COMP 5700/6700/6706 Software Process

Spring 2016

David Umphress

Refactoring

- Lesson: Refactoring
- Strategic Outcome:
  - To understand refactoring
- Tactical Outcomes:
  - To know the rationale and purpose of refactoring
  - To understand programming etiquette
  - To know fundamental "bad smells"
  - To understand fundamental refactorings
  - To be able to apply refactoring to a sample problem
- Readings
  - "Refactoring" http://en.wikipedia.org/wiki/Refactoring
- Instant take-aways:
  - Refactoring
  - Programming etiquette
- Bookshelf items
  - Fowler, M. 2000. *Refactoring: Improving the Design of Existing Code.* Addison Wesley
  - www.refactoring.com
  - McConnell, S. 2004. *Code Complete*. 2nd Ed. Microsoft Press.

Syllabus

- Software engineering raison d'être
- Process foundations
- Common process elements
- Construction
- Reviews
- Refactoring
- Analysis
- Architecture
- Estimation
- Scheduling
- Integration
- Repatterning
- Measurements
- Process redux
- Process descriptions*
- Infrastructure*
- Retrospective

- **Programming Etiquette**
- **Refactoring**
  - **rationale**
  - **definition**
  - **bad smells**
  - **fundamental refactorings**

COMP5700/6700/6706 Goal Process

Minimal Guiding Indicators

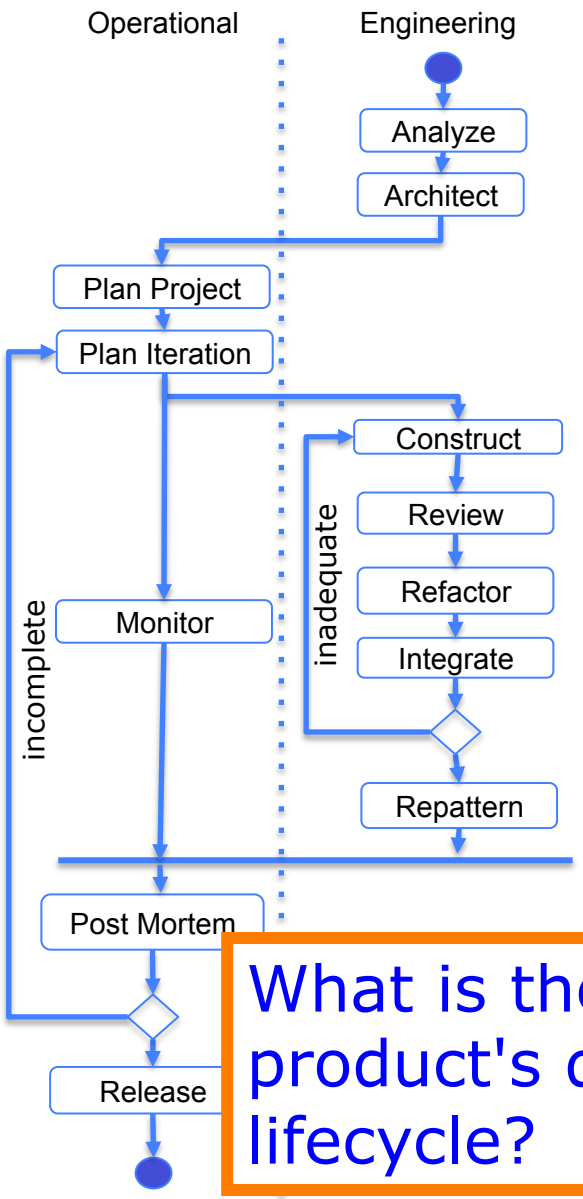
Goal	Indicator
Cost:	None
Schedule:	PV/EV >.75
Performance:	
Product:	
NFR:	none
FR:	100% BVA
Process:	pain < value

Minimal Sufficient Activities

Engineering Activities
- Envision
  - Analyze
- Synthesize
  - Architect
- Articulate
  - Construct
  - Refactor
- Interpret
  - Review
  - Integrate
  - Repattern

