Quality of Service

csc 408 – tutorial #8

Product Quality:
    Customer Satisfaction
Process Quality:
    Developer Satisfaction

The project raw mark depends …

- W: Your web service users
- C: Web services you used
- N: Number of integrated systems delivered
- B_i: Number of bugs found for integrated system
- $Q_i = f(B_i)$: Quality of each integrated system
  f is a monotonic increasing function ranges from 0 to 1
- Total quality:
  TQ = 1 – [w_wProd(1-Q_i | i in W) + w_cProd(1-Q_i | i in C)]
  $w_w + w_c = 1$, $w_w > w_c$
- Mark: M = g(|W union C|) * h(TQ)
  g, h are monotonic increasing functions, to be decided

Satisfaction

- Customer is satisfied with good quality product and support
- Developer is satisfied with good quality process
- Satisfaction has multiple dimensions:
  Correctness (required)
  Reliability (required)
  Performance, Scalability (desired)
  Maintainability (desired)
- How to guarantee them?
  management, measuring, tuning, configuration

Correctness – verification

- Verification of the web service
  - Does their implementation match their specification?
  - A fault can be found by a test according to *their* test cases.
    - i.e. Verifying their claim

Correctness – your webservice

- The first task for developing your client is to negotiate with the web service provider
  - –Syntax
  - –Semantics

Correctness – validation

- Validation
  - –Does their implementation match *my* specification?
  - –A fault can be found by a test according to *my* test cases.

Stock Price Example

- Verifying Interface (syntax differences)

```
float getQuote(String name, String marketplace);
// market place stands for NASDAQ, NYSE, etc
float getQuote(String name);
```

- Checking Specification (Semantics differences)

```
float getQuote(String name);
// precondition: name = ticker symbol
// postcondition: return -1 if name does not exist
float getQuote(String name);
// precondition: name = part of the full name
// postcondition: return -1 if name does not exist,
//                       -2 if multiple matches
```

Reliability

- Reliability means the software does not fail
  - At least high confidence it does not fail
- Also measured by how quickly a failure is fixed
- These are both non-functional qualities
  - Highly desirable
  - Can be expensive (profitable?) to provide

Reliability

- Failure for installation and deployment
  - Web services alleviate the problem by allowing updating implementation without installation
  - However, the WSDL interface should not be changed frequently
- Failure for execution
  - Memory leaks
  - Too many clients running at the same time
  - Exceptions not handled
  - DoS attacks
  - Shutdown of the machine (high risk)
- Bugzilla: bug in bugzilla has a unfixed duration

Performance and complexity

- See tutorial 5

Developer satisfaction: Refactoring for Maintainability

- Maintainability = Understandable and Flexible
  - Simplicity helps maintainability
  - Good structure also helps maintainability

Refactoring

- What is refactoring?
  A sequence of small changes to a program that improve its structures without changing observable behaviors
- The following activities are not refactoring:
  - Adding more functionalities
  - Correcting system errors is not refactoring
  - Performance tuning is not refactoring
    because it may not improve the maintainability

Refactoring

- We emphasize refactoring, for project
  - Maintenance & Clean-up
  - Make Unit-test cases first!
- Commit early, commit often
  - Less overhead, stay in synch
  - Logical: take big problem, break it down into manageable, documented, progressive steps

Refactoring Examples

Martin Fowler, the Refactoring book.

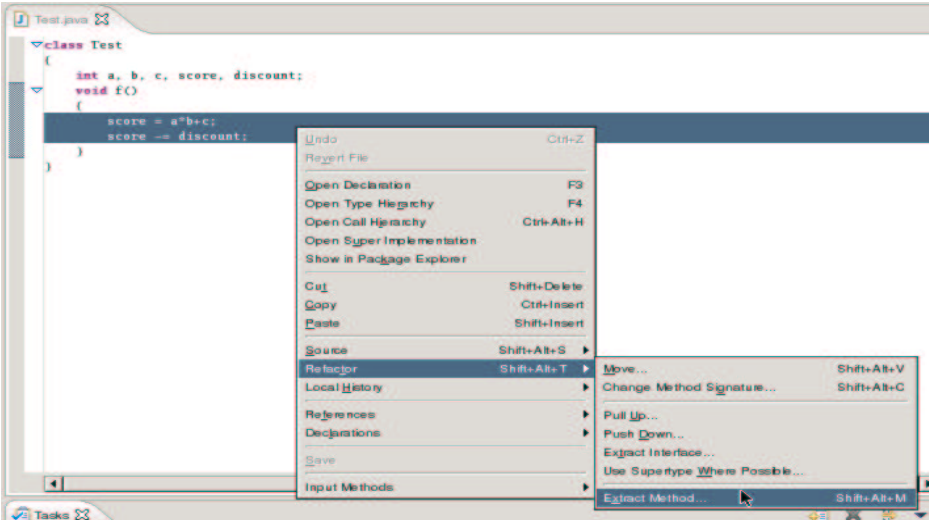
- Refactoring mechanisms supported by Eclipse
- Examples
  - Extract Method
  - Move Method
  - Lift Method to additional class

