Software Engineering CS3003

Lecture 4: Software structure, code smells and refactoring

Lecture schedule

Week	Lecture Topic	Lecturer	Week Commencing
1	Introducing the module and Software Engineering	Steve Counsell	20th Sept.
2	Software maintenance and Evolution	Steve Counsell	27th Sept.
3	Software metrics	Steve Counsell	4th Oct.
4	Software structure, refactoring and code smells	Steve Counsell	11th Oct.
5	Test-driven development	Giuseppe Destefanis	18th Oct.
6	Software complexity **Coursework released Tues 26th Oct.**	Steve Counsell	25th Oct.
7	**ASK week**	**N/A**	**1st Nov**
8	Software fault-proneness	Steve Counsell	8th Nov.
9	Clean code	Steve Counsell	15th Nov.
10	Human factors in software engineering	Giuseppe Destefanis	22nd Nov.
11	SE techniques applied in action	Steve Counsell	29th Dec.
12	Guest Lecture (tba) **Coursework hand-in 6th December**	Guest Lecture	6th Dec.

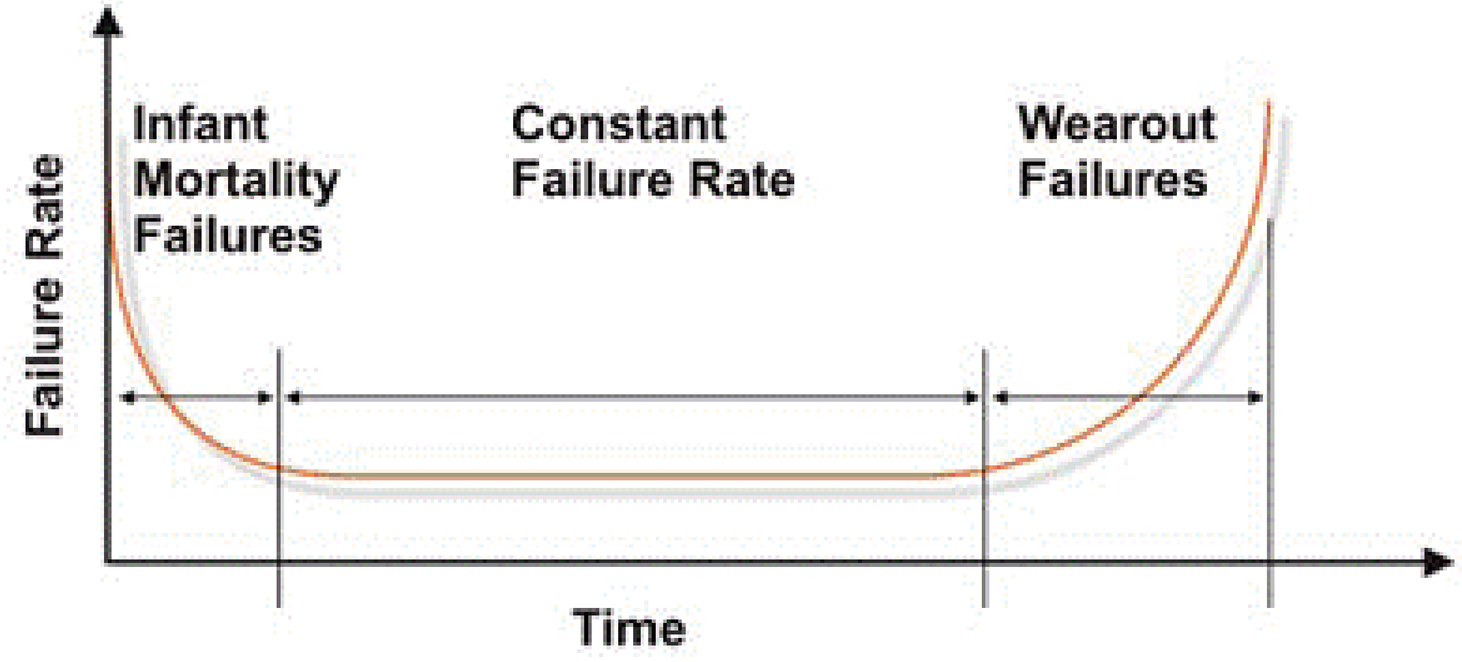
Lab schedule

Week	Labs	Week Commencing
1	**No labs**	20th Sept.
2	Lab (Introduction)	27th Sept.
3	Lab	4th Oct.
4	Lab	11th Oct.
5	Lab	18th Oct.
6	**No lab**	25th Oct.
7	**ASK week**	**1st Nov.**
8	Lab	8th Nov.
9	Catch-up Lab	15th Nov.
10	Work on coursework (no Lab)	22nd Nov.
11	Work on coursework (no Lab)	29th Nov.
12	**No lab**	6th Dec.

Why does software degrade?

- Bugs lead to improper maintenance and rush jobs
- As code ages, coupling tends to grow, leading to increased complexity
- Technical debt rises as a system ages
  - Putting off work on the code now and paying the price later
- Link with previous lectures
  - Relationship to the Bathtub curve…..

Bathtub curve re-visited



Often criticized (unjustly) the bathtub curve is more of a conceptual tool than a predictive tool.