Operational Activities
- Plan
  - Plan project
  - Plan iteration
- Monitor
- Release

Minimal Viable Process



Minimal Effective Practice

MSA	MEP
Analyze	Scenarios
Architect	CRC
Plan Project	Component-based estimation
Plan Iteration	Component-iteration map
Construct	TDD
Review	Review checklist Test code coverage
Refactor	Ad hoc sniffing
Integrate	Ad hoc
Repattern	Ad hoc
Monitor	Time log Change log Burndown
Post Mortem	PV/EV
Release	Eclipse zip spreadsheets

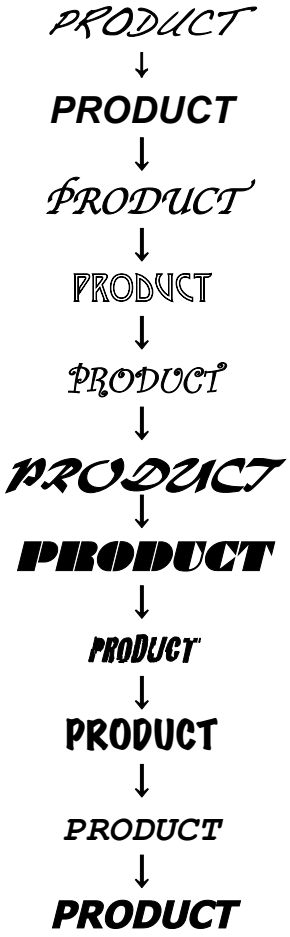
What is the biggest risk to a product's design in an iterative lifecycle?

Evolutionary design

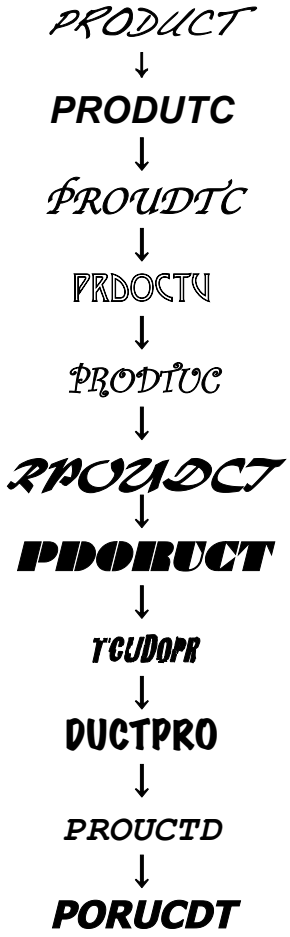
- PCSE prescribes an iterative lifecycle
  - advantages:
    - software is built incrementally as we gain an increasingly accurate insight into functionality
    - we have a working system at the end of each iteration
  - disadvantages:
    - have to deal with an evolving design … meaning, our design has the possibility of changing with each iteration

Evolutionary design (con't)

ideally, we would like this to happen:



what we often get is this:



Programming Etiquette

- "A [person's] manners are a mirror in which he shows his portrait." Johann Wolfgang von Goethe
- Don't …
  - …obfuscate code to show technical prowess
  - …use documentation to clarify bad code
  - …use variable names that are difficult to distinguish
    - X vs x
    - ClientRecs vs ClientReps
  - …make assumptions

Programming Etiquette (con't)

- Do …
  - …use coding conventions
  - …use names that are semantically transparent
    - generic nouns and noun phrase for classes
    - nouns and noun phrases for instances
    - verbs and verb phrases for methods
    - accepted conventions for accessors and mutators
  - …understand the domain
  - …follow sound encapsulation/information hiding principles
  - …write code in the simplest fashion
  - …use comments constructively

