COSC345

Week 11
Introduction to Refactoring (A)

Example

- I have a Smalltalk library.
- There is an Image class with several subclasses.
- anImage draw: aGeometric object
  - calls aGeometricObject drawOn: anImage
  - which calls back to anImage
- I recently added a Plotter class to interface to the GNU plotutils library

Example continued

- aPlotter draw: aGeometricObject
  - calls aGeometricObject drawOn: aPlotter
  - which calls back to aPlotter
- I have have one class or the other in a program but not both, because the bodies of #drawOn: are difference.
- Solution 1:
  - Rename draw: and drawOn: to plot: and plotOn:
  - Works but unsatisfactory.

Example (3)

- Solution 2: keep draw: and drawOn: because there *is* common meaning there
- Refactor drawOn: to just unpack the necessary private data and forward it to *new* methods like drawLineSegmentFrom: x1 y: y1 to: x2 y: y2
- Implement the new methods in Image and in Plotter.
- But what order should we do this?

One step at a time

- Implement drawLineSegmentFrom:y:to:y: in one class.
- Test it.
- Implement it in the other class.
- Test it.
- Refactor LineSegment>>drawOn: to use the new interface.
- Test it.
- Repeat with other methods.

Don't use seven-league boots

- From fairy-tales: a league is an hour's walk, seven-league boots let you travel a day's walk in each step.
- Programming analogue: a big change that takes you a long distance.
- If I rewrite 200 lines of code, I have lots of opportunities to make mistakes; how do I figure out what I broke?
- Small steps mean I can find a mistake quickly.

Change a constant

- Add a new named constant
  - only a problem if it clashes with another name
- Remove a named constant
  - what do you do about existing uses?
- Change the value of a named constant
  - Shouldn't be a problem; the reason for named constants is to allow this.
- Change the type of a constant
  - Have to address all uses

Change a variable

- Add, remove, change initial value, change type
  - similar to constants
- Protect (add 'const' to exported interface)
  - have to do something about all changes
- Change variable to function
  - Hard in language using f(), easy if using f
  - Can you do f() := e?
  - Setter functions can log and check changes.

Types

- Ada has **subtype** T **is** Old_Type (alias) and
- **type** T **is new** Old_Type (new incompatible type with same representation)
- Go uses **type** T = Old_Type (alias) and
- **type** T Old_Type (new incompatible type).
- Add and remove like constants and variables.
- Changing really needs aliases (C **typedef**).
- Changing types in Java is really hard.

Functions/procedures

- Add, remove, and rename similar.
- Changing the body of a method does not affect any interfaces, but may *silently* break code. Must test after such changes.
- Add an argument.
  - Easier if language allows optional parameters.
- Remove an argument.
- Change the type of an argument.

Take several steps.

- Make a new function with the same interface as the old one.
- Move the old body to the new function.
- Make the new function call the old one.
- Test.
- Change the interface of the new function and change the old function's body to match.
- Test.

Take several steps (2)

- Change the old calls a few at a time to call the new function with the changed interface.
- Test as you go.
- Keep the old function and mark it deprecated.
- Delete deprecated functions when no uses remain.