Refactoring
What?

“Improving a computer program by reorganising its internal structure without altering its external behaviour.”

(http://dictionary.reference.com/browse/refactoring)
Why?

- Improves the overall design of software
- Makes software easier to understand
- Helps you find bugs
- Helps you program faster
Improves

- Parameterization
- Understandability
- Maintainability
- Flexibility
- Abstraction
- Efficiency
- etc ...
- Reforestation (marked with a cross)
When?

Martin Fowler in the Book; *Refactoring: Improving the Design of Existing Code* says;

- Refactor when you recognize a warning sign (a “bad smell”) and know what to do
- Otherwise use the “Rule of three” by Don Roberts
  - Refactor when you add function
  - Refactor when you need to fix a bug
  - Refactor as you do a code review
“Rule of three” (alternate)

- The first time you code a task, just do it.
- 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!
Limits?

- Any programming construct can be made more abstract ... but that’s not necessarily a good thing.
  - Why?
- Don’t spin you wheels designing and coding the most abstract system you can imagine.
  - Practice Just-in-Time abstraction.
  - *Expect that you will be rearranging your code constantly. Don’t worry about it. Embrace it.*
Bad smells in code
God class

god. Sum(2,3,)

god. Open (file)
God class

- AKA Large class, The Blob
- Class is trying to do too much.
- Too many instance variables.
- Too many different subparts and methods
God class - solution

- Two step solution:
  - Gather up the little pieces into aggregate subparts.
  - Delegate methods to the new subparts.
- Likely, you’ll notice some unnecessary subparts that have been hiding in the forest!
- Resist the urge to micromanage the subparts!
God class

- Counter example:
  - Library classes often have large, fat interfaces (many methods, many parameters, lots of overloading)
Duplicated code

- “The #1 bad smell”
- If you see the same code structure in more than one place, you can be sure that your program will be better if you find a way to unify them.
- Why?
Duplicated code - solution

- The simplest duplicated code problem is when you have the same expression in two methods of the same class
  - Perform Extract Method and invoke the code from both places
Duplicated code - solution

- Same code in two related (sibling or otherwise) classes?
  - Push common features into closest mutual ancestor and parameterize
  - Use模板方法Design Pattern for variation in subtasks
  - In other words; perform Extract Method in both classes then Pull Up Field
Duplicated code - solution

- Same code in two *unrelated classes*?
  - Should they be related?
    - Introduce abstract parent
    - Extract class, Pull up method
Duplicated code - solution

- Same code in two *unrelated classes*?
  - Does the code really belongs to just one class?
    - Make the other class into a client
    - Extract method
Duplicated code - solution

- Same code in two unrelated classes?
  - 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 Design Pattern allows for polymorphic variation of methods-as-objects
    - Replace method with method object
Long method

• The longer a procedure is the more difficult it is to understand.

• Often a sign of:
  • Trying to do too many things
  • Poorly thought out abstractions and boundaries
  • Micromanagement anti-pattern
Long method - solution

- 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 methods
- Delegate subtasks to subobjects that “know best” (i.e., template method DP)
  - Extract class/method, Replace data value with object
Long method - solution

- 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”. 
Long parameter list

- 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
Long parameter list

- In the old days, structured programming taught the use of 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 (yet are still hidden from the rest of the program).
  - i.e., You don’t need to pass a subpart of yourself as a parameter to one of your own methods.
Long parameter list - Solution

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

- ... too far from home?
  - Localize passing of parameters; don’t 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 change

- Occurs when one class is commonly changed in different ways for different reasons
- Likely, this class is trying to do too much and contains too many unrelated subparts
- This is a sign of *poor cohesion*
  - Unrelated elements in the same container
Divergent change

- Over time, some classes develop a "God complex"
  - They acquire details/ownership of subparts that rightly belong elsewhere
Divergent change - Solution

- Break it up
- Reshuffle
- Reconsider relationships and responsibilities
- Extract class
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!
Shotgun surgery - Solution

- Look to do some gathering, either in a new or existing class.
  - You have to use *Move Method* and *Move Field* to put all the changes in a single class
  - If no current class looks like a good candidate then create one
  - Often you can use *Inline Class* to bring a whole bunch of behaviour together
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`
- Feature envy is more of an issue when both `A` and `B` have interesting data
Feature envy - Solution

- The method clearly wants to be elsewhere, so use *Move Method* to get it there.

- Sometimes only part of the method suffers from envy so in that case you can use *Extract Method* on the jealous bit and *Move Method* to get it home.
Feature envy - Exceptions

- Some design patterns that are used to combat the divergent change smell needs this smell to be present.
  - Strategy (Gang of Four)
  - Visitor (Gang of Four)
  - Self Delegation (Beck)
- In these patterns whole point is to decouple the data from the algorithm
Data clumps

- You see a set of variables that seem to “hang out” together
  - e.g., passed as parameters, changed/accessed at the same time

- Usually, this means that there’s a coherent subobject waiting to be recognized and encapsulated
Data clumps - Solution

```cpp
void Scene::setTitle (string titleText,
                     int titleX, int titleY,
                     Colour titleColour){...}

void Scene::getTitle (string& titleText,
                      int& titleX, int& titleY,
                      Colour& titleColour){...}
```

- In the example, a Title class is dying to be born.

- If a client knows how to change a title’s x, y, text, and colour, then it knows enough to be able to “roll its own” Title objects.
  - However, this does mean that the client now has to talk to another class.

- The first step is to look for where the clumps appear as fiels and use Extract Class to turn the clumps into an object.
- For method parameters use Introduce Parameter Object or Preserve Whole Object to slim them down.
Data clumps - Solution

- The first step is to look for where the clumps appear as fields and use *Extract Class* to turn the clumps into an object.
- For method parameters use *Introduce Parameter Object* or *Preserve Whole Object* to slim them down.

```c++
void Scene::setTitle (string titleText, 
                     int titleX, int titleY, 
                     Colour titleColour){...}

void Scene::getTitle (string& titleText, 
                      int& titleX, int& titleY, 
                      Colour& titleColour){...}
```
Data clumps - Remarks

- This will greatly shorten and simplify your parameter lists (which aids understanding) and makes your class conceptually simpler too.
- Moving the data may create feature envy initially!
  - May have to iterate on the design until it feels right.
Primitive obsession

- All subparts of an object are instances of primitive types
  - `int`, `string`, `bool`, `double`, etc.
- Examples?
  - `Dates`, `