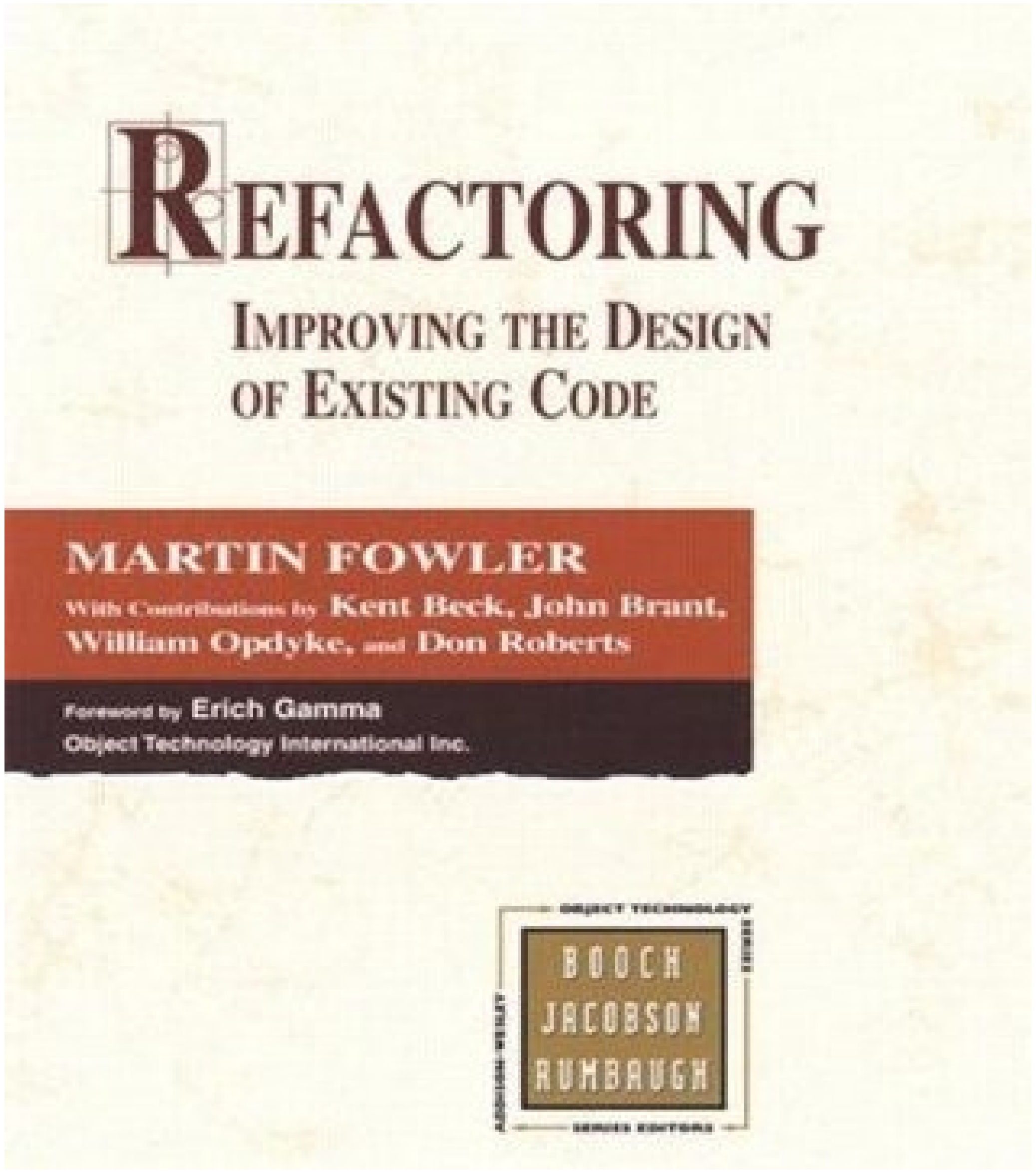
Why do developers write bad code?

- Requirements change over time, making it hard to update your code leading to less optimal designs
- Time and money cause you to take shortcuts
- You learn a better way to do something:
  - The second time you paint a room, it's always better than the first because you learned during the first time

Structure of this lecture

This lecture will answer the questions:

- What is refactoring?
- What are some of the classic refactorings
- What are code bad smells
- Explain the Law of Demeter



REFACTORING

IMPROVING THE DESIGN OF EXISTING CODE

MARTIN FOWLER

With Contributions by Kent Beck, John Brant, William Opdyke, and Don Roberts

Foreword by Erich Gamma
Object Technology International Inc.

OBJECT TECHNOLOGY SERIES

BOOCH
JACOBSON
RUMBAUGH

SERIES EDITORS

ADDISON-WESLEY

Quote from Fowler (refactoring.com)

“When a software system is successful, there is always a need to keep enhancing it, to fix problems and add new features. After all, it's called software for a reason! But the nature of a code-base makes a big difference on how easy it is to make these changes. Often enhancements are applied on top of each other in a manner that makes it increasingly harder to make changes. Over time new work slows to a crawl. To combat this change, it's important to refactor code so that added enhancements don't lead to unnecessary complexity.”

What is refactoring?

‘The process of changing a software system in such a way that it does not alter the external behaviour of the code, yet improves its internal structure’ (Fowler et al 1999)

- Does not change behaviour and user should not notice any difference
- Constant regression testing important
- A form of preventative and perfective maintenance
- Important in an XP environment and TDD

Refactoring

- Starts with an existing code base and improve the internal structure
  - Reduce duplicate code
  - Improve cohesion, reduce coupling
  - Improve understandability, maintainability, etc.
- Fowler et al 1999 suggest:
  - 22 Code smells – rotten structures in code
  - 72 Refactorings – how to eliminate code smells

How do you do refactoring?