Example – extract method

```
void f() {
    ...
    // Compute score
    score = a * b + c;
    score -= discount;
}
```

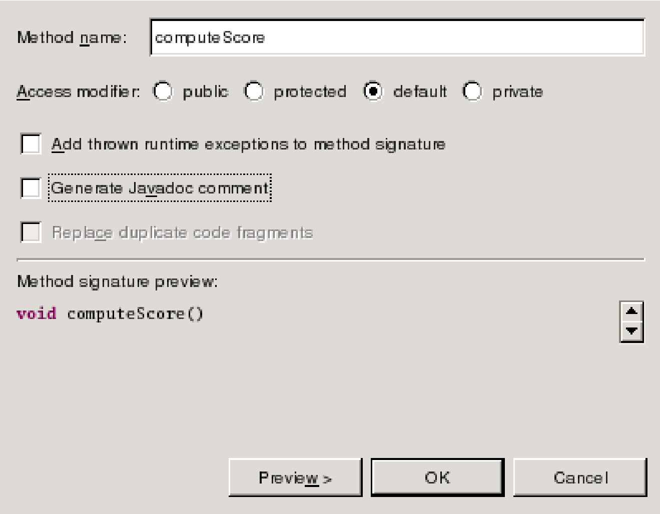
```
void f() {
    ...
    computeScore();
}

void computeScore() {
    score = a * b + c;
    score -= discount;
}
```

Example – extract method



```
class Test
{
    int a, b, c, score, discount;
    void f()
    {
        score = a*b+c;
        score -= discount;
    }
}
```