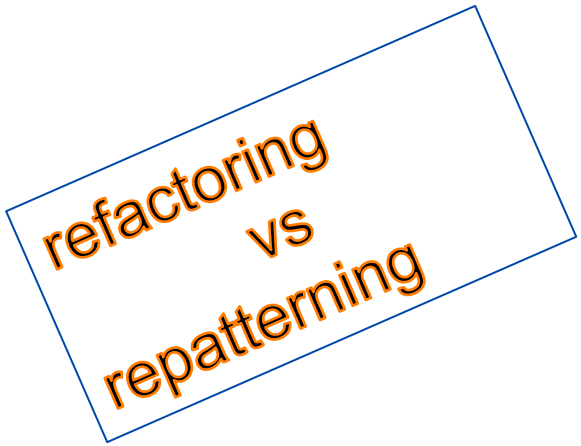
Programming Etiquette (con't)

- DO (cont')
  - …Make "gotchas" explicit*
    - TODO
      - Means there's more to do here, don't forget.
    - BUG: [bugid] topic
      - means there's a known bug here, explain it and optionally give a bug ID.
    - KLUDGE
      - When you've done something ugly say so and explain how you would do it differently next time if you had more time.
    - TRICKY
      - Tells somebody that the following code is very tricky and may have side effects.
    - WARNING
      - Beware of something.
    - COMPILER
      - Sometimes you need to work around a compiler problem. Document it. The problem may go away eventually.
    - CITATION: http://www.agorithmsOrUs.com/AVL.html
      - Gives a reference to a published source
    - PERFORMANCE
      - Gives rationale for using a particular technique for performance
  - …Refactor

* from http://www.possibility.com/Cpp/CppCodingStandard.html#mge

Refactoring

- Refactoring is the process of changing a software system such that
  - the external (i.e., functional) behavior of the system does not change
  - the internal structure of the system is improved
- Concept:
  - it is difficult to get a correct design the first time
    - due to requirements changes
    - due to learning more about user needs, programmer capabilities, technology improvements, etc.
  - improving the design after implementation is not the same as performance optimization
- Purpose:
  - to make software easier to understand
  - to help find bugs
  - to prepare software for next iteration
  - to speed development process



Refactoring

- Side effects:
  - code size is often reduced
  - confusing structures are transformed into simpler constructs
- Lessons learned:
  - management buy-in is necessary
  - must follow systematic approach to refactoring
  - refactor at the appropriate point in the lifecycle
    - TDD purists: continuously
    - PCSE: at end of iteration

Refactoring (con't)

- Simple example: Consolidating duplicate conditional fragments
  - Original code

```
if (isSpecialDeal())
        total = price * .95
        send()
else
        total = price * .98
        send()
end
```

  - Refactored code

```
if (isSpecialDeal())
        total = price * .95
else
        total = price * .98
end
send()
```

What to Refactor?

How do we know what to refactor?

Bad smells

- Duplicated code
- Long method
- Large class
- Long parm list
- Shotgun surgery
- Feature envy
- Data clumps
- Primitive obsession
- Switch statements
- Parallel inheritance hierarchies
- Lazy class
- Speculative generality
- Temporary field
- Message chains
- Middle man
- Inappropriate intimacy
- Alternate class with different interfaces
- Incomplete library classes
- Data class
- Refused bequest
- Comments

Bad Smells

- Duplicate code
- Long method
- Large class
- Long parm list
- Shotgun surgery
  - a change requires a lot of little changes to a lot of different classes
- Feature envy
  - a method in a class seems not to belong

Bad Smells (con't)

- Data clumps
  - member fields that clump together but are not part of the same class
- Primitive obsession
  - characterized by the use of primitives in place of class methods
- Switch statements
  - often duplicated code that can be replaced by polymorphism
- Parallel inheritance hierarchies
  - duplicated code in subclasses that share a common ancestor

Bad Smells (con't)

- Speculative generality
  - methods (often stubs) that are placeholders for future features
- Temporary field
  - a variable that is used only under certain circumstances and is reused later under other circumstances
- Message chains
  - object that requests an object for another object which asks …
- Middle man
  - a class that is just a “pass-through” method with little logic

Bad Smells (con't)

- Inappropriate intimacy
  - violation of private parts
- Alternate class with different interfaces
  - two methods that do the same thing, but have different interfaces
- Incomplete library classes
  - a framework that doesn't do everything you need
- Refused bequest
  - a subclass that over-rides most of the functionality provided by its superclass
- Comments
  - text that explains bad code (vs fixing the code)
- Standards violations
  - code that doesn't adhere to a pre-determined standards, such as a coding standard