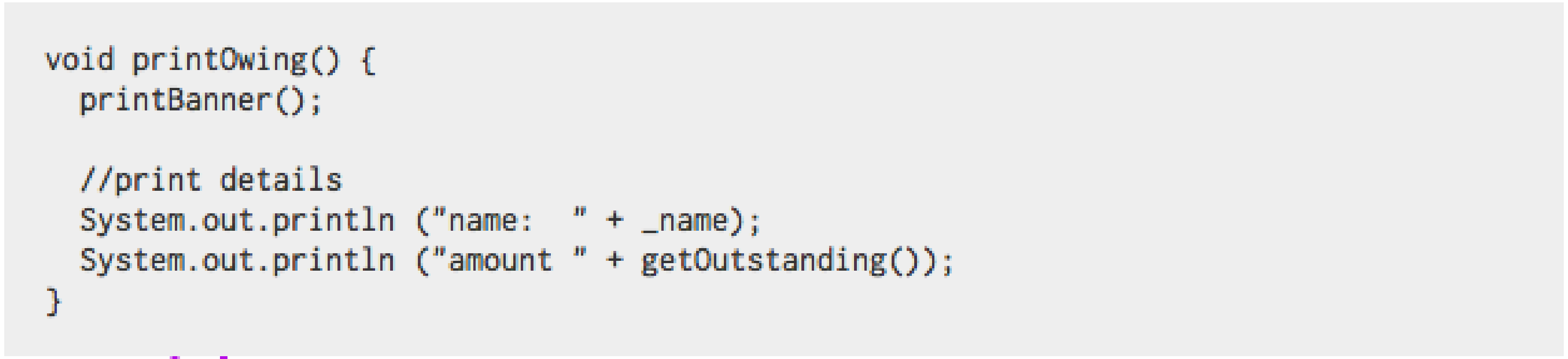
- Fowler says refactoring is not something you should dedicate two weeks every six months to…rather, you should refactor consistently and mercilessly!!
- Refactor when you:
  - Recognize a warning sign (a "bad smell")
  - Have to fix a bug
    - After fixing a bug, you could refactor the code affected
  - Do a code review (as part of pair programming)
    - Pair programming – good or bad?
- Many tools support refactoring, e.g. Eclipse, NetBeans

Extract Method

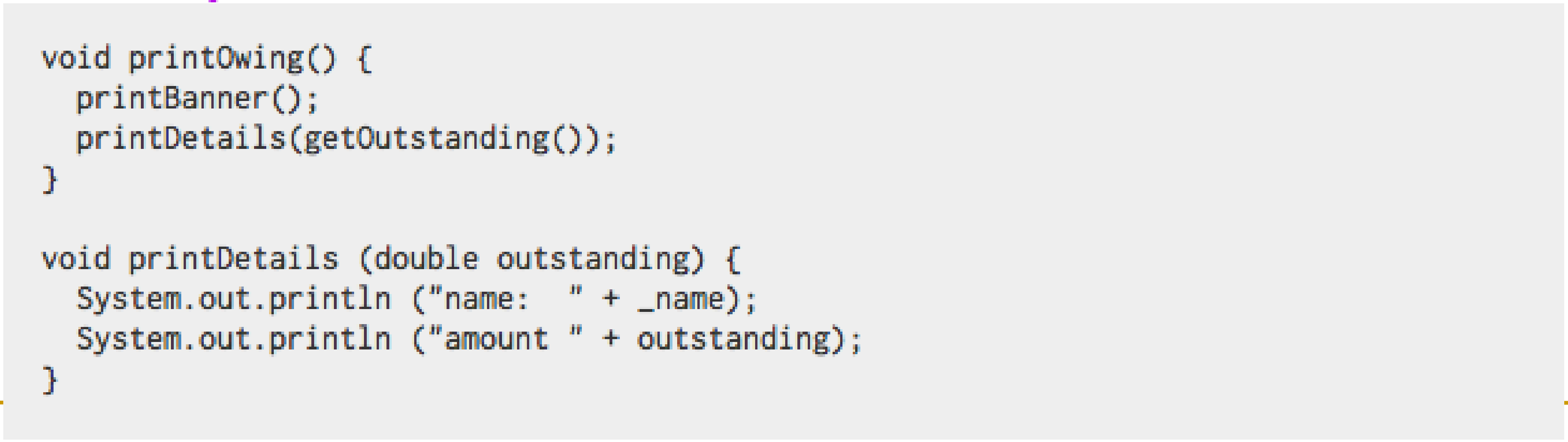
Applies when you have a code fragment inside some code block where the lines of code should always be grouped together

- *Turn the fragment into a method whose name explains the purpose of the block of code*

Extract Method Refactoring Example



```
void printOwing() {
  printBanner();

  //print details
  System.out.println ("name:  " + _name);
  System.out.println ("amount " + getOutstanding());
}
```



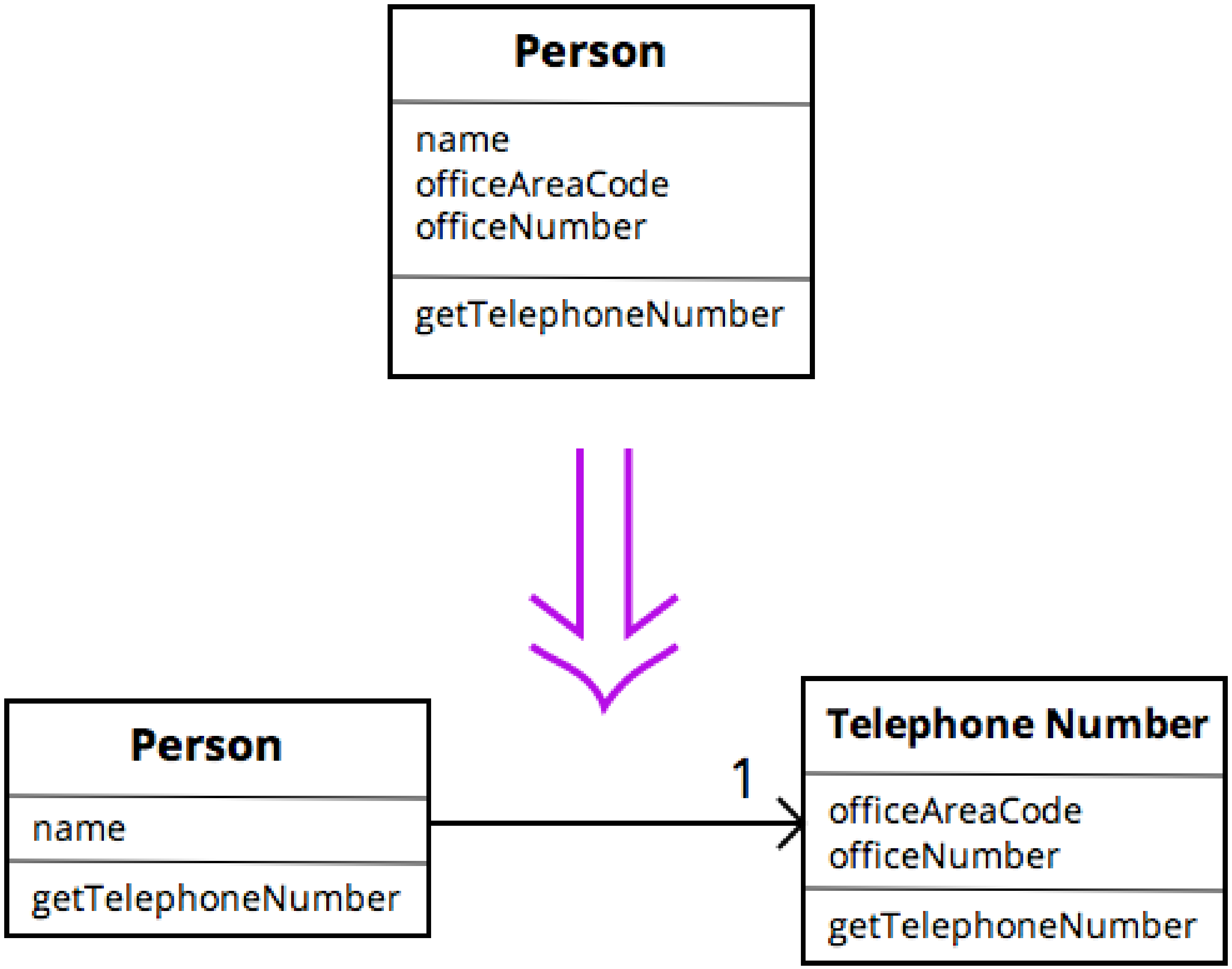
```
void printOwing() {
  printBanner();
  printDetails(getOutstanding());
}

void printDetails (double outstanding) {
  System.out.println ("name:  " + _name);
  System.out.println ("amount " + outstanding);
}
```

