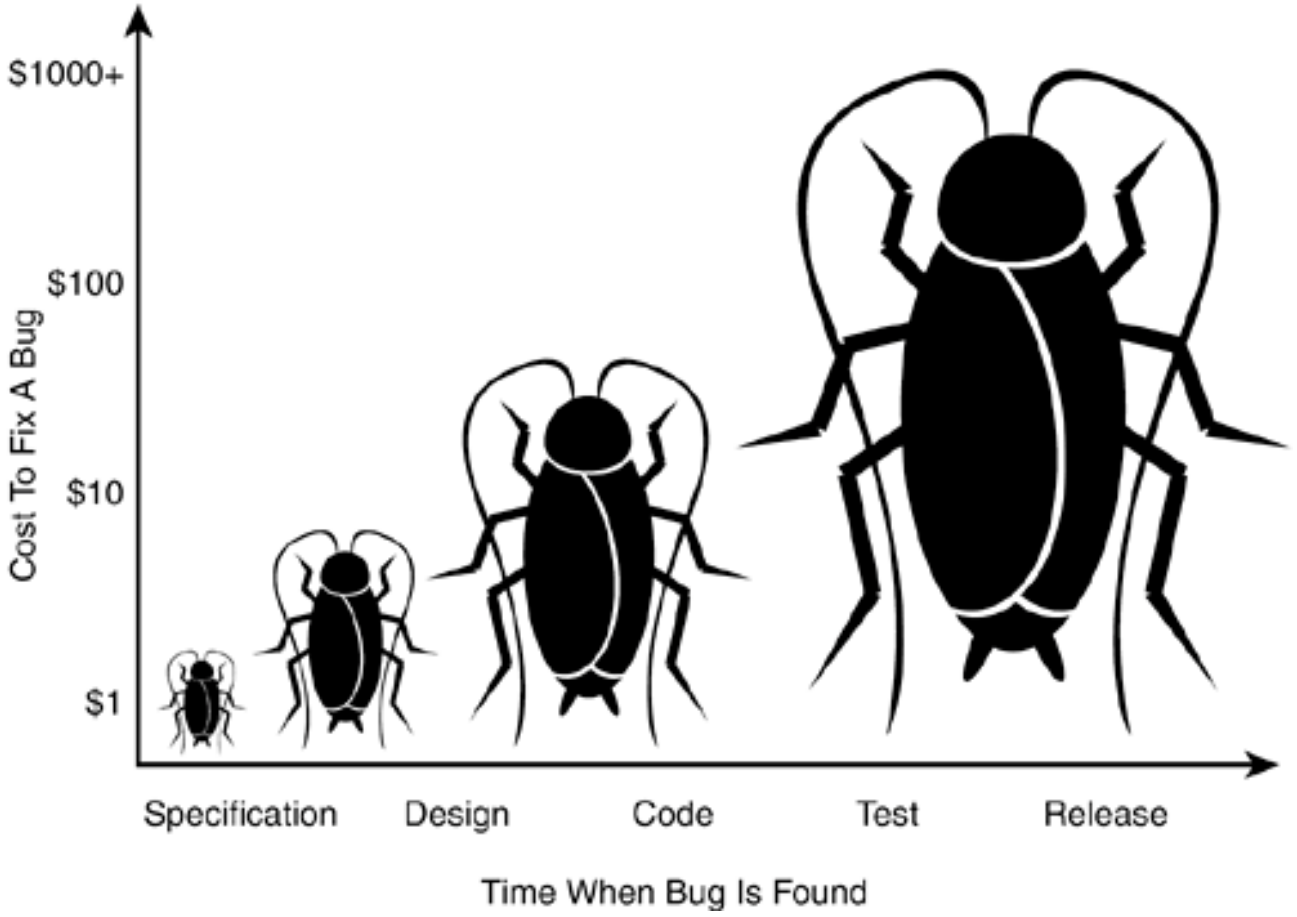
Object Oriented Design and Analysis
CPE 372

Lecture 11

Refactoring

Dr. Sally E. Goldin
Department of Computer Engineering
King Mongkut's University of Technology Thonburi
Bangkok, Thailand

Why Design?



Find problems and resolve issues as early as possible in the development process

Want to design systems with:

High cohesion within modules

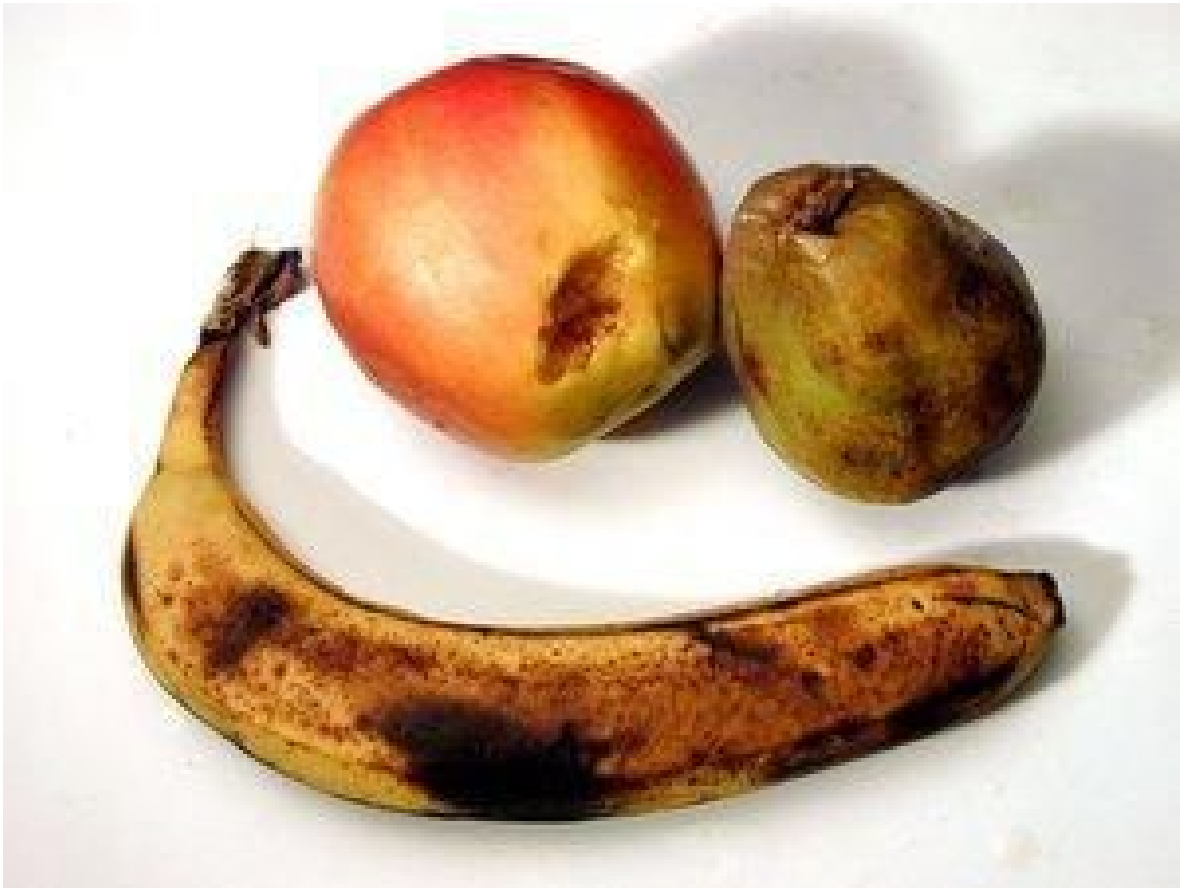
Each module has a limited number of clearly defined functional responsibilities

