from left
        t.getRight().accept(this);
        ts = es.visitRight(ts,t); // upon completion of this node
    }

    public void visit(Nil t) { ts = es.visitNil(ts,t); }

    public Object getResult(){ return ts; } // accessor

    private EulerStrategy es; // encapsulates state changing ops
    private Object ts; // traversal state
}
```

- The navigation logic is in the `visit()` method.
- It exploits `accept()` to pass to the next node.
- The concrete actions are defined in an object implementing `EulerStrategy`.
- The strategy is injected with the constructor and can be changed dynamically.
Conclusions

• Software Framework design is a complex task
• Starting point: families of homogeneous software applications
• Identification of frozen spots and hot spots
• Use of design patterns and of other techniques for greater generality and for reducing coupling
• Inversion of control and in particular dependency injection arise naturally
• Suggested reading: *Why do I hate Frameworks?* By Joel Spolsky, co-founder of Stack Overflow