Extract Class

You have one class doing work that should be done by two different classes

- *Create a new class and move the relevant fields and methods from the old class to the new class*

Extract Class Example



Extract Subclass

When a class has features (attributes and methods) that would only be useful in specialized instances, we can create a specialization of the class and give it those features

```
public class Person
{
 private String name;
 private String jobTitle;
}
```

→

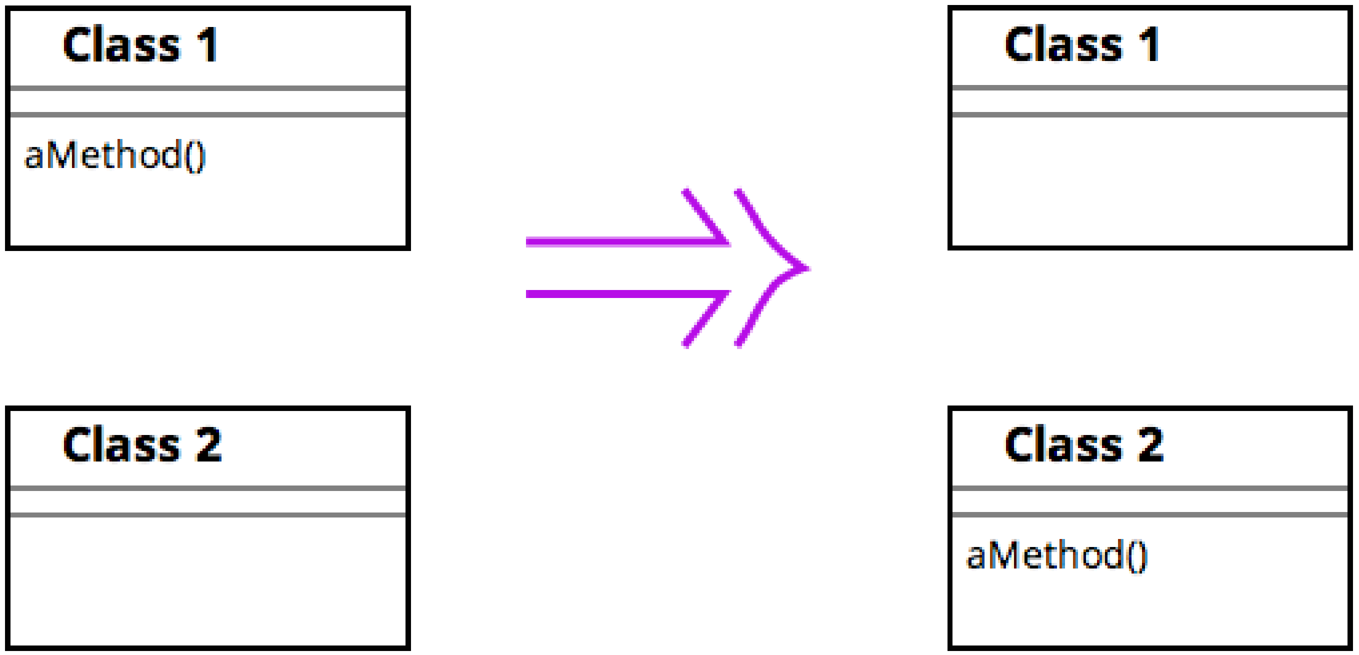
```
public class Person
{
 protected String name;
}

public class Employee extends Person
{
 private String jobTitle;
}
```

Move Method

A method is used more by another class, or uses more code in another class, than its own class.