Low coupling between modules

Communication and dependencies between modules are minimized to the greatest extent possible

Software with these characteristics will be easier to build, understand, reuse, extend and change

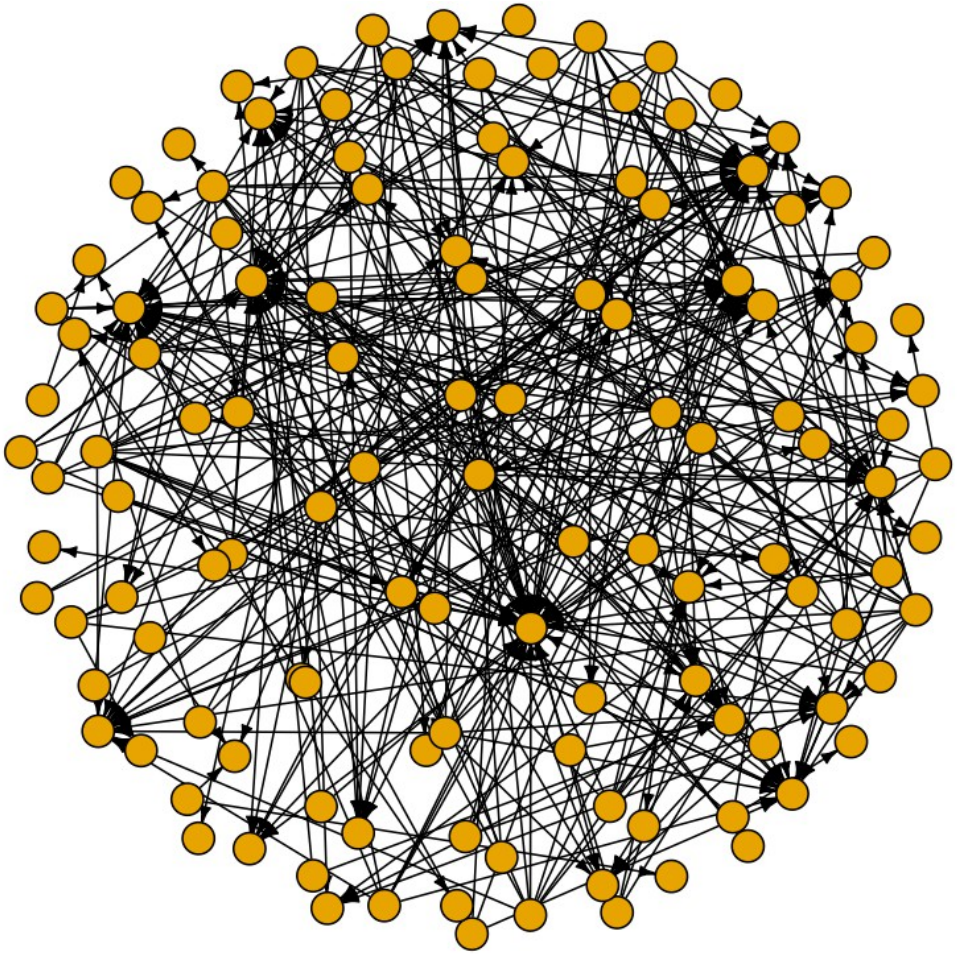
Over time, software rots!

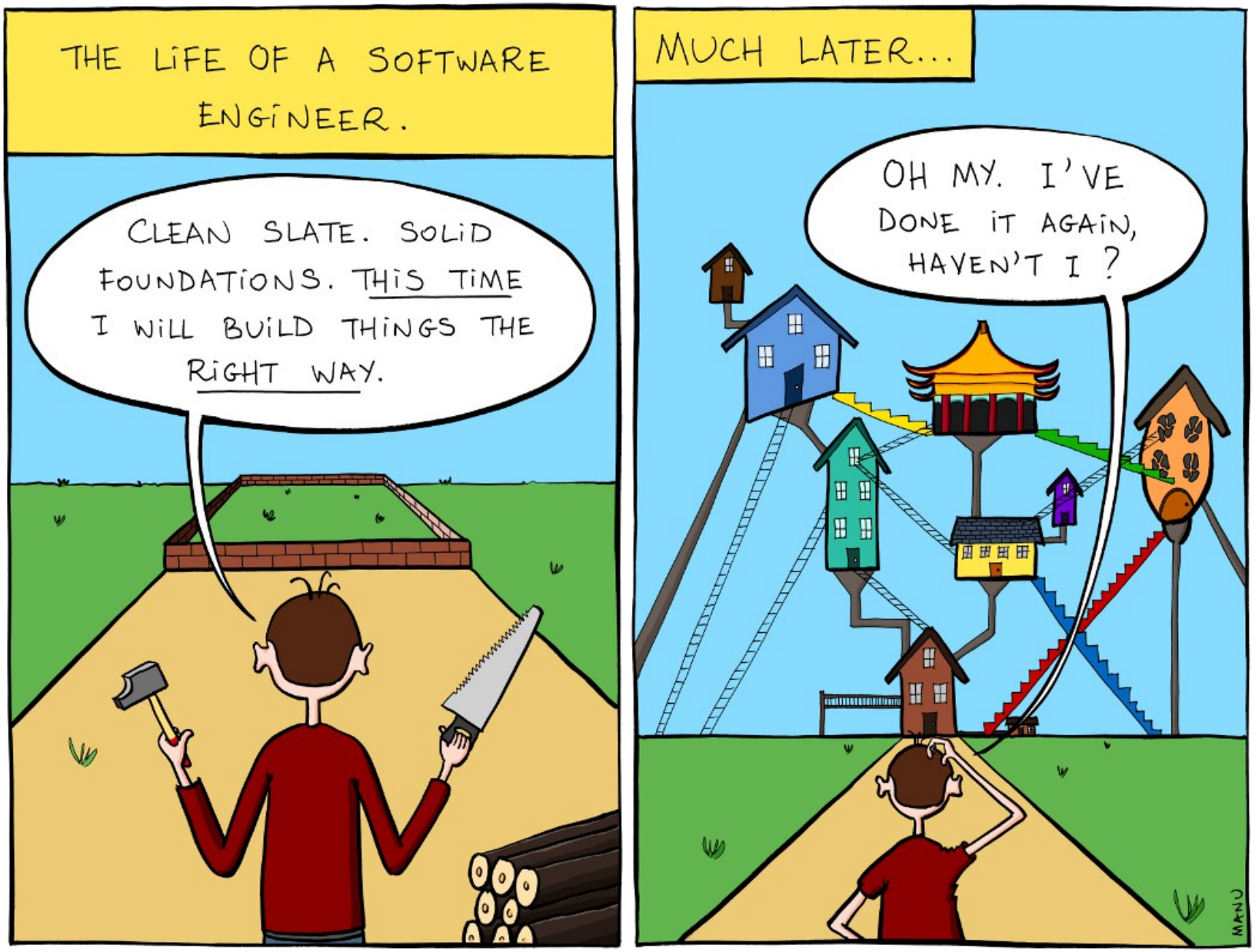


Software changes or extensions *reduce cohesion*, introduce *undesirable dependencies,* and *muddy the distinctions* in the original design.

