Refactoring

Today's goals

- How do we reduce code complexity?
- What is refactoring, and how should we apply it?
- When should we refactor?

The Tale

"Once upon a time there was a Good Software Engineer whose customers knew exactly what they wanted. The Good Software Engineer worked very hard to design the Perfect System that would solve all the Customer's problems now and for decades. When the Perfect System was designed, implemented, and finally deployed, the Customers were very happy indeed. The Maintainer of the System had very little to do to keep the Perfect System up and running, and the Customers and the Maintainer lived happily ever after". [OO Reengineering Patterns]

The Reality: Things Change

*"The **only constant** in life is **change**"- Heraclitus.*

The Reality: Things Change

The Law of Change

"The only guaranteed Constant in this World is Change"

Kinds of change:

- requirements change
- design changes
- technological changes
- corrections
- social changes

The Reality: Complexity Kills

The Law of Increasing Complexity

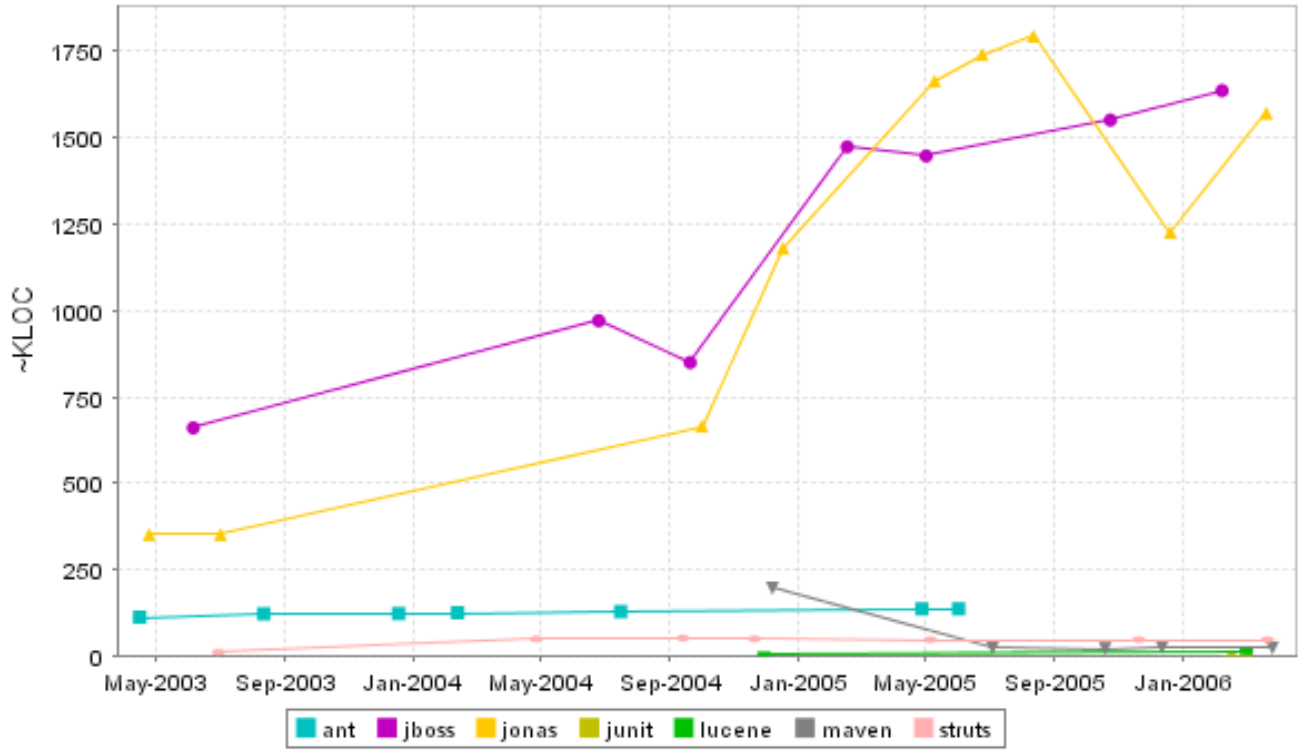
- a variation on the same theme: The Second Law of Thermodynamics

New York Times article about why Windows is so slow:

"**Complexity kills. It sucks the life out of developers**, it makes products difficult to plan, build and test, it introduces security challenges and it causes end-user and administrator frustration."