Example – extract method



Method name: computeScore

Access modifier: ○ public ○ protected ● default ○ private

□ Add thrown runtime exceptions to method signature

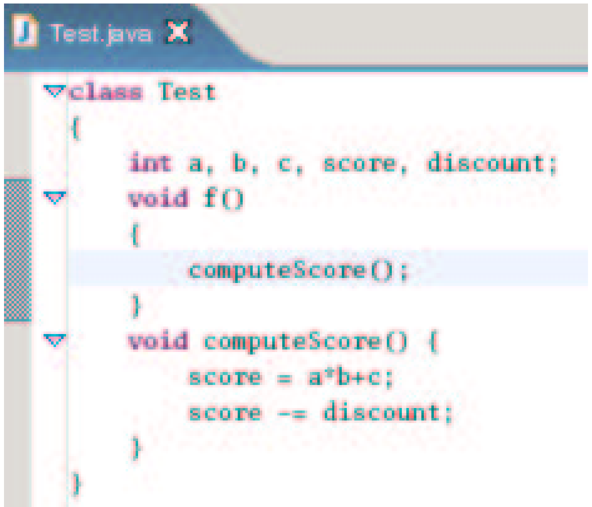
□ Generate Javadoc comment

□ Replace duplicate code fragments

Method signature preview:

```
void computeScore()
```

Example – extract method



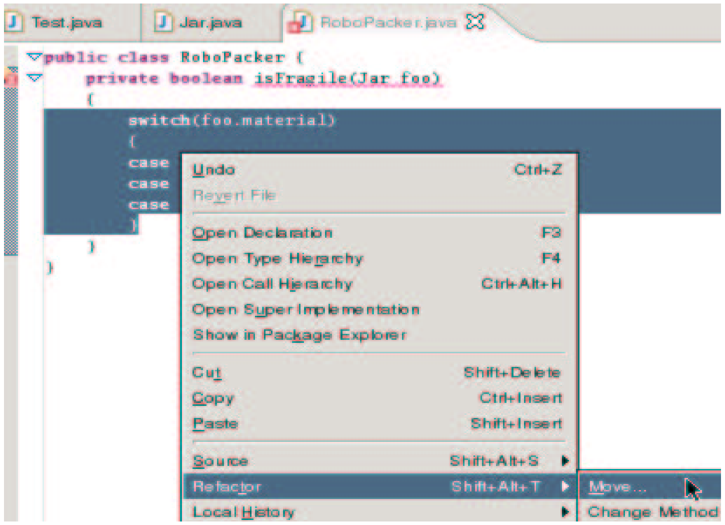
```
class Test
{
    int a, b, c, score, discount;
    void f()
    {
        computeScore();
    }
    void computeScore() {
        score = a*b+c;
        score -= discount;
    }
}
```

Example – move method

```
class Jar {
   ...
}
class RoboPacker {
   private bool isFragile(Jar foo) {
      switch(foo.material) {
      case GLASS: return true;
      case WOOD: return true;
      case TIN: return false;
      }
   }
}
```

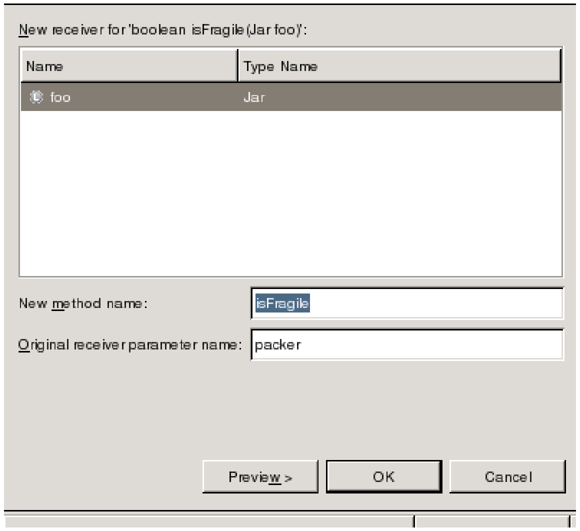
```
class Jar {
   bool isFragile() {
      switch(material) {
      case GLASS: return true;
      case WOOD: return true;
      case TIN: return false;
} } }
class RoboPacker {
   private bool isFragile(Jar foo) {
      return foo.isFragile();
   }
}
```

Example – move method



```
public class RoboPacker {
    private boolean isFragile(Jar foo)
    {
        switch(foo.material)
        {
        case
        case
        case
        }
    }
}
```

Example – move method



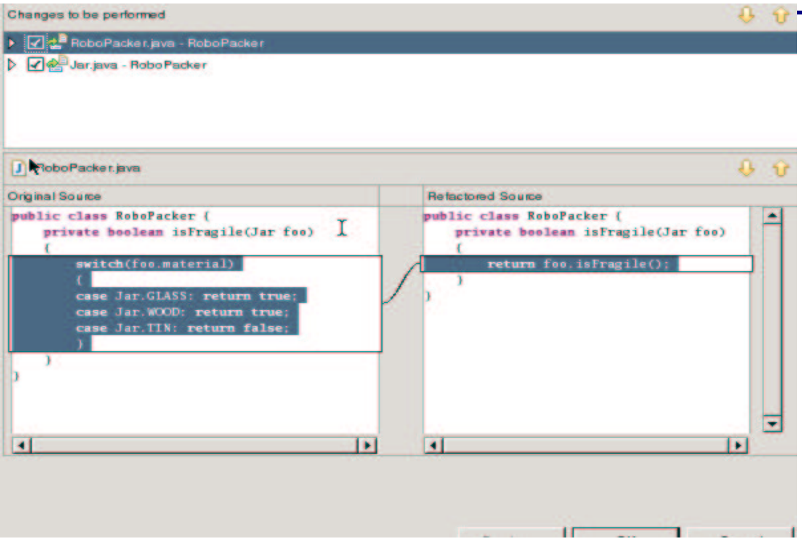
New receiver for 'boolean isFragile(Jar foo)':