Dependency Graph of Mature Software

From: http://www.designsmells.com/articles/does-your-architecture-smell/

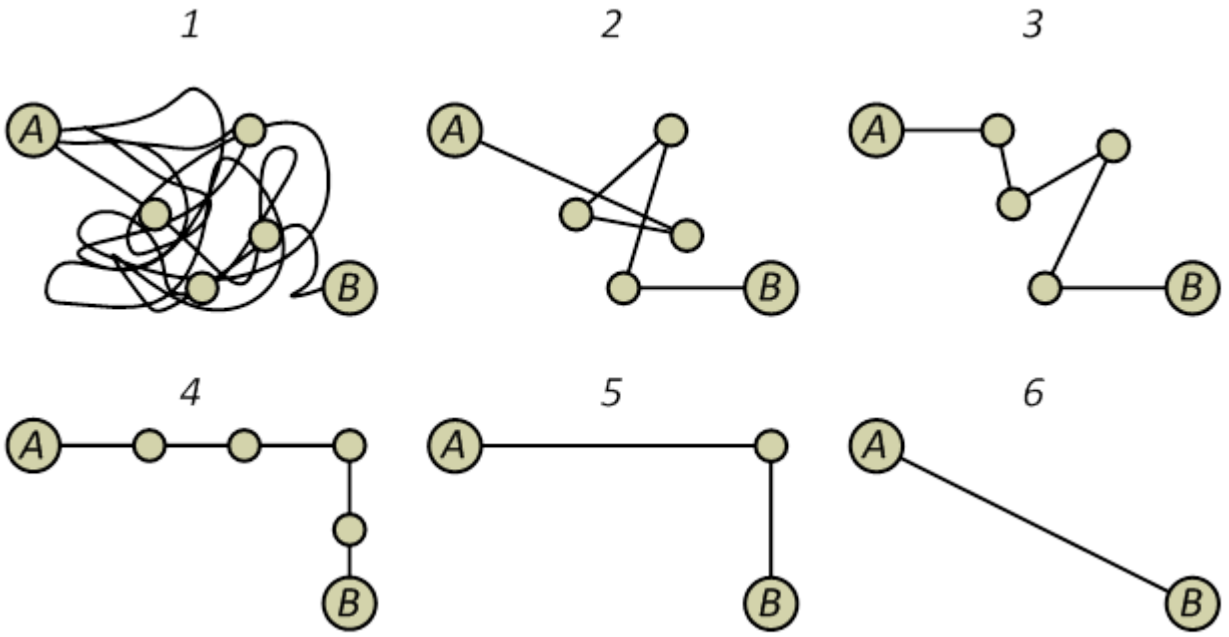



THE LIFE OF A SOFTWARE ENGINEER.
CLEAN SLATE. SOLID FOUNDATIONS. THIS TIME I WILL BUILD THINGS THE RIGHT WAY.
MUCH LATER...
OH MY. I'VE DONE IT AGAIN, HAVEN'T I ?
MANU

Refactoring can help

What is ***refactoring***?

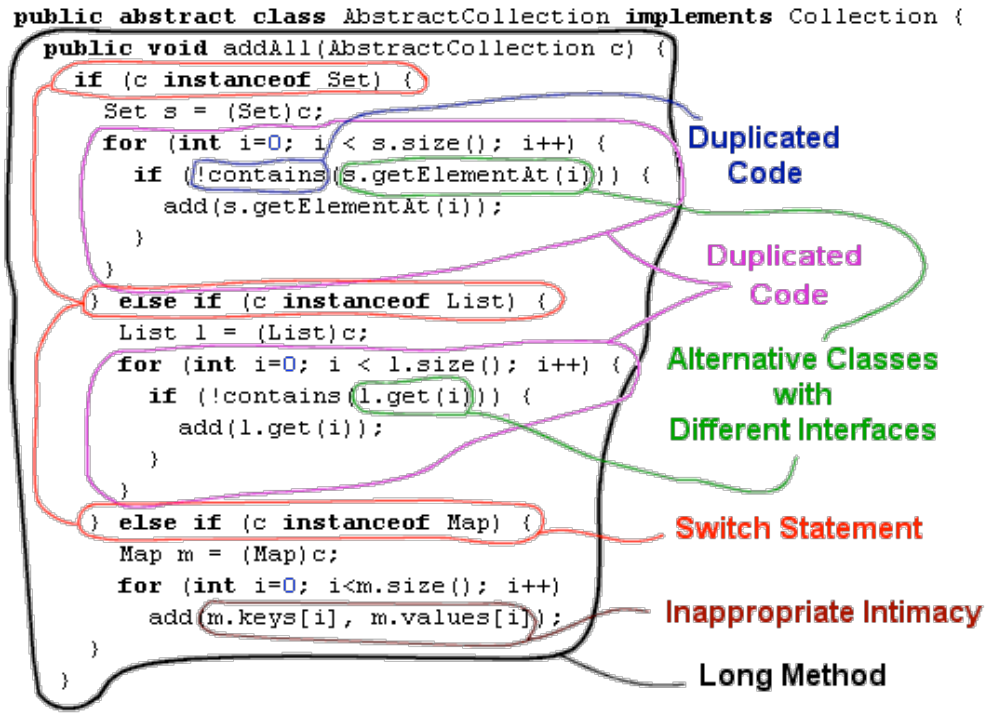
Changing the design of existing code without modifying its behavior



Refactoring can occur at many levels

Method level

- Splitting overly long methods
- Extracting common code into new methods
- Eliminating redundant code
- Increasing consistency in code structure

