Software Testing and Maintenance
Agile Software Development

Jeff Offutt

SWE 437
George Mason University
2008

Based on *Agile Estimating and Planning*, Cohn, Prentice Hall, Chapters 1-3
Thanks to Ian Sommerville

Agile Software Development

1. **Agile Process Overview**
2. **Extreme Programming**
3. **Refactoring**
4. **Refactoring Techniques**

Agile Software Development Manifesto

"We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan That is, while there is value in the items on the right, we value the items on the left more."

–Kent Beck *et al*.

What is "Agility"?

- **Effective (rapid and adaptive) response to change**
- **Effective communication among all stakeholders**
- **Drawing the customer onto the team**
- **Organizing a team so that it controls the work**

Yielding ...

Rapid, incremental delivery of software

An Agile Process

- **Is driven by customer descriptions of what is required (scenarios)**
- **Recognizes that plans are short-lived**
- **Develops software iteratively with a heavy emphasis on construction activities**
- **Delivers multiple "software increments"**
- **Adapts as changes occur**

Lots of agile processes have been defined, XP is the most widely known ...

Agile Software Development

1. **Agile Process Overview**
2. **Extreme Programming**
3. **Refactoring**
4. **Refactoring Techniques**

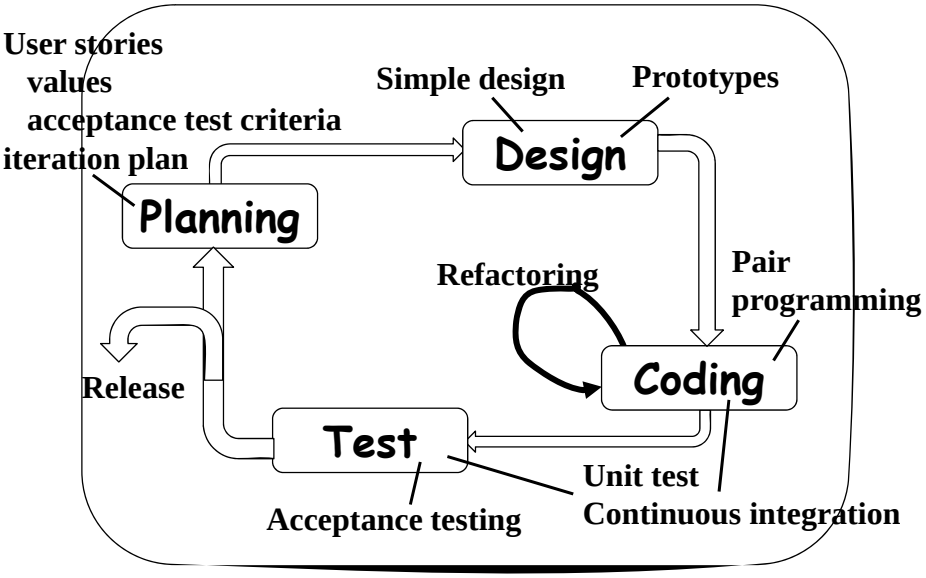
Extreme Programming (XP)

- **The most widely used agile process**
- **XP Planning**
  - **Begins with the creation of "user stories"**
  - **Agile team assesses each story and assigns a cost**
  - **Stories are grouped into deliverable increments**
  - **A commitment is made on delivery date**
  - **After the first increment "project velocity" is used to help define subsequent delivery dates for other increments**

Extreme Programming (XP)

- **XP Design**
  - **Follows the KISS principle**
  - **For difficult design problems, suggests the creation of "spike solutions"—a design prototype**
  - **Encourages "refactoring"—an iterative refinement of the internal program design**
- **XP Programming**
  - **Recommends the construction of unit tests *before* programming starts**
  - **Encourages "pair programming"**
- **XP Testing**
  - **All unit tests are executed daily**
  - **"Acceptance tests" are defined by the customer and executed to assess customer visible functionality**

Extreme Programming (XP)



Agile Software Development

1. Agile Process Overview
2. Extreme Programming
3. Refactoring
4. Refactoring Techniques

Refactoring

Refactoring is a disciplined process of changing a software system in such a way that it does not alter the external behavior of the code while at the same time improves its internal structure

- **(*Noun*) – A change made to internal structure of software to make it easier to understand and modify without changing its observable behavior**
- **(*Verb*) – To structure software by applying a series of refactorings without changing its observable behavior**

Refactoring

- **Basic metaphor**
  - **Start with an existing code base and make it better**
  - **Change the internal structure (in-the-small to in-the-medium) while preserving the overall semantics**
    - ***i.e.*, rearrange the "factors" but end up with the same final "product"**
- **The idea is that you should significantly improve the code :**
  - **Reducing near-duplicate code**
  - **Improved cohesion, less coupling**
  - **Improved parameterization, understandability, maintainability, flexibility, abstraction, efficiency, *etc* …**
- **This is much harder if the high level architecture of the software is poorly designed**

Refactoring : Why, When, Who?