[Ray Ozzie, chief technical officer at Microsoft]

Complexity Increases with Time/Size [www.structure101.com]



The Solution: Refactoring

Rather than run from change we must cope with change

- **Refactoring = a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour**

It's a key practice in XP

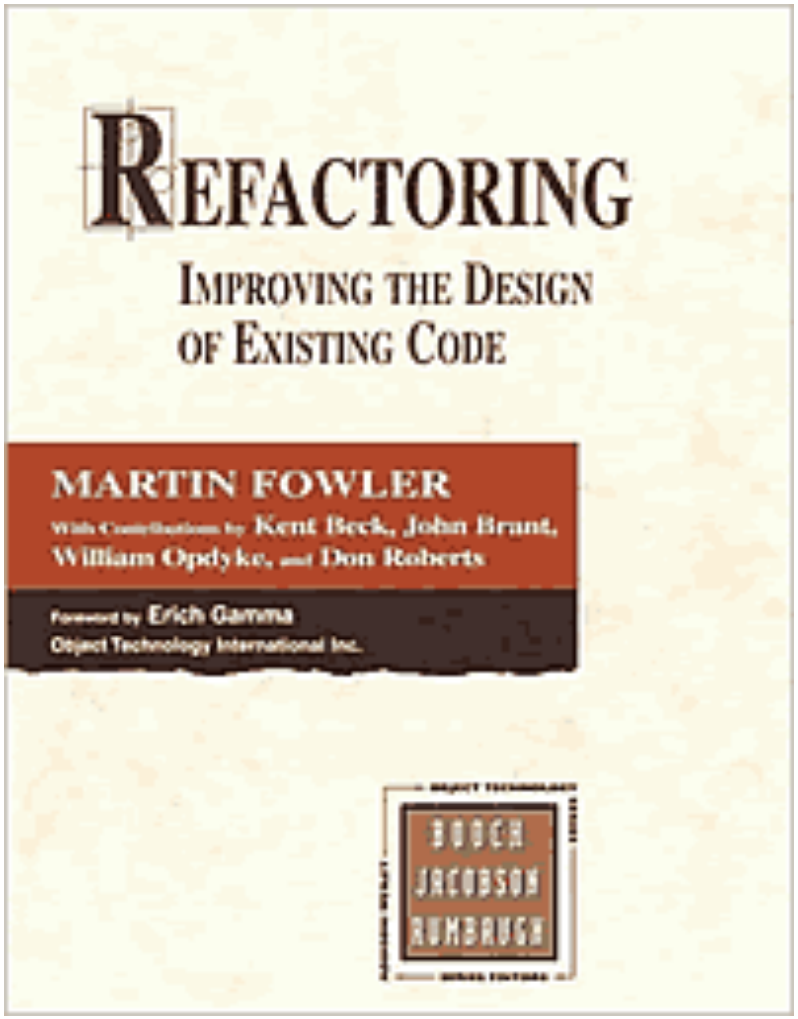
Which of the following problems could not be fixed by performing a refactoring?

- A) A loop that is too deeply nested
- B) A method that has a non-descriptive name
- C) A bug in the code
- D) A class that doesn't do very much
- E) A class that has poor cohesion

Refactoring

- Changing the structure of a program without changing its function
- Key part of XP
- Key part of software evolution
- Key part of making reusable software
- Commonly done, often in secret

Refactoring: Improving the Design of Existing Code



Martin Fowler with contributions by Kent Beck, John Brant, William Opdyke, and Don Roberts

Addison Wesley, 1999.

www.refactoring.com

Add Parameter

- A method needs more information from its caller.
  - Need to vary a constant
  - Forgot to include some information

Add Parameter

- Check superclasses and subclasses
- Make copy of old method, add parameter
- Change body of old method so that it calls new one – passing the previous constant as a parameter
- Find all references to the old method and change them to refer to the new
- Test should run after each change
- Remove old method

Lessons

- Refactoring requires tests.
- Refactoring requires software version control
- Knowing how to perform a refactoring makes it safer, easier, and faster
- Just because you can perform a refactoring doesn't mean you should

Introduce Parameter Object

- There is a group of parameters that naturally go together
  - Many methods have same parameters
  - Parameters are passed unchanged from one method to another
  - Method has too many parameters

Introduce Parameter Object

- Make a new class for the group of parameters
- Use Add Parameter for the new class. Use a new object for the parameter in all the callers.
- For each of the original parameters, ...