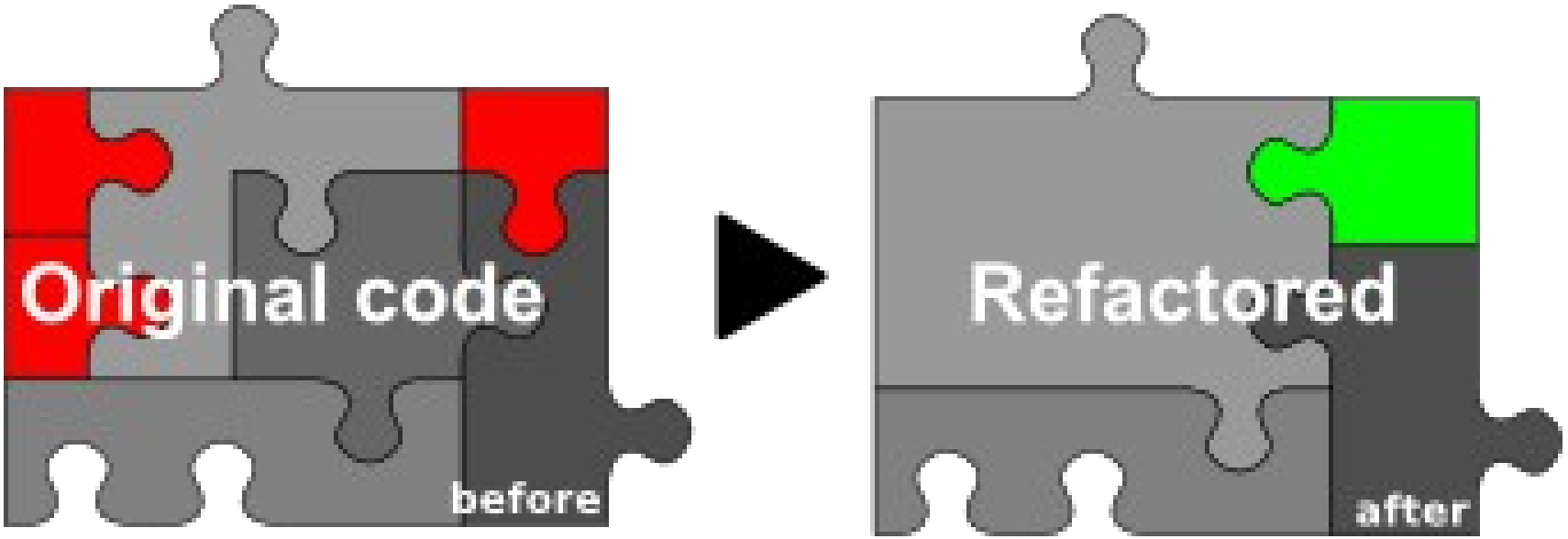


Class level

- Splitting class functionality into multiple classes
- Extracting common functionality into a superclass
- Pushing functionality into subclasses
- Moving functionality from one class to another
- Adding new classes or interfaces to reduce dependencies (e.g. adapters, facades)

Module/Package level

- Moving some classes into a new package
- Creating a superclass or interface in one package to be implemented in another package



ShapeReader.java (Exercise 4) – my first attempt

```
public AbstractShape readShape()
    {
     AbstractShape newShape = null;
     boolean bError = false;
     String line;
     do
     {
         line = getNextLine();
         if (line != null)
         {
             String fields[] = line.split(" ");
                newShape = null;
          if (fields.length >= 4)  /* should be at least four fields */
          {
              if (fields[0].equalsIgnoreCase("TRIANGLE"))
              {
               if (fields.length == 7)
                   /* commmand plus 3 x,y points */
               {
                   int[] x = new int[3];
                   int[] y = new int[3];
                   for (int i=0; i < 3; i++)
                   {
                      x[i] = convertToInt(fields[i*2+1]);
                      y[i] = convertToInt(fields[i*2+2]);
                      if ((x[i] < 0) || (y[i] < 0))
                      {
                       System.out.println("\t\tInvalid integer in triangle specification");
                      }
                   }
                   newShape = new Triangle(x[0],y[0],x[1],y[1],x[2],y[2]);
               }
              }
              else if (fields[0].equalsIgnoreCase("SQUARE"))
                 {  ....
```

Refactored to shorten and reduce repetition

```
  public AbstractShape readShape()
     {
     AbstractShape newShape = null;
     boolean bError = false;
     String line;
     do
     {
         line = getNextLine();
         if (line != null)
         {
         newShape = parseCheckShapeCommand(line);
         if (newShape == null)
             System.out.println("\t\tBad line: '" + line + "' ==>
Skipping");
         }
     } while ((newShape == null) && (line != null));
     return newShape;
```

1) Pulled out all the code to check input and create shapes, and moved to **parseCheckCommand()**

More method refactoring

```
private AbstractShape parseCheckShapeCommand(String line)
 {
  AbstractShape newShape = null;
  String fields[] = line.split(" ");
  if (fields.length >= 4)  /* should be at least four fields */
  {
      if (fields[0].equalsIgnoreCase("TRIANGLE"))
      {
       newShape = createTriangle(fields);
      }
      else if (fields[0].equalsIgnoreCase("SQUARE"))
      {
       newShape = createSquare(fields);
      }
      else if (fields[0].equalsIgnoreCase("DIAMOND"))
      {
       newShape = createDiamond(fields);
      }
      else if (fields[0].equalsIgnoreCase("CIRCLE"))
      {
       newShape = createCircle(fields);
      }
      else /* not a valid line */
      {
       System.out.println("\t\tInvalid shape command");
      }
  }
  else
  {
      System.out.println("\t\tLine has too few fields");
  }
  return newShape;  /* could be null */
 }
```