- *Well, then, move it. Create a new method with a similar body in the class it uses most.*

Move Method Example



Encapsulate Field

```
public String _name
```

$$\Downarrow$$

```
private String _name;
public String getName() {return _name;}
public void setName(String arg) {_name = arg;}
```

Replace Magic Number with Symbolic Constant

Before:

```
// Calculate the length of the outer crust of each
// piece of pizza given the diameter of the pizza
// and number of desired pieces.
public float lengthOfOuterCrust(float diameter, int numberOfPieces) {
    return (3.1415 * diameter) / numberOfPieces;
}
```

After:

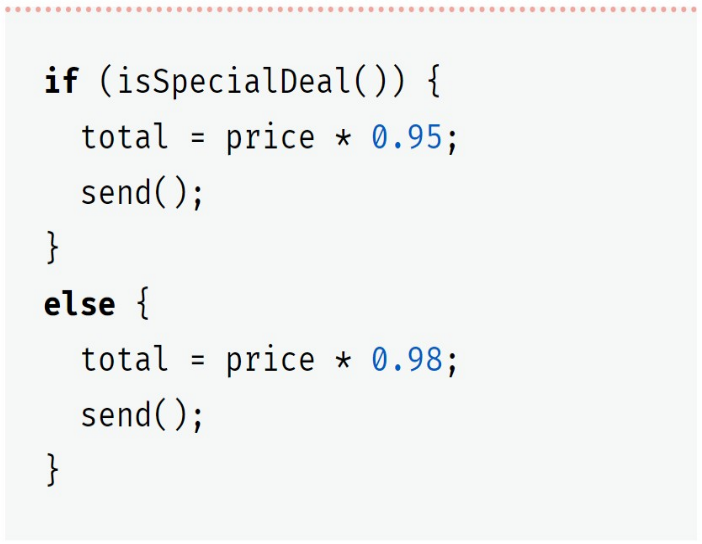
```
public static final float PI = 3.1415;
public float lengthOfOuterCrust(float diameter, int numberOfPieces) {
    float circumference = PI * diameter;
    return circumference / numberOfPieces;
}
```

Example taken form sce2.umkc.edu

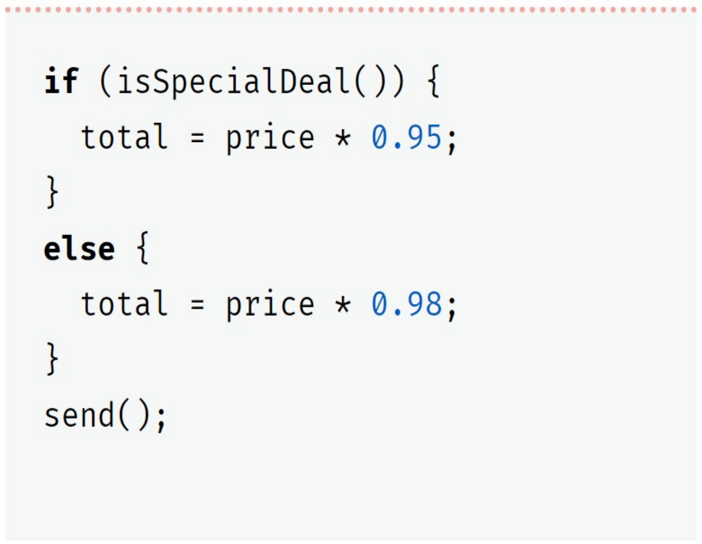
Remove Dead Code

```
public class DeadCodeInJava
{
   void DeadCode_method(boolean b)
   {
      if(b)
      {
         return;
      }
      else
      {
         return;
      }
       System.out.println(“Dead Statement");
   }
}
```

Consolidate conditional duplicate fragments



```java
if (isSpecialDeal()) {
  total = price * 0.95;
  send();
}
else {
  total = price * 0.98;
  send();
}
```



```java
if (isSpecialDeal()) {
  total = price * 0.95;
}
else {
  total = price * 0.98;
}
send();
```

We’ve moved the ‘send()’ to outside the if statement

Code smells

What are Code Smells?

- http://martinfowler.com/bliki/CodeSmell.html
- http://www.codinghorror.com/blog/2006/05/code-smells.html

Code Smells

- Large Class
  - classes try to do too much, which reduces cohesion.
  - You break it up with the extract class refactoring
- Long Parameter List
  - Hard to understand, can become inconsistent
  - You resolve it with the remove parameter refactoring

Code Smells (cont.)

- Lazy Class
  - A class that no longer "pays its way"
    - e.g. may be a class that was downsized by a previous refactoring, or represented planned functionality that did not pan out
- Solution sprawl
  - You need to add or remove a feature from a project, but in order to do so must change many different components in many different classes

Code Smells (cont.)

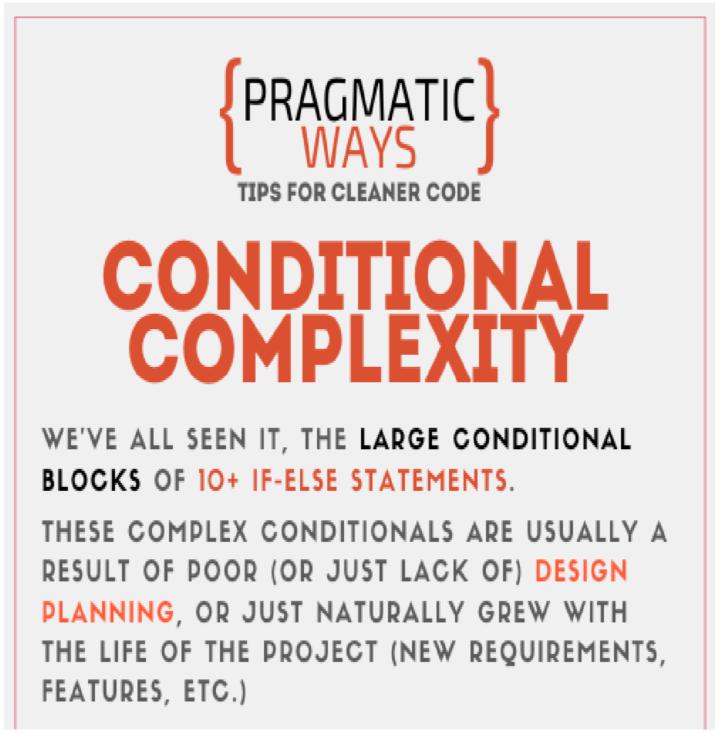
- Data Class
  - These are classes that have fields, getting and setting methods for the fields, and nothing else; they are data holders, but objects should be about data AND behaviour
- Refused Bequest
  - A subclass ignores most of the functionality provided by its superclass
- Comments
  - Comments are sometimes used to hide bad code
  - What is a good comment………..