Introduce Parameter Object

- For each of the original parameters,
  - modify caller to store parameter in the new object and omit parameter from call
  - modify method body to omit original parameter and to use the value stored in the new parameter
  - if method body calls another method with parameter object, use existing parameter object instead of making a new one

```
class Account …
  double getFlowBetween (Date start, Date end) {
    double result = 0;
    Enumeration e = _entries.elements();
    while (e.hasMoreElements()) {
       Entry each = (Entry) e.nextElement();
       Date date = each.getDate();
       if (date.equals(start) || date.equals(end)
  ||
          (date.after(start) && date.before(end)))
       {
          result += each.getValue();
       }}
  return result;
}
```

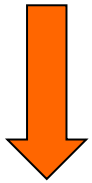
```
class DateRange {
  private final Date _start;
  private final Date _end;

  DateRange (Date start, Date end) {
    _start = start;
    _end = end;
  }
  Date getStart() {
    return _start;
  }
  Date getEnd() {
    return _end;
  }
}
```

```
Class Account …
   double getFlowBetween (Date start, Date end,
                                        DateRange range) {
      double result = 0;
      Enumeration e = _entries.elements();
      while (e.hasMoreElements()) {
         Entry each = (Entry) e.nextElement();
         Date date = each.getDate();
         if (date.equals(start) || date.equals(end) ||
            (date.after(start) && date.before(end)))
         {
            result += each.getValue();
         }}
   return result;
}
```

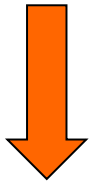
Changing callers

```
Double flow =
  anAccount.getFlowBetween(startDate,
  endDate);
```



```
Double flow =
  anAccount.getFlowBetween(startDate,
  endDate, new DateRange(null,null))
```

```
Double flow =
  anAccount.getFlowBetween(startDate,
  endDate, new DateRange(null,null))
```

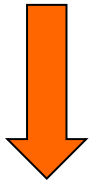


```
Double flow =
  anAccount.getFlowBetween(endDate, new
  DateRange(startDate,null))
```

```
class Account …
   double getFlowBetween (Date end, DateRange range){
     double result = 0;
     Enumeration e = _entries.elements();
     while (e.hasMoreElements()) {
        Entry each = (Entry) e.nextElement();
        Date date = each.getDate();
        if (date.equals(range.getStart()) ||
   date.equals(end) ||
           (date.after(range.getStart()) &&
   date.before(end)))
        {
           result += each.getValue();
        }}
   return result;
}
```

```
class Account …
   double getFlowBetween (DateRange range) {
      double result = 0;
      Enumeration e = _entries.elements();
      while (e.hasMoreElements()) {
         Entry each = (Entry) e.nextElement();
         Date date = each.getDate();
         if (date.equals(range.getStart()) ||
            date.equals(range.getEnd()) ||
            (date.after(range.getStart()) &&
               date.before(range.getEnd())))
         {
            result += each.getValue();
         }}
return result;
```

```
Double flow =
  anAccount.getFlowBetween(endDate, new
  DateRange(startDate,null))
```



```
Double flow = anAccount.getFlowBetween(new
  DateRange(startDate,endDate))
```

Introduce Parameter Object

After introducing a parameter object, look to see if code should be moved to its methods.