2) Separated the parsing from the shape creation

Advantages of this refactoring

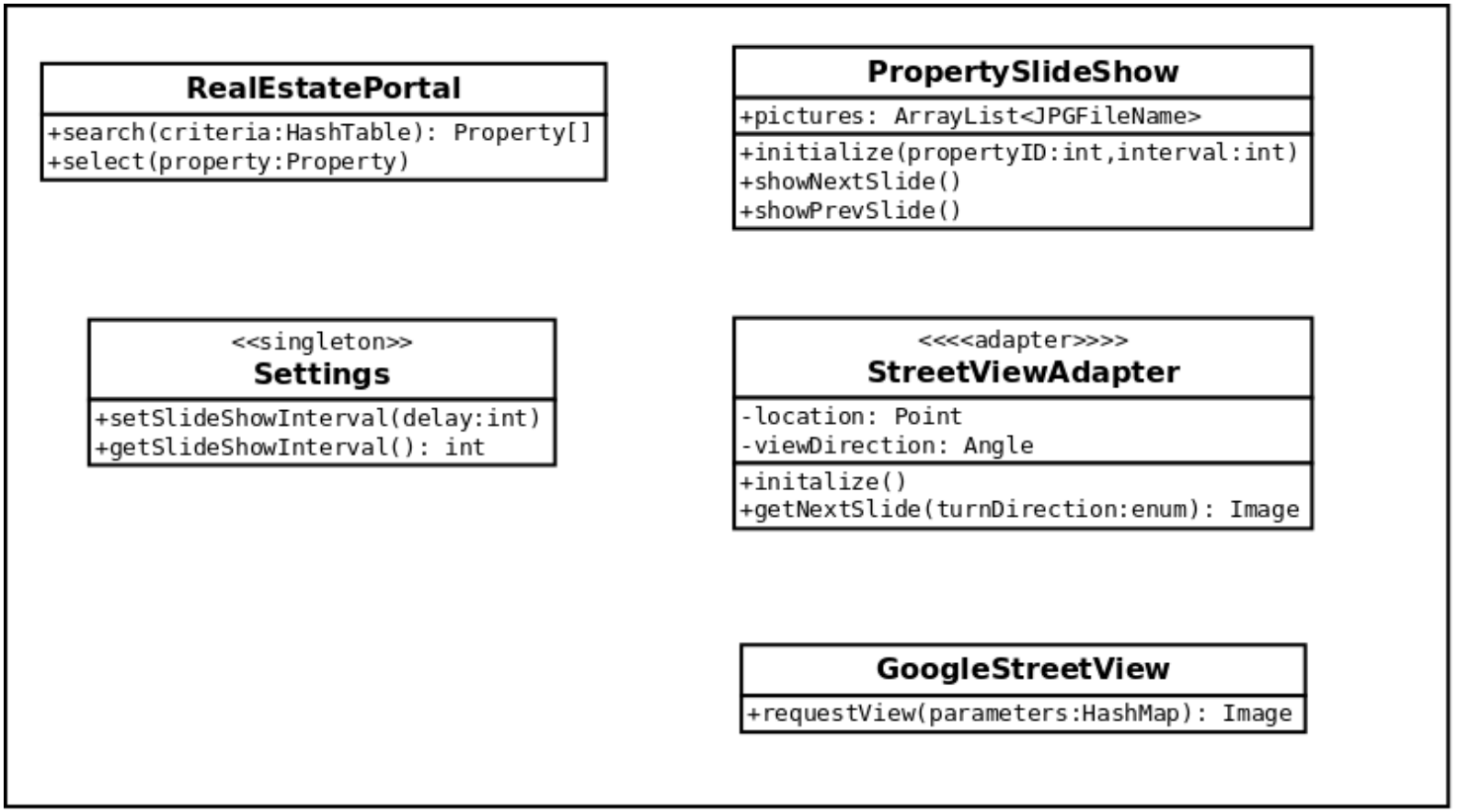
- ✓ Methods are much shorter, easier to understand
- ✓ Simple to add new shapes

Compare the code (in *demos/Lecture11*):

OriginalShapeReader.java

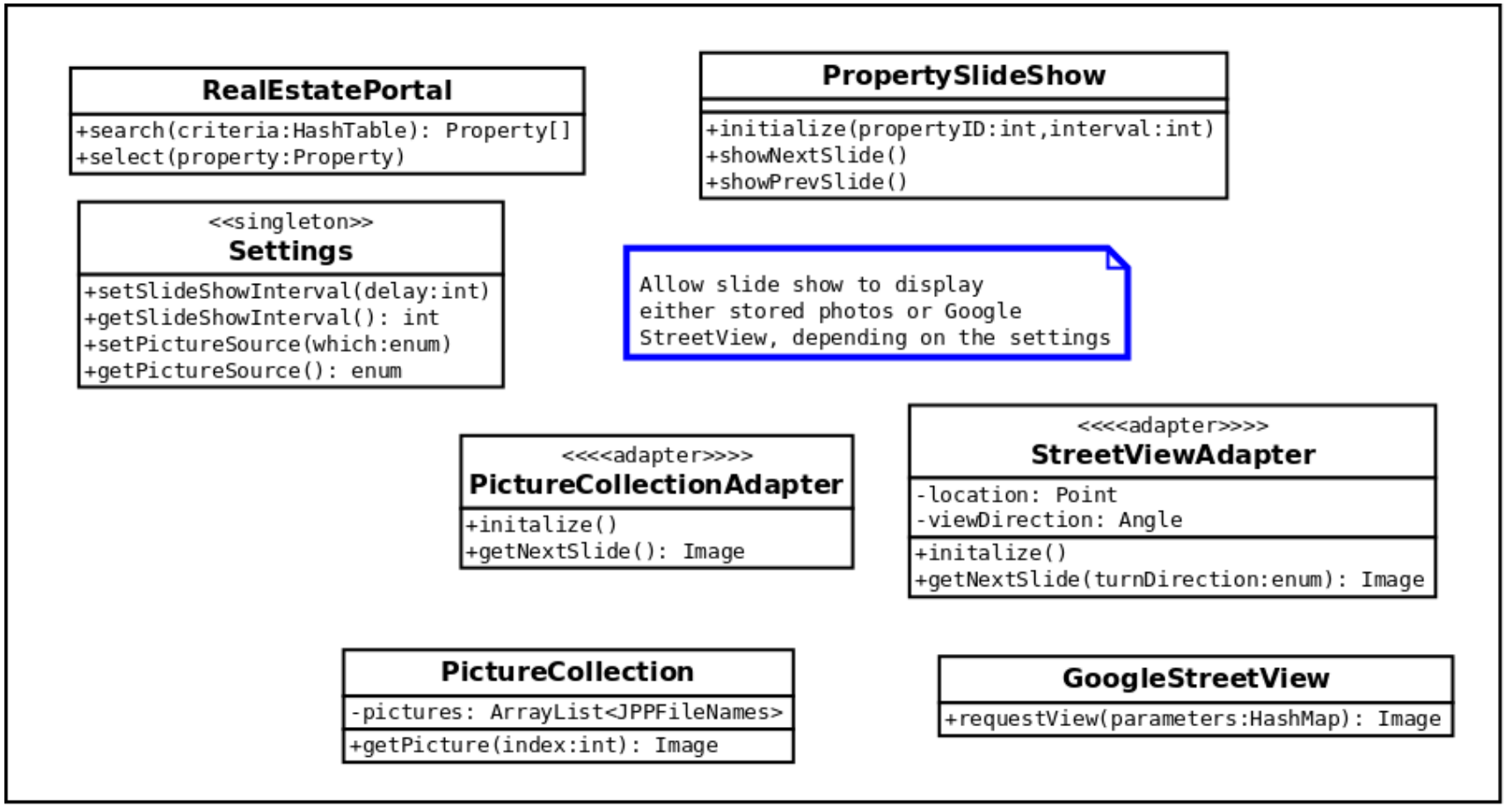
RefactoredShapeReader.java

Class Level Refactoring Example



This design has the undesirable feature that *PropertySlideShow* needs to do very different things depending on whether the source of the images is local photos or StreetView