Comments Code Smell

- Fowler claims that comments are a sign of:
  - Complicated code
  - Code that is in need of explanation but that actually needs restructuring rather than commenting
- Fowler advocates:
  - Self-evident coding practices by making methods short, comments meaningful
  - Commenting the decision making process for the solution implemented
    - i.e., why something is there, not what is does

Conditional complexity

```
int temperature = 0;

if (temperature <= 0)
{
   System.out.println("Water will freeze; the freezing point of water is
0°C.");
}
else if (temperature >= 208 )
{
   System.out.println("A good temperature for brewing black tea.");
}
else if (temperature >= 180 )
{
   System.out.println("A good temperature for brewing oolong tea.");
}
else if (temperature >= 170 )
{
   System.out.println("A good temperature for brewing white and green
teas.");
}
else
{
   System.out.println("Water is not cold enough to freeze,");
   System.out.println("nor a good temperature for brewing tea.");
```



{PRAGMATIC WAYS}
TIPS FOR CLEANER CODE

CONDITIONAL COMPLEXITY

WE'VE ALL SEEN IT, THE **LARGE CONDITIONAL BLOCKS** OF 10+ IF-ELSE STATEMENTS.

THESE COMPLEX CONDITIONALS ARE USUALLY A RESULT OF POOR (OR JUST LACK OF) DESIGN PLANNING, OR JUST NATURALLY GREW WITH THE LIFE OF THE PROJECT (NEW REQUIREMENTS, FEATURES, ETC.)

When should you refactor?

- You should refactor:
  - Any time that you see a better way to do things
    - “Better” means making the code easier to understand and to modify in the future
  - Mercilessly and relentlessly
- You should not refactor:
  - Stable code that doesn’t need to change
    - - Example: code that has been archived
  - Someone else’s code
    - Unless the other person agrees to it or it belongs to you
- Refactoring is NOT the same as re-engineering

When should you not refactor?

Common reasons not to do it:

a) *'Don't have time; features more important'* – refactoring is a luxury that we can't afford.

b) *'We might break something'* – If it ain't broke, don't fix it

c) *'I want to get promoted and companies don't recognize refactoring work'* -- This is a common problem. Solution varies depending on company. Seek buy-in from your team, gather data about regressions and flaws in the design, and encourage them to buy-in to code quality."

-- Victoria Kirst,
Software Engineer, Google

Refactoring in a team context

Amount of overhead/communication needed depends on size of refactor:

- A *small refactor:* Just do it, check it in, get it code reviewed
- *medium refactor:* Possibly involve the technical lead or another developer
- *large refactor:* Meet with team, flush out ideas, do a design doc or design review, get approval before beginning

Talking avoids 2 possible bad scenarios:

- Two developers refactor same code simultaneously
- Refactoring breaks another developer's new feature they are adding

Refactoring at Google (Marty Stepp)

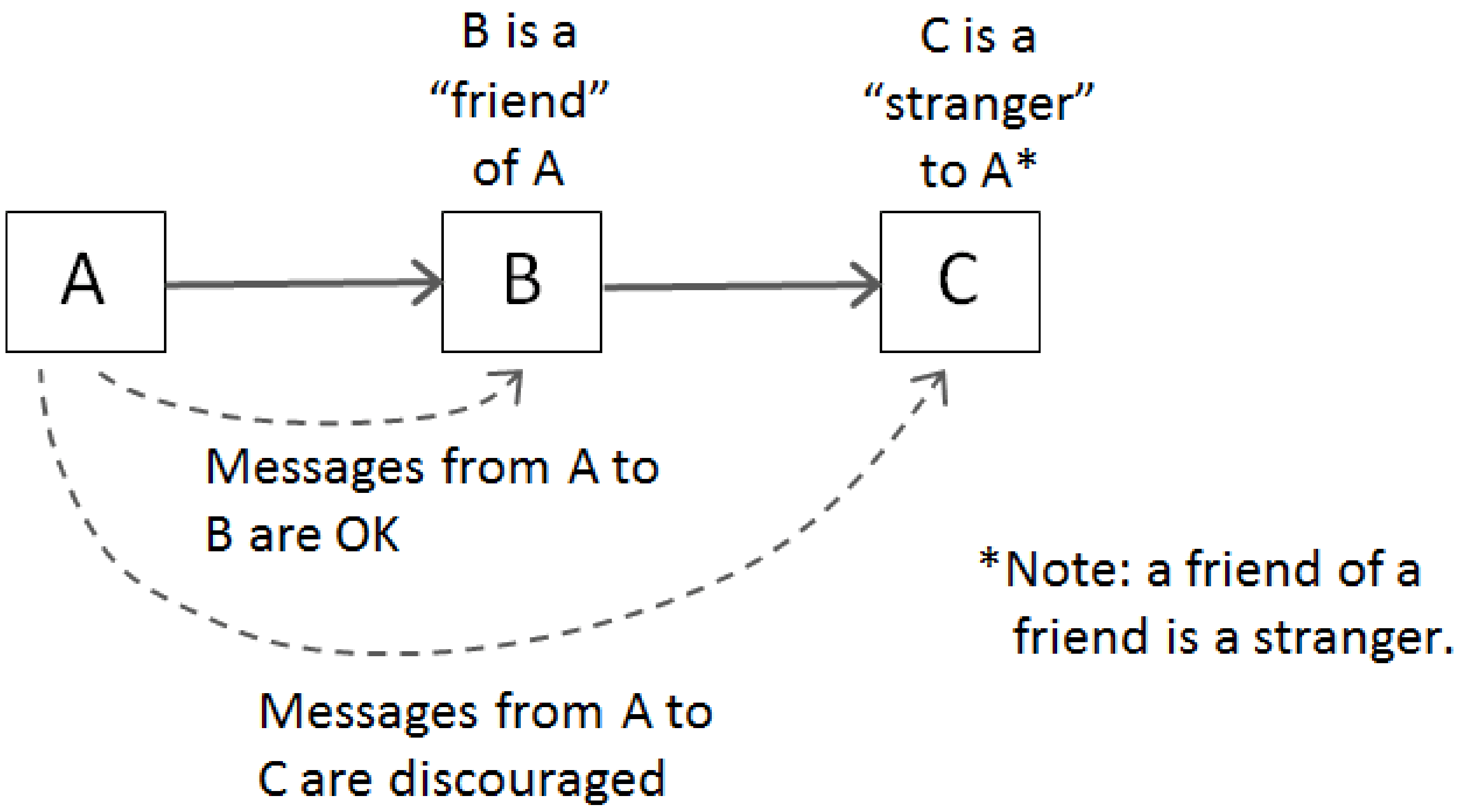
"At Google, refactoring is very important and necessary/inevitable for any code base. If you're writing a new app quickly and adding lots of features, your initial design will not be perfect. Ideally, do *small* refactoring tasks early and often, as soon as there is a sign of a problem."