- **Improve the Design**
  - **Without refactoring, the design of the program will decay**
  - **As people change code – changes to realize short-term goals or changes made without a full comprehension of the design of the code – the code loses its structure**
- **Make Software Easier to Understand**
  - **Several users of code – the computer, the writer, and the updater**
  - **The most important by far is the updater !**
  - **Who cares if the compiler takes a few more cycles to compile your code ?**
  - **If it takes someone 3 weeks to update your code that is a problem !!**

Refactoring : Why, When, Who?

- **Helps find faults**
  - **Part of refactoring code is understanding the code and putting that understanding back in**
  - **That process helps clarify the program**
  - **That clarification allows faults to be found**
- **Program faster**
  - **Refactor continuously as you develop**
    - **Every day, look at yesterday's work to see if it needs to be improved**
  - **Without a good design, you can progress quickly for a while, but soon poor design starts to slow you down**
  - **You spend time finding and fixing faults and understanding the system instead of adding new function**
  - **New features need more coding as you patch over patches**

Refactoring, Design and Performance

- **Refactoring complements design**
- **By doing some of the design "*in process*" programmers avoid the problems of over designing for reuse, flexibility, or extendibility that is never needed**
- **In the short term refactoring may make the code slower**
- **Optimize for performance separately**
- **Typically, only 10% of the software accounts for 90% of the execution time – only optimize that 10%**

Agile Software Development

1. **Agile Process Overview**
2. **Extreme Programming**
3. **Refactoring**
4. **Refactoring Techniques**

Refactoring "Catalog"

- ***Refactoring: Improving the Design of Existing Code*, by Martin Fowler (*et al.*), 1999, Addison-Wesley**
- **Clarifies and catalogs many of the strategies that good OO programmers have been doing for years**
- **22 "*bad smells*" … issues in the code that don't look quite right**
- **72 "*refactorings*" … ways to change the problems in the code**

Rules of Three

- **The first time you code a task, *just do it***
  - **Don't worry if it's not quite perfect or general**
- **The second time you code the same idea, *wince* and code it up again**
- **The third time you code the same idea, it's time to *refactor* !**
  - **Any programming construct can be made more abstract … but that's not necessarily a good thing**
    - **Generality (flexibility) costs too**
  - **Don't spin wheels designing and coding the most abstract system you can imagine**
    - **Practice Just-in-Time abstraction**
    - ***Expect* that you will be re-arranging your code constantly – that's a good thing**

Bad "Smell" #1 – Duplicated Code

- **Same expression in two methods in the same class?**
  - **Make it a private ancillary routine and parameterize it**

(Extract method)

- **Same code in two related classes?**
  - **Push commonalities into closest mutual ancestor and parameterize**
  - **Use *template method* DP for variation in subtasks**

(Form template method)

Bad "Smell" #1 – Duplicated Code (2)

- **Same code in two *unrelated* classes?**
  - **Should they be related?**
    - **Introduce abstract parent**

(Extract class, Pull up method)

  - **Does the code really belongs to just one class?**
    - **Make the other class into a client *(Extract method)***

(Extract method)

  - **Can you separate out the commonalities into a subpart or a functor or other function object?**
    - **Make the method into a *subobject* of both classes.**
    - ***Strategy* DP allows for polymorphic variation of methods-as-objects**

(Replace method with method object)

Method is Too Long

- **Often a sign of:**
  - **Trying to do too many things**
  - **Poorly thought out abstractions and boundaries**
  - ***Micromanagement* anti-pattern**
- **Best to think carefully about the major tasks and how they inter-relate – be aggressive!**
  - **Break up into smaller `private` methods within the class**

(Extract method)

  - **Delegate subtasks to sub-objects that "know best" (*i.e.*, template method DP)**

(Extract class/method, replace data value with object)