After Refactoring



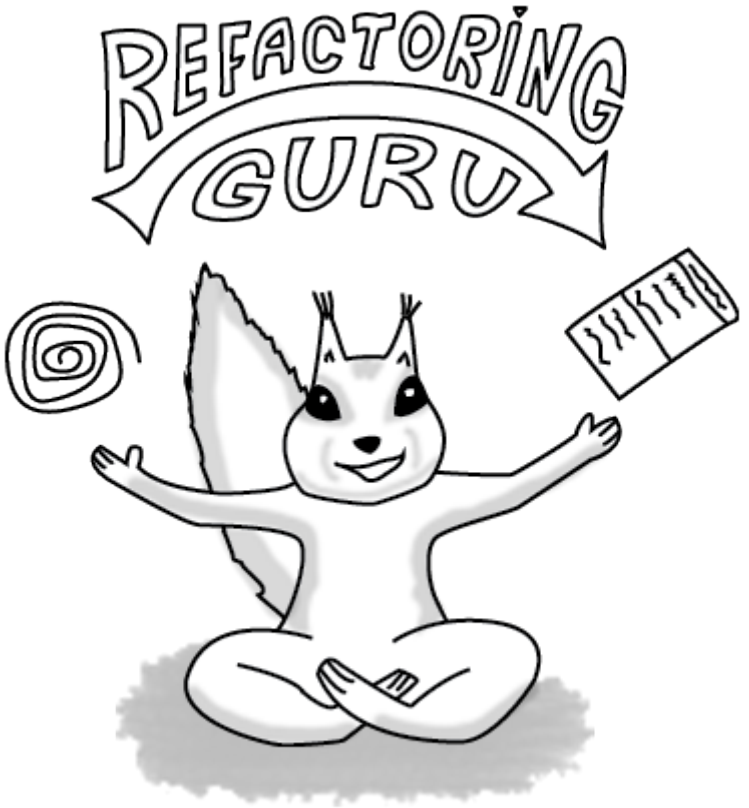
We have split out the *PhotoCollection* class from *PropertySlideShow* and added an adapter for local image display that parallels the adapter class for StreetView

How do you know that you need to refactor?

Watch out for tell-tale problems: "code smells" and "design smells"

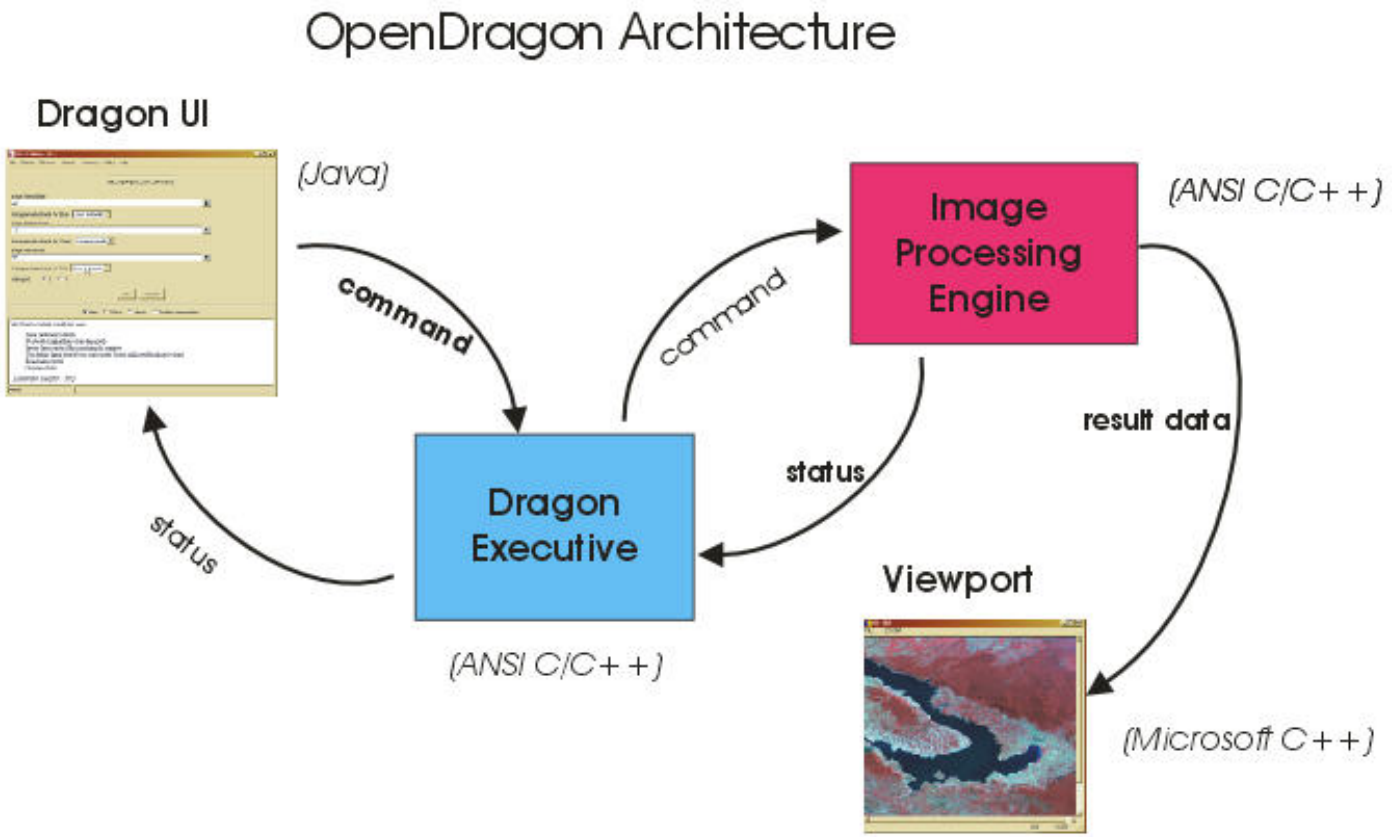
For a list of common "smells" see https://en.wikipedia.org/wiki/Code_smell

Experience is the best teacher...



So... we are going to do an exercise with a real software system that truly needs to be refactored!

OpenDragon Architecture



Design and implementation of Java-based UI started in 2000,continued through 2006!

Less code rot than in some systems because only one developer (me!)