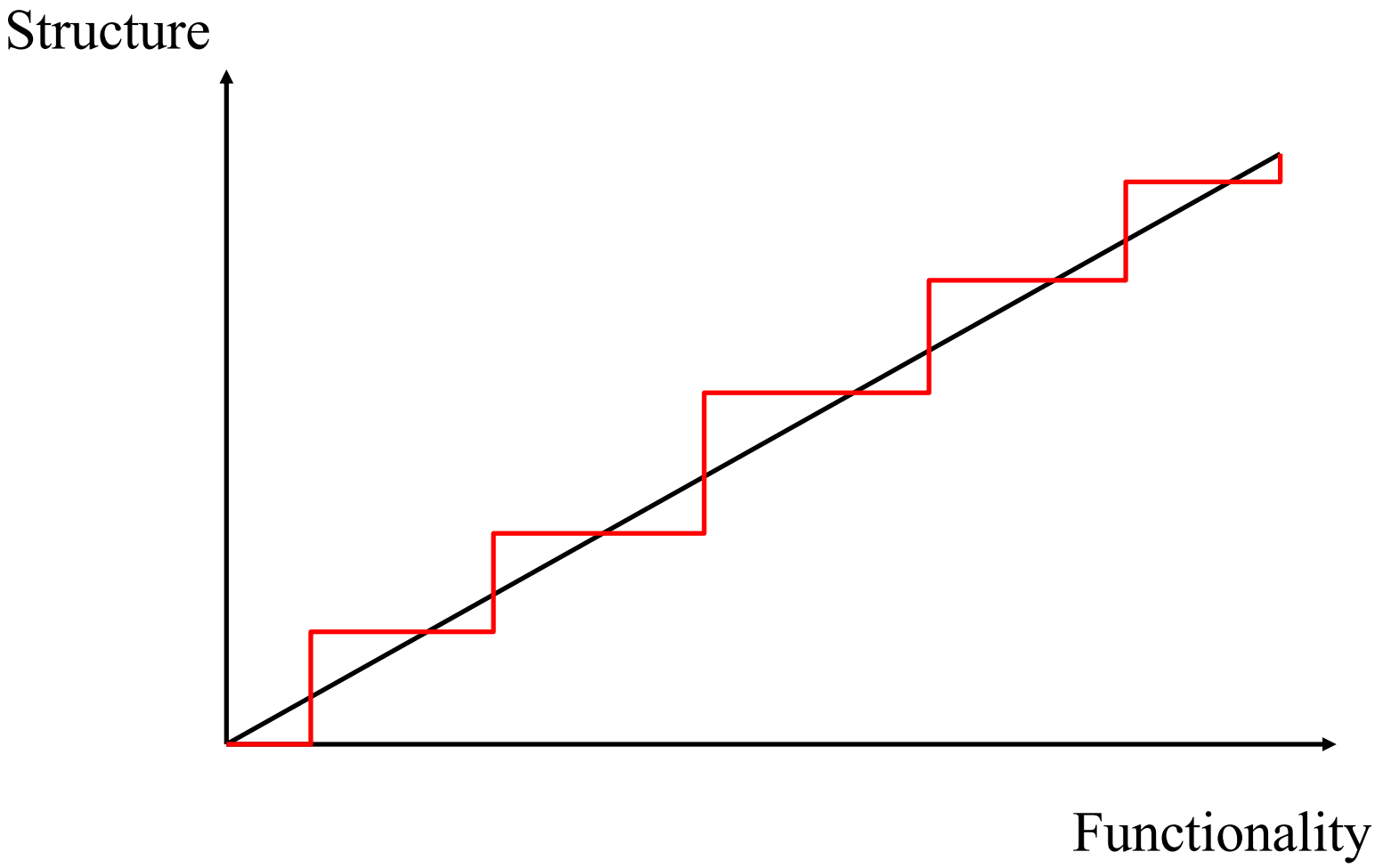
```
class DateRange …
boolean includes (Date arg) {
   return (arg.equals(_start) ||
     arg.equals(_end) || (arg.after(_start)
     && arg.before(_end)));
}
```

```
class Account …
  double getFlowBetween (DateRange range) {
    double result = 0;
    Enumeration e = _entries.elements();
    while (e.hasMoreElements()) {
      Entry each = (Entry) e.nextElement();
      if (range.includes(each.getDate()))
      {
        result += each.getValue();
      }}
    return result;
  }
```

Lessons

- Refactorings should be small.
- Check after each step to make sure you didn't make a mistake.
- One refactoring leads to another.
- Major (architectural) change requires many refactorings.

Refactorings separate Structure from Functionality



When should you refactor?

- Separate changing behavior from refactoring
  - changing behavior requires new tests
  - refactoring must pass all tests
- Only refactor when you need to
  - before you change behavior
  - after you change behavior
  - to understand

The question "Are your methods shorter than 50 lines?" could be a good one to use in the Grading Rubric. What refactoring should you perform to eliminate this problem?

- A) Move method
- B) Extract method
- C) Change method signature
- D) Rename method
- E) Push up method

Refactorings

- Composing methods
  - extract method
  - inline method
  - inline temporary variable
  - introduce explaining variable
  - replace method with method object

Refactorings

- Moving features between objects
- Organizing data
- Simplifying conditional expressions
- Making method calls simpler
- Generalization

Automated support for refactoring

- Deciding where to refactor
  - tools for measuring cohesion, size, etc
  - tools for measuring code duplication - DupLoc
- Performing the change
  - Automated refactoring tools are always language-specific
  - Eclipse JTD one of the best for Java
  - Photran the best for Fortran

What refactoring will create getters and setters for a chosen field?