- **Fowler's heuristic:**
  - ***When you see a comment, make a method***
  - **Often, a comment indicates:**
    - **The next major step**
    - **Something non-obvious whose details detract from the clarity of the routine as a whole**
  - **In either case, this is a good spot to "break it up"**

Class is Too Large

- **Too many different subparts and methods**
- **Two-step solution :**
  1. **Gather up the little pieces into aggregate subparts**

(Extract class, replace data value with object)

  2. **Delegate methods to the new subparts**

(Extract method)

- **You might notice some unnecessary subparts that have been hiding in the forest**
- **Resist the urge to micromanage the subparts**
- **Exception : Library classes have large, fat interfaces (many methods, many parameters, lots of overloading)**
  - **That is okay if the methods are there to support flexibility**

Too Many Parameters

- **Long parameter lists make methods difficult for clients to understand**
- **This is often a symptom of**
  - **… trying to do too much**
  - **… too far from home ?**
  - **… with too many disparate subparts**
- **In 1980, structured programming taught parameterization as a cure for global variables**
  - **With modules / OOP, objects have mini-islands of state that can be reasonably treated as "global" to the methods**
  - **No need to pass a subpart of yourself as a parameter to your own method**

Too Many Parameters – Solution

- **Trying to do too much?**
  - **Break up into sub-tasks**

(Extract method)

- **… too far from home?**
  - **Localize passing of parameters; don't blithely pass down several layers of calls**

(Preserve whole object, introduce parameter object)

- **… with too many disparate subparts?**
  - **Gather up parameters into aggregate subparts**
  - **Your method interfaces will be much easier to understand!**

(Preserve whole object, introduce parameter object)

Divergent Changes

- Occurs when one class is changed in different ways for different reasons
- Likely, this class is trying to do too much and contains too many unrelated subparts
- Over time, some classes acquire details and even ownership of subparts that rightly belong elsewhere
- This is a sign of poor cohesion
  - Unrelated elements in the same container
- Solution :
  - Break it up, reshuffle, reconsider relationships and responsibilities

(Move method or field)

Shotgun Surgery

- The opposite of divergent change
  - Each time you want to make a single, seemingly coherent change, you have to change lots of classes in little ways
- Also a classic sign of poor cohesion
  - Related elements are not in the same container !
- Solution :
  - Look to do some gathering, either in a new or existing class

(Move method or field)

Feature Envy

- **A method seems more interested in another class than the one it's defined in**
  - ***e.g.*, a method `A::m()` calls lots of get/set methods of class `B`**
- **Solution:**
  - **Move `m()` (or part of it) into `B`!**

(Move method or field, extract method)

- **Exceptions:**
  - ***Visitor / iterator / strategy* where the whole point is to decouple the data from the algorithm**
    - **Feature envy is more of an issue when both `A` and `B` have interesting data**

Data Clumps

- **A set of variables that seem to "hang out" together**
  - ***e.g.*, passed as parameters, changed/accessed at the same time**
- **This usually means that a coherent sub-object is just waiting to be recognized and encapsulated**

```
void Scene::setTitle (string titleText, int titleX, int titleY,
                      Color titleColor){…}

void Scene::getTitle (string& titleText, int& titleX,
                      int& titleY, Color& titleColor){…}
```

- **A `Title` class is almost dying to be born**
- **If a client knows all these parameters, the client could more easily create its own classes**

Data Clumps (2)

- **Creating a new class will shorten and simplify parameter lists**
  - **Program is easier to read, understand and maintain**
  - **Class is conceptually simpler too**
- **Moving the data may create feature envy (the last "bad smell")**
  - **Iterate on the design …**

(Preserve whole object, extract class, introduce parameter object)

Primitive Obsession

- **All subparts of an object are instances of primitive types (`int, string, bool, double,` *etc.*)**

  For example: dates, currency, SIN, tel.#, ISBN, special string values
- **These small objects often have interesting and non-trivial constraints that can be modeled**

  For example: **fixed number of digits/chars, check digits, special values**
- **Solution:**
  - **Create some "small classes" that can validate and enforce the constraints**
  - **This makes your system more *strongly typed***

(Replace data value with object, extract class, introduce parameter object)

Switch Statements

- **Switch statements can often be redesigned with polymorphism**

```
Double getSpeed () {
   switch (_type) {
      case EUROPEAN:
         return getBaseSpeed();
      case AFRICAN:
         return getBaseSpeed() –
            getLoadFactor() * _numCoconuts;
      case NORWEGIAN_BLUE:
         if (_isNailed) return 0
         else return getBaseSpeed (_voltage);
   }
}
```