"Refactoring is unglamorous because it does not add features. At many companies, people don't refactor because you don't get promoted for it, and their code turns into hacky beasts."

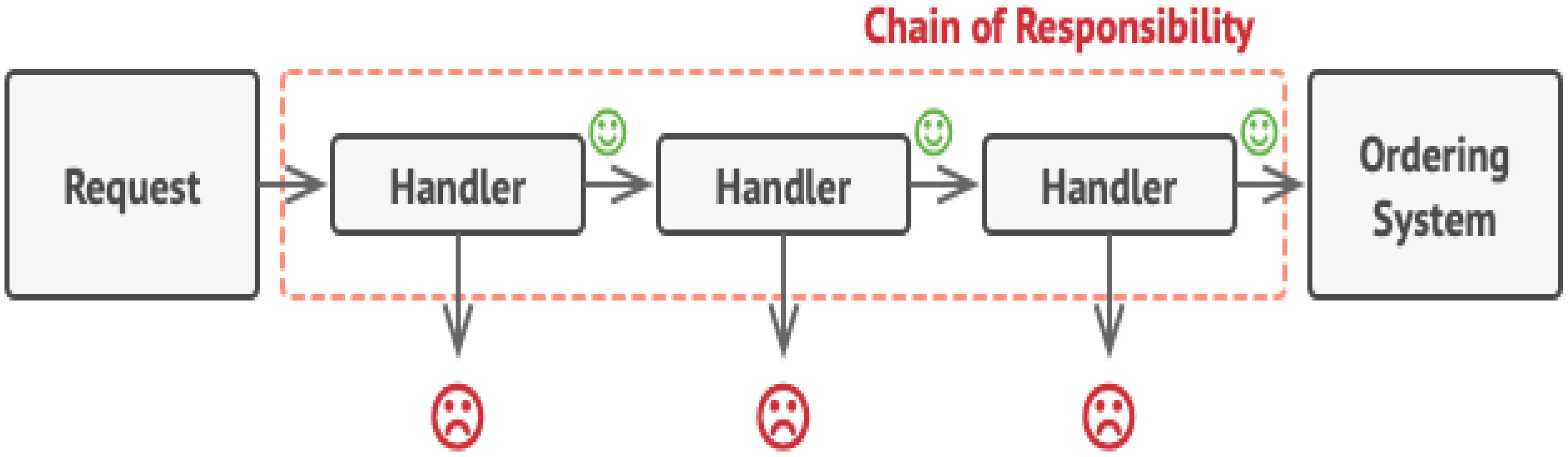
"Google feels refactoring is so important that there are company-wide initiatives to make sure it is encouraged and rewarded."

Law of Demeter (LoD)

- The Law of Demeter or principle of least knowledge
- A design guideline for developing software, particularly where coupling is a big factor
- Proposed by Ian Holland at Northeastern University (USA) in 1987:
  - Each class should have only limited knowledge about other classes: only classes "closely" related to the current class
  - Each class should only talk to its friends; don't talk to strangers