Refactorings

- Warning: there are a large number of individual refactorings www.refactoring.com
- Categories of refactorings:
  - composing methods
    - goal: ensure loose coupling
  - moving features between objects
    - goal: ensure tight cohesion
  - organizing data
    - goal: encapsulate data appropriately
  - simplifying conditional expressions
    - goal: unclutter decision points
  - making method calls simpler
    - goal: use parameterization sensibly
  - dealing with generalization
    - goal: use inheritance structure properly
  - big refactorings
    - goal: re-architect if necessary

A Look at A Simple Refactoring

- Composing Methods
  - Extract method
  - Inline method
  - Inline temp
  - Replace temp with query
  - Introduce explaining variable
  - Split temporary variable
  - Remove assignments to parameter
  - Replace method with method object
  - Substitute algorithm

Refactoring (con't)

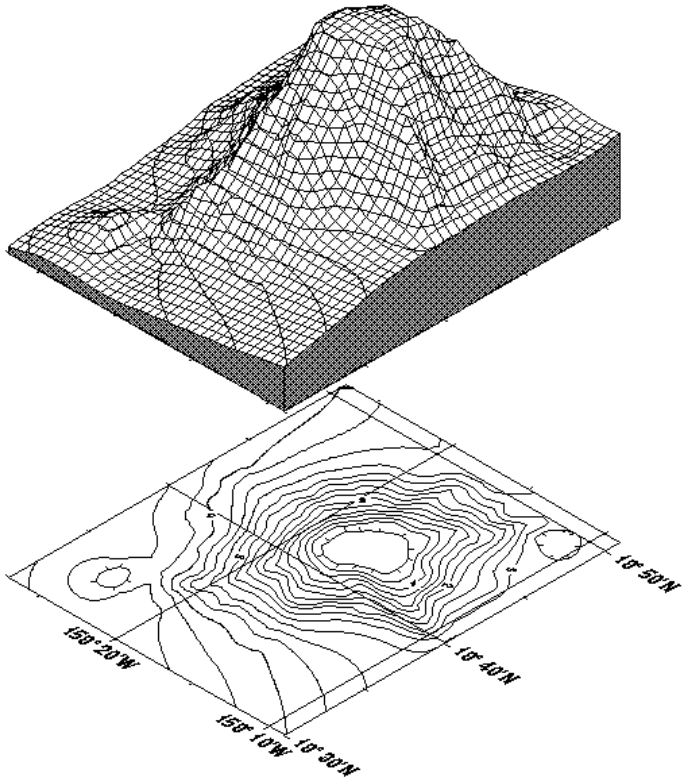
- Extract Method
  - context
    - Suppose you have a gob of code and, after a sniff test, decide that you should put some of the code in a separate method.
  - Motivation
    - cohesion
    - clarity
    - put "semantic distance" between what code does and how it does it

Refactoring (con't)

- Extract Method
  - Mechanics
    - Create new method
      - name by what it does, not how it does it
      - rule of thumb: if you can't come up with a meaningful name, then don't extract
    - Copy extracted code from source to target
    - Locate variables used in target but defined in source scope. Choices:
      - if modified by target, consider method as a query
      - make variables parms to method
      - if can used only in extracted code, make temps
    - Replace extracted code in source with call
      - remove extraneous variables
    - Test

```
def printOwing(order):
    outstanding = 0.0

    output = ["**************************"]
    output.append("***  Customer Owes  ***")
    output.append("**************************")

    for i in range(0,len(order.invoice)):
        tmp = order.invoice[i].getAmount()
        outstanding = outstanding + tmp

    output.append(order.name)
    output.append(outstanding)

    return output
```