- A) Rename field
- B) Inline temporary variable
- C) Move field
- D) Encapsulate field
- E) Convert variable into field

XP rules

- Make it work - make tests run
- Make it right
  - Meaningful - easy to understand
  - As simple as possible
  - Eliminate duplication
- Make it fast

Refactoring and Design

- XP creates a design by refactoring
- Refactor parts of the system that are hard to understand
- Eventually nothing is hard to understand
- Code matches the design
- Refactoring is a good way to understand a complex system

Code Smells

- Duplicated code
- Long method
- Large class
- Long parameter list
- Message chain
- Feature envy

- Data class
- Switch statements
- Speculative generality
- Temporary field
- Refused bequest

Eliminating Duplicate code

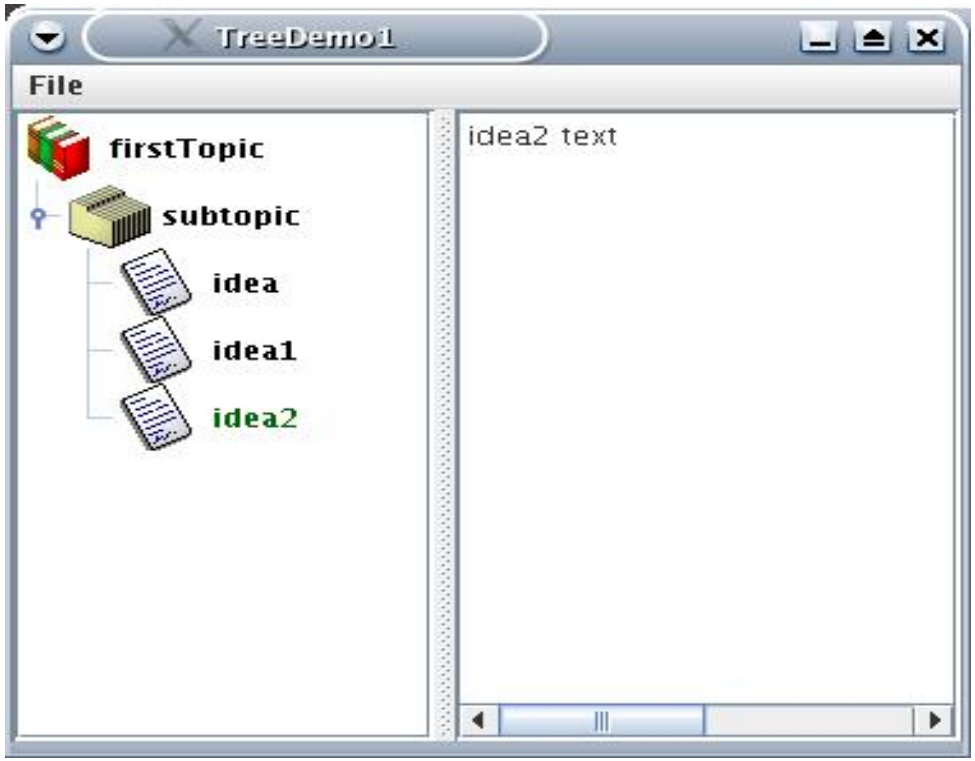
- Duplicate methods in subclasses
  - Move to superclass, possibly create superclass
- Duplicate expressions in same class
  - Extract method
- Duplicate expressions in different classes
  - Extract method, move to common component