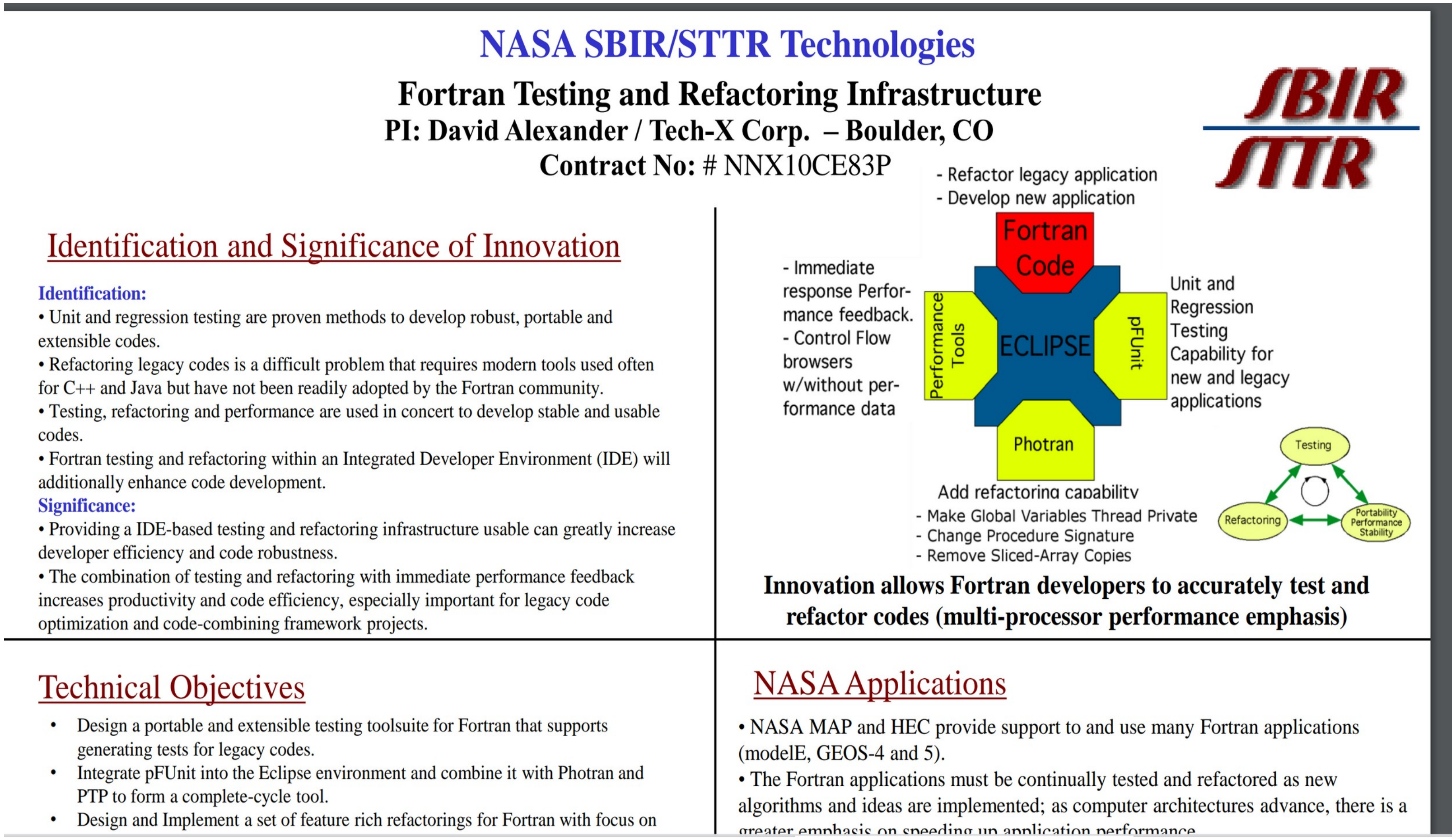

B is a “friend” of A
C is a “stranger” to A*
A
B
C
Messages from A to B are OK
Messages from A to C are discouraged
*Note: a friend of a friend is a stranger.

Why you should not violate LoD



Refactoring.guru

NASA uses refactoring



NASA SBIR/STTR Technologies

Fortran Testing and Refactoring Infrastructure
PI: David Alexander / Tech-X Corp. – Boulder, CO
Contract No: # NNX10CE83P

SBIR STTR

Identification and Significance of Innovation

Identification:
- Unit and regression testing are proven methods to develop robust, portable and extensible codes.
- Refactoring legacy codes is a difficult problem that requires modern tools used often for C++ and Java but have not been readily adopted by the Fortran community.
- Testing, refactoring and performance are used in concert to develop stable and usable codes.
- Fortran testing and refactoring within an Integrated Developer Environment (IDE) will additionally enhance code development.

Significance:
- Providing a IDE-based testing and refactoring infrastructure usable can greatly increase developer efficiency and code robustness.
- The combination of testing and refactoring with immediate performance feedback increases productivity and code efficiency, especially important for legacy code optimization and code-combining framework projects.

- Refactor legacy application
- Develop new application
Fortran Code
- Immediate response Performance feedback.
- Control Flow browsers w/without performance data
Performance Tools
ECLIPSE
pFUnit
Unit and Regression Testing Capability for new and legacy applications
Photran
Add refactoring capability
- Make Global Variables Thread Private
- Change Procedure Signature
- Remove Sliced-Array Copies
Testing
Refactoring
Portability Performance Stability

Innovation allows Fortran developers to accurately test and refactor codes (multi-processor performance emphasis)

Technical Objectives

- Design a portable and extensible testing toolsuite for Fortran that supports generating tests for legacy codes.
- Integrate pFUnit into the Eclipse environment and combine it with Photran and PTP to form a complete-cycle tool.
- Design and Implement a set of feature rich refactorings for Fortran with focus on

NASA Applications

- NASA MAP and HEC provide support to and use many Fortran applications (modelE, GEOS-4 and 5).
- The Fortran applications must be continually tested and refactored as new algorithms and ideas are implemented; as computer architectures advance, there is a greater emphasis on speeding up application performance

Questions

- **Question 1:** Describe any **FOUR** code smells that you know of. Include in your answer an explanation of why you think each smell is problematic for code.
- **Question 2:** What are the benefits of refactoring code?
  - Can you think of any disadvantages of refactoring?
- **Question 3:** What is the Law of Demeter? Use a diagram to explain your understanding.
- **Question 4:** What would your advice be on the writing of comments in code to ensure that you make the code as maintainable as possible?

Question 2 – Refactoring advantages

- Reduces duplicate and inefficient code
- Improve cohesion, reduce coupling
- Improves the understandability of code, maintainability, etc. by reducing complexity
- Can help reduce technical debt
- Can help reduce bugs since the code is more maintainable
- Can be done at any stage of a system’s life

Question 2 – Refactoring disadvantages

- Opportunity cost - there are many other things that can be done instead of refactoring and this might be time better spent
- Complex in many cases and always needs significant re-testing
- No guarantee that the refactoring won't break despite testing or that it makes code any less bug-prone
- Time – it takes time to do refactoring and communication overhead if you're working in a team
- There is not a huge amount of evidence that refactoring actually provides the benefits it claims
- Can lead to developers going down "rabbit holes"

Reading for the week

- Refactoring: Improving the Design of Existing Code, Martin Fowler(et al.), 1999, Addison-Wesley
- What refactorings do you apply to a smell?
  - http://wiki.java.net/bin/view/People/SmellsToRefactorings
  - https://sourcemaking.com/refactoring
- https://www.codeproject.com/Articles/13212/Code-Metrics-Code-Smells-and-Refactoring-in-Practi

Reading for the week (cont.)

- www2.ccs.neu.edu/research/demeter/demeter-method/LawOfDemeter/general-formulation.html
- www.altexsoft.com/blog/engineering/code-refactoring-best-practices-when-and-when-not-to-do-it/
- refactoring.com/
- https://thenewstack.io/refactoring-is-not-bad-until-it-is/

Reading for the week (cont.)

- https://www.codeproject.com/Articles/771894/When-Refactoring-Is-A-Bad-Idea