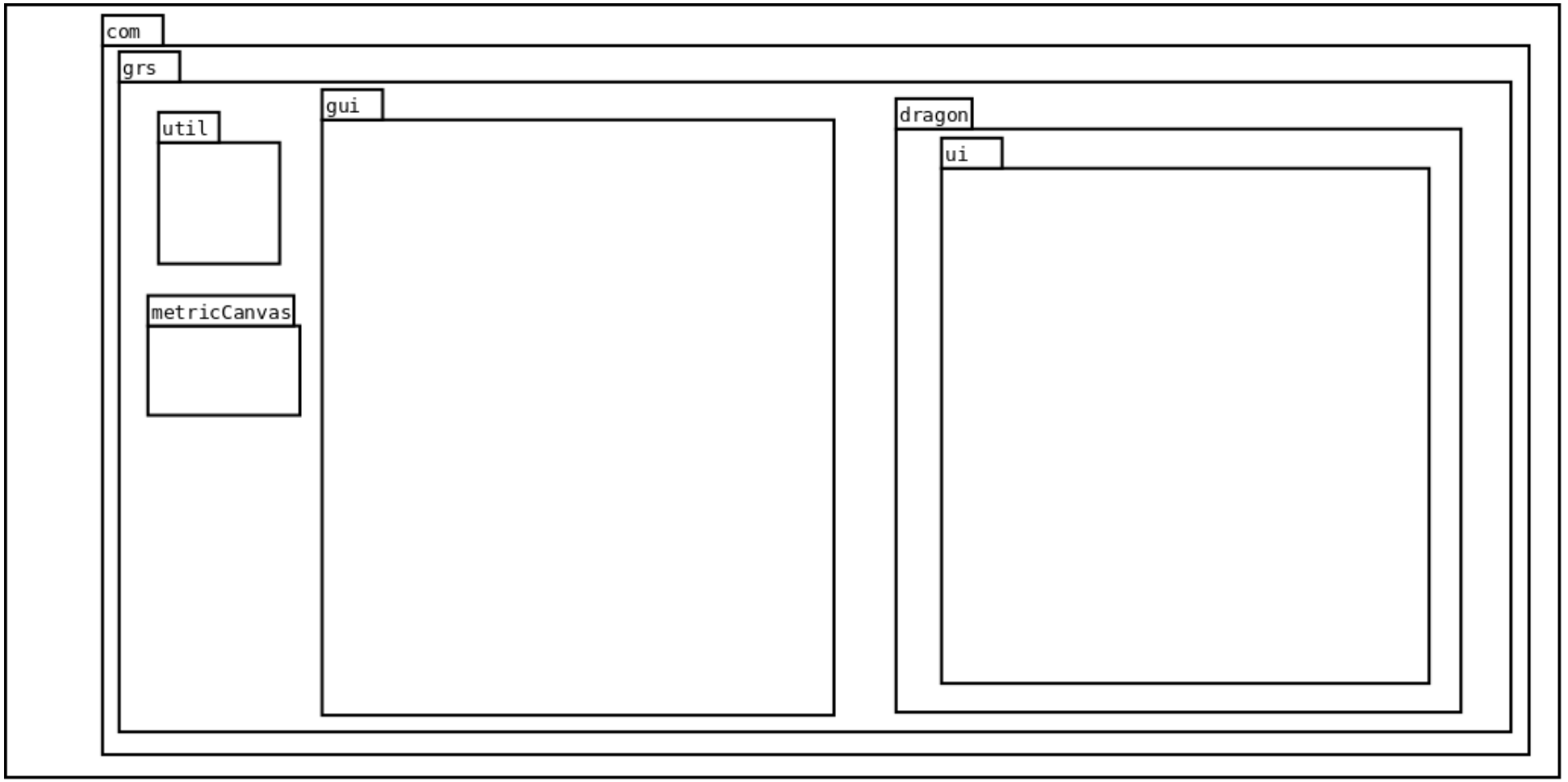
Design Goals

- **Instead of hard-coding the UI screens, develop a system that can generate the screens at run time based on a descriptive (XML) representation**
- **The entire UI must be internationalized**
- **UI generation system should be able to be applied to any application, not just OpenDragon**

Demonstrations

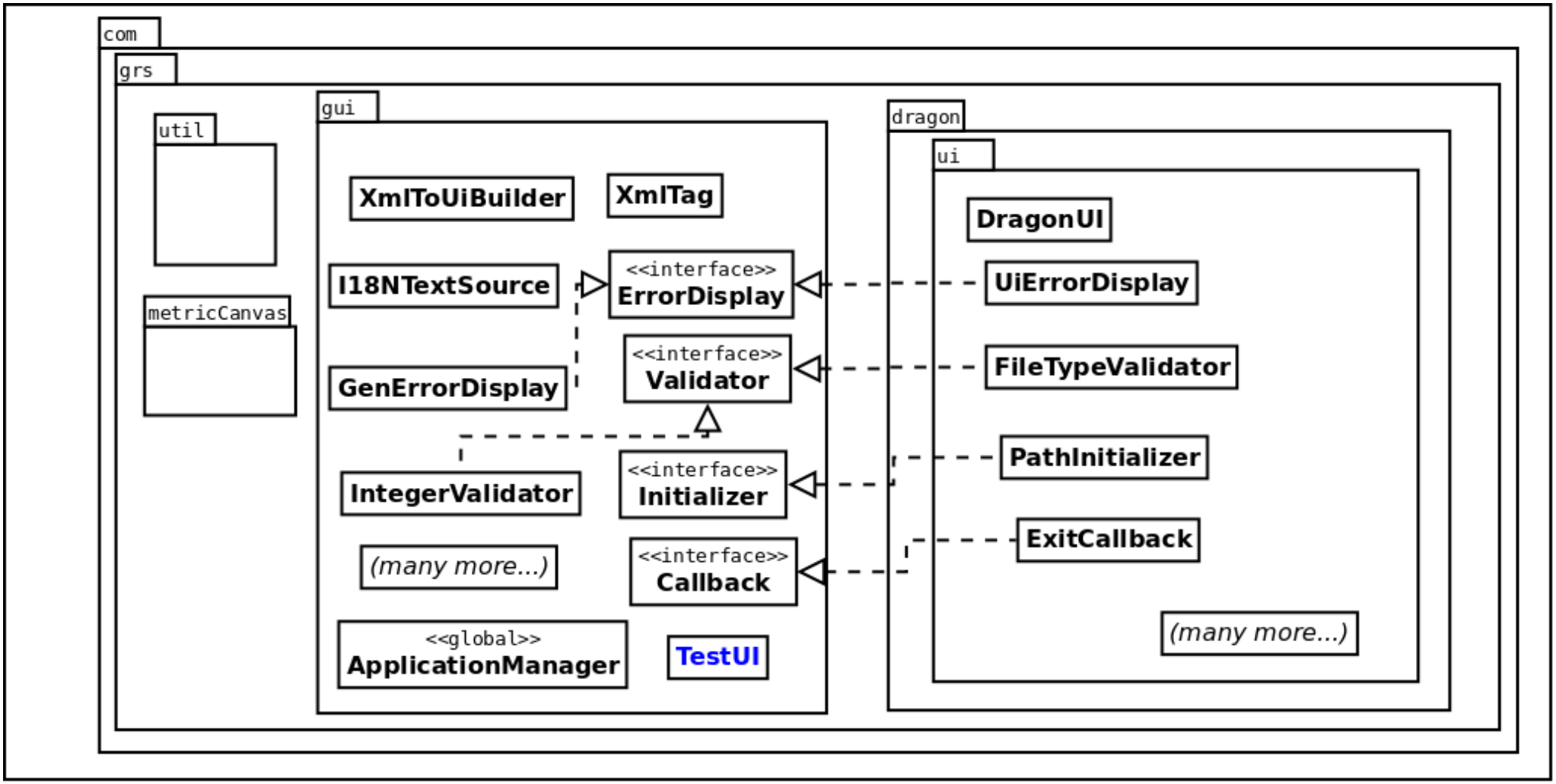
- Overview of the source tree
- Building the XmlToUiBuilder test application
- Generating and displaying a user interface
- Exploring the XML descriptions