"Islands" of abstraction



```
def printOwing(order):
    outstanding = 0.0

    output = ["**********************"]
    output.append("***  Customer Owes  ***")
    output.append("**********************")

    for i in range(0,len(order.invoice)):
        tmp = order.invoice[i].getAmount()
        outstanding = outstanding + tmp

    output.append(order.name)
    output.append(outstanding)

    return output
```

```
def printOwing(order):
    outstanding = 0.0

    output = generateBanner()

    for i in range(0,len(order.invoice)):
        tmp = order.invoice[i].getAmount()
        outstanding = outstanding + tmp

    output.append(order.name)
    output.append(outstanding)

    return output
```

```
def generateBanner():
    output = ["**********************"]
    output.append("***  Customer Owes  ***")
    output.append("**********************")
    return output
```

```
def printOwing(order):
    outstanding = 0.0

    output = generateBanner()

    for i in range(0,len(order.invoice)):
        tmp = order.invoice[i].getAmount()
        outstanding = outstanding + tmp

    output.append(order.name)
    output.append(outstanding)

    return output
```

```
def printOwing(order):
    outstanding = 0.0

    output = generateBanner()

    for i in range(0,len(order.invoice)):
        tmp = order.invoice[i].getAmount()
        outstanding = outstanding + tmp

    generateOutstanding(order.name, outstanding, output)

    return output
```

```
def generateOutstanding(name, outstanding, output):
    output.append(name)
    output.append(outstanding)
```

```
def printOwing(order):
    outstanding = 0.0

    output = generateBanner()

    for i in range(0,len(order.invoice)):
        tmp = order.invoice[i].getAmount()
        outstanding = outstanding + tmp

    generateOutstanding(order.name, outstanding, output)

    return output
```

```
def printOwing(order):
    output = generateBanner()
    outstanding = getOutstanding(order.invoice)
    generateOutstanding(order.name, outstanding, output)
    return output
```

```
def getOutstanding(invoice):
    outstanding = 0.0
    for theInvoice in invoice:
        outstanding = outstanding + theInvoice.getAmount()

    return outstanding
```

```
def getOutstanding(invoice):
    outstanding = 0.0
    for theInvoice in invoice:
        outstanding = outstanding +  theInvoice.getAmount()
    return outstanding
```

```
def generateOutstanding(name, outstanding, output):
    output.append(name)
    output.append(outstanding)
```

```
def generateBanner():
    output = ["**********"]
    output.append("***  Customer Owes  ***")
    output.append("**********")
    return output
```

```
def printOwing(order):
    output = generateBanner()
    outstanding = getOutstanding(order.invoice)
    generateOutstanding(order.name, outstanding, output)
    return output
```

```
def printOwing(order):
    outstanding = 0.0

    output = ["**********"]
    output.append("***  Customer Owes  ***")
    output.append("**********")

    for i in range(0,len(order.invoice)):
        tmp = order.invoice[i].getAmount()
        outstanding = outstanding + tmp

    output.append(order.name)
    output.append(outstanding)

    return output
```

```
def getOutstanding(invoice):
    outstanding = 0.0
    for theInvoice in invoice:
        outstanding = outstanding + theInvoice.getAmount()
    return outstanding
```

```
def generateOutstanding(name, outstanding, output):
    output.append(name)
    output.append(outstanding)
```

```
def generateBanner():
    output = ["**********"]
    output.append("***  Customer Owes  ***")
    output.append("**********")
    return output
```

```
def printOwing(order):
    output = generateBanner()
    outstanding = getOutstanding(order.invoice)
    generateOutstanding(order.name, outstanding, output)
    return output
```

Refactoring (con't)

- Our approach
  - the inventory of refactorings is large (and getting larger)
  - our goal is to eliminate the most odious smells:
    - Duplicate code
    - Long method
    - Large class
    - Inappropriate intimacy
    - Temporary field
    - Comments
    - Coding standard violations

Summary

Topics

- Programming Etiquette
- Refactoring
  - rationale
  - definition
  - bad smells
  - fundamental refactorings

Key Points

- Programming etiquette commits us to writing code that meaningful
- Refactoring is the practice of cleaning up component internals
  - Items to refactor are indicated by bad smells
  - A "refactoring" is a formal set of steps for eliminating a specific bad smell
  - PCSE advocates refactoring at the end of each iteration