Eliminating Switch statement

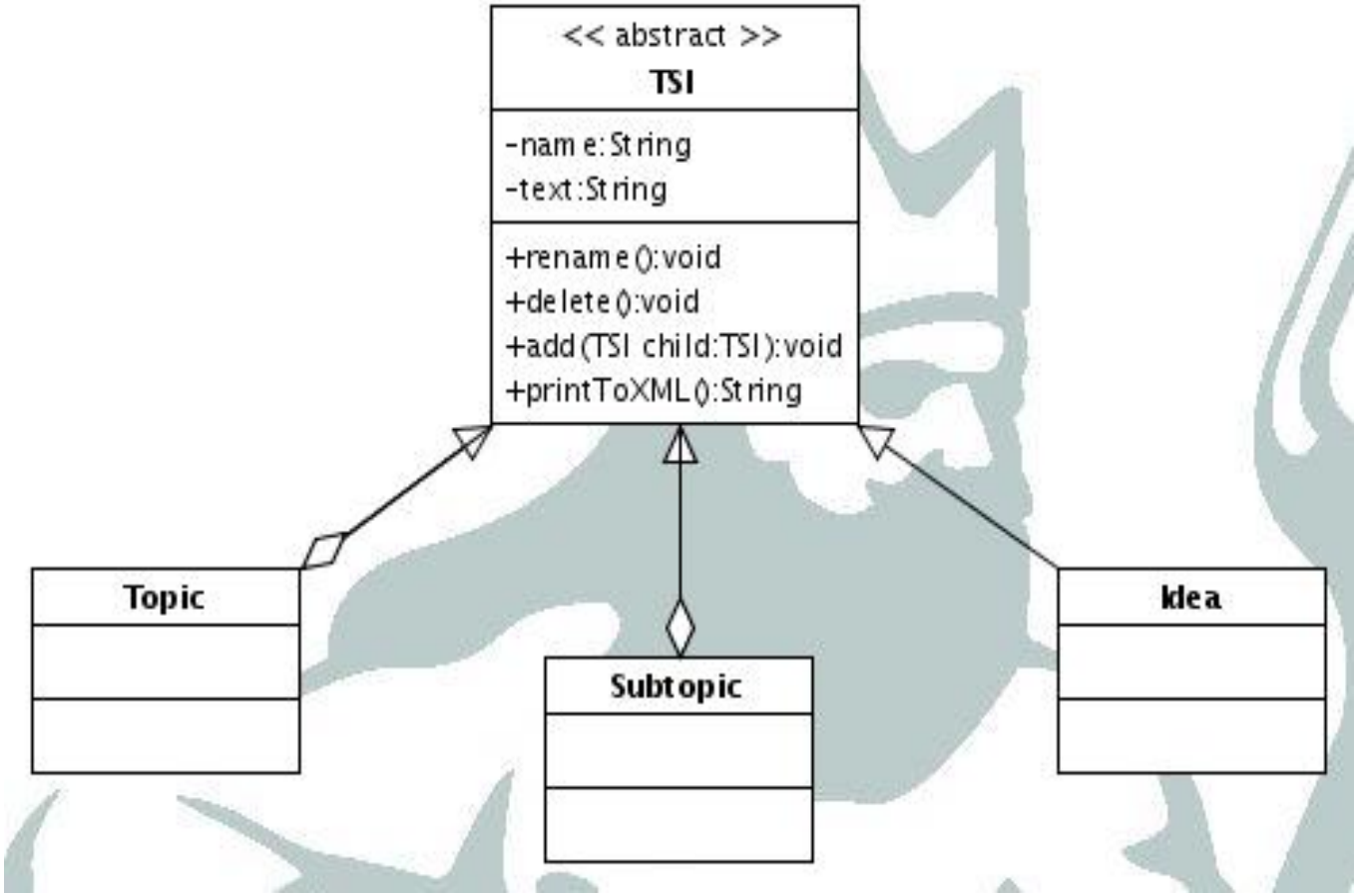
- Replace switch (or nested if) with a method call
- Make subclass for each case

IdeaIncubator

IdeaIncubator is a tool for organizing all kinds of things (mainly thoughts/ideas).



IdeaIncubator's UML diagram



Example (cont)

```
public abstract class TSI {
.....
        public void printToXML(){
                if (getType() == "Topic"){
                        // print info specific to a Topic
                } else if (getType() == "Subtopic"){
                        // print info specific to Subtopic
                } else if (getType() == "Idea"){
                         // print info specific to Idea
                }
         }
      }
```

Q: What's the problem with type checking(conditional) code? (see Open-Closed Principle)

Example resolved

Solution: Replace conditional code (type checking) with polymorphism.

Choose one of the subclasses and create a method which overrides the conditional one . Copy the body of that branch into the new method and adjust it.

```
public class Topic extends TSI {
.....
public void printToXML(){
      // print info related to Topic
  }
}
```

Summary

- Separate refactorings from changes to behavior
- Refactor in small steps
- Refactor every day
- Practice!

Refactoring with Intellij IDEA

- https://www.jetbrains.com/help/idea/tutorial-introduction-to-refactoring.html
- https://www.jetbrains.com/help/idea/refactoring-source-code.html#popular-refactorings

Coming next

- April 2: Grad students present progress in grad project
- April 5: Undergrad project M5
  - Show us refactoring examples (code snapshots before and after)
  - Extend the core Object Business layer