Partial Package Diagram

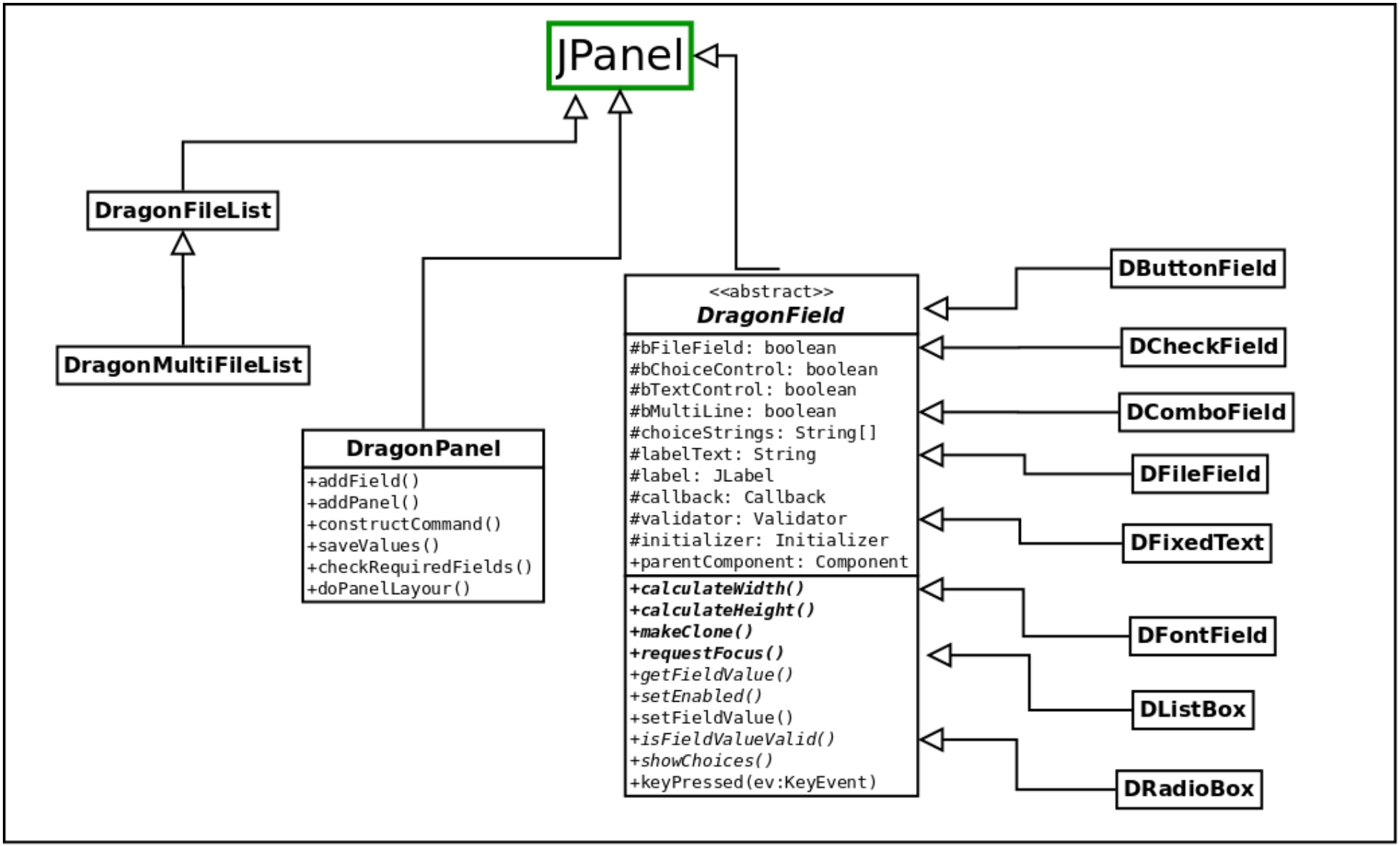


Package diagrams are a UML diagram type used to show the way classes are grouped and nested

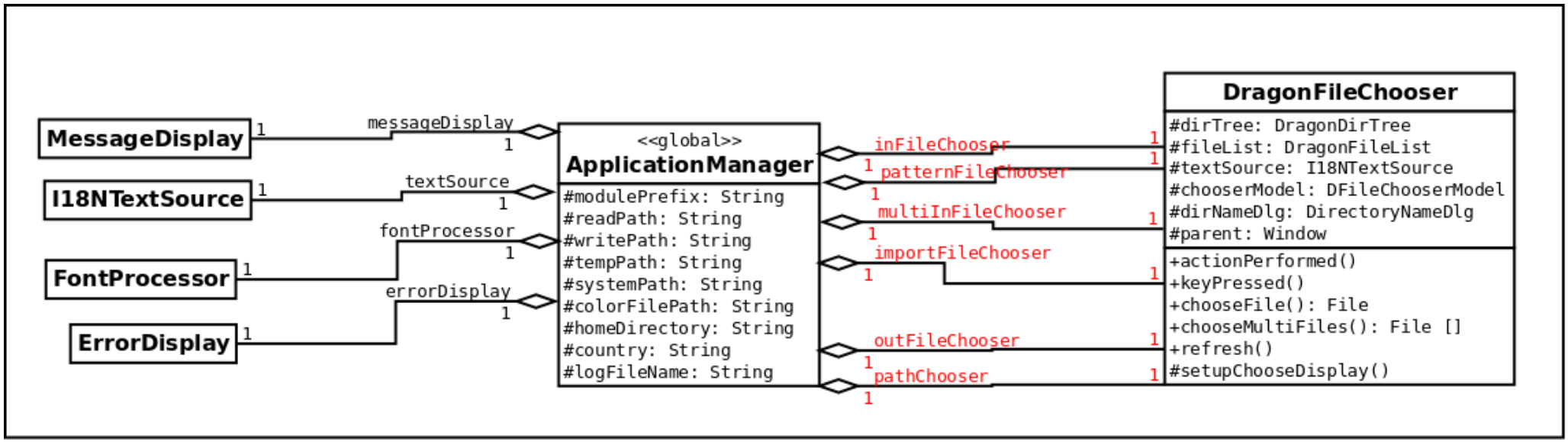
Packages with Selected Classes



Partial Class Diagram: UI Controls



ApplicationManager has many dependencies



What are the problems?

*The **XmlToUiBuilder** class is much too big – very difficult to understand and modify*

*The **XmlToUiBuilder** class incorporates too much Dragon-specific knowledge*

*The **ApplicationManager** class is a major "kludge"*

- Global "catchall" class breaks encapsulation
- Should not need to initialize the ***ApplicationManager*** in order to generate (any) UI

We need to refactor!

Hands-on Exercise

Get together with your project team mate

Download the zip file **DragonExercise.zip** and unpack on your computer

Build and run the test version of ***XmlToUiBuilder*** using ***SimpleUi.xml.*** *(See the instructions document.)*

Choose either ***XmlToUiBuilder.java*** or ***ApplicationManager.java***. Do you have any ideas how you could refactor the class to make it shorter, less complicated and/or have fewer dependencies?

- Create a class diagram illustrating your ideas for refactoring
- You will work for about an hour – then share your ideas with the rest of the class