- **This displays a lack of understanding of the proper use of polymorphism and encapsulation**
- **Redesign as a polymorphic method of PythonBird**

(Replace conditional with polymorphism, replace type with subclasses)

Lazy Class

- **Classes that don't do much that's different from other classes**
- **If several sibling classes do not exhibit polymorphic behavioral differences, consider collapsing them back into the parent and adding some parameters**
- ***Lazy classes* are often legacies of ambitious design or a refactoring that removed interesting behavior from the class**

(Collapse inheritance / polymorphism hierarchy, inline class)

Speculative Generality

- *"We might need this one day …"*
  - That's okay … but did you really need it ?
  - Extra classes and features add complexity – decreasing maintainability
- XP philosophy
  - "As simple as possible but no simpler"
  - "Rule of three"
- Keep in mind that refactoring is an ongoing process
  - If you really need it later, you can add it back in

(Collapse hierarchy, inline class, remove parameter)

Message Chains

- Client asks an object, which asks a sub-object, which asks a sub-object, …
  - This multi-layer "drill down" may result in sub-sub-sub-objects being passed to the original requesting client
- The client must understand the object structure, even if it is going through several intermediaries !
- Need to rethink the abstraction …
  - Why is a deeply nested sub-part needed up above ?
  - Why is the sub-part so simple that it's useful so far from home ?

(Hide delegate)

Middle Person

- **"*All hard problems in software engineering can be solved by an extra level of indirection*"**
  - **Many OO design principles are some variation of this statement, although they are usually stated in more clever and elegant ways**
- **If most of a class's methods simply use services of delegate sub-objects, then something is wrong with this abstraction**
- **The behavior of an object should be more than the sum of its parts !**

(Remove middle person, replace delegation with inheritance)

Inappropriate Intimacy

- **Sharing of secrets between classes, especially outside of inheritance**
  - **`public` variables, too many get / set methods, C++ friendship, `protected` data in classes, …**
- **Leads to data coupling, intimate knowledge of internal structures and implementation decisions**
  - **Makes clients brittle, hard to evolve, easy to break**
- **Solution :**
  - **Appropriate use of get / set methods**
  - **Rethink basic abstraction**
  - **Merge classes when it helps**

(Move/extract method/field, change bidirectional association to unidirectional, hide delegate)

Alternative Classes with Different Interfaces

- **Classes and methods seem to implement the same or similar abstraction – yet are otherwise unrelated**
  - **This is not a criticism of overloading, just haphazard design**
- **Solution :**
  - **Move the classes "closer" together**
  - **Find a common interface**
  - **Find a common subpart and remove it**

(Extract [super] class, move method/field, rename method)

Refused Bequest

- **Subclass inherits methods and variables but does not use some of them**
  - **Sometimes this is a good sign : the parent manages the common behaviors and the child manages the differences**
  - **Need to look at clients to see if clients use the class and its parent like that**
  - **Do clients use parent's methods ?**
- **Did the subclass inherit simply to get some functionality cheaply?**
  - **If so, better to use delegation**

(Replace inheritance with delegation)

- **Parent has features that only some children use**
- **Create more intermediate abstract classes in the hierarchy**
  - **Move the methods down one level**

(Push down field or method)

Comments

- **Comments are essential to readability and maintainability**
  - **They are also pretty helpful during debugging !**
- **Very long comments, however, are sometimes a sign that the code is too long, complicated, and impossible to maintain**
  - **Comments should be used to explain *why*, not *what***
- **Instead of explaining code that is too hard to read, restructure it so people can use it !**

(Restructure complicated logic)

Summarizing Refactoring

- **Instead of thinking of maintenance as something that happens**
  - **… separately from programming …**
  - **… in response to needs for change …**
  - **… by someone else …**
- **Think of *refactoring* as a process of**
  - **Continuously …**
  - **Smoothly …**
  - **Improving the software by the developer**

Only fools (and software engineering professors) think programmers can design and build all the software right the first time

Summary

- **The 1980-style view of software development …**
  - **analyze … design … program … test … deploy … maintain …**
- **Is as out of date as punk music, portable CD players and floppy disks !**
- **We can not effectively find problems in designs until we write the program**
- **If we want to build integrated collections of continuously *evolving cities*, we must view software development as**

continuous evolution

1. **The process must be agile**
2. **Testing must be seamlessly integrated with development**
3. **Software design must be continuously evolved and refactored**