Name	Type Name
foo	Jar

New method name: isFragile

Original receiver parameter name: packer

Example – move method



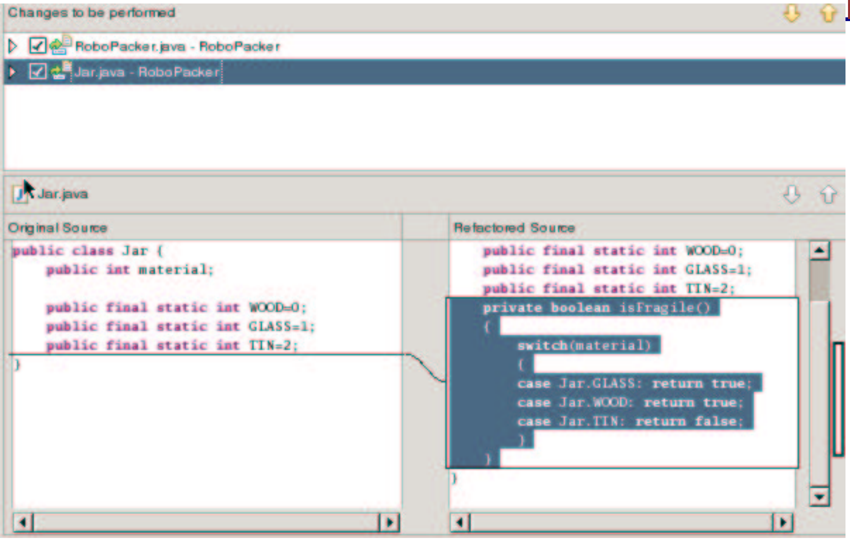
Original Source:

```
public class RoboPacker {
    private boolean isFragile(Jar foo)
    {
        switch(foo.material)
        {
        case Jar.GLASS: return true;
        case Jar.WOOD: return true;
        case Jar.TIN: return false;
        }
    }
}
```

Refactored Source:

```
public class RoboPacker {
    private boolean isFragile(Jar foo)
    {
        return foo.isFragile();
    }
}
```

Example – move method



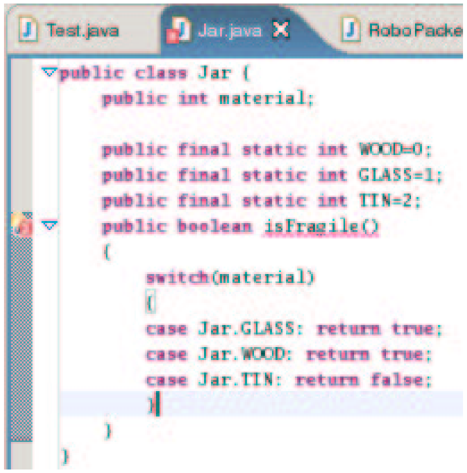
Original Source:

```
public class Jar {
    public int material;

    public final static int WOOD=0;
    public final static int GLASS=1;
    public final static int TIN=2;
}
```

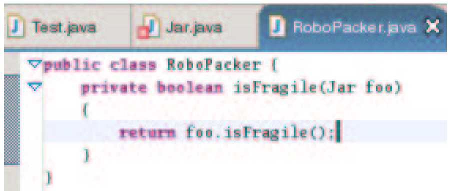
Refactored Source:

```
    public final static int WOOD=0;
    public final static int GLASS=1;
    public final static int TIN=2;
    private boolean isFragile()
    {
        switch(material)
        {
        case Jar.GLASS: return true;
        case Jar.WOOD: return true;
        case Jar.TIN: return false;
        }
    }
}
```

Example – move method



```
public class Jar {
    public int material;

    public final static int WOOD=0;
    public final static int GLASS=1;
    public final static int TIN=2;
    public boolean isFragile()
    {
        switch(material)
        {
        case Jar.GLASS: return true;
        case Jar.WOOD: return true;
        case Jar.TIN: return false;
        }
    }
}
```



```
public class RoboPacker {
    private boolean isFragile(Jar foo)
    {
        return foo.isFragile();
    }
}
```

Example – lift method

```
class Jar {
  bool isFragile() {
    switch(material) {
    case GLASS:
      // complex glass calculation
    case WOOD:
      // complex wood calculation
    case TIN:
      // complex tin calculation
} } }
```

```
class Jar {
  bool isFragile() {
    return material.isFragile();
} }

interface Material { ... }
class GlassMaterial:Material { ... }
class WoodMaterial:Material { ... }
class TinMaterial:Material { ... }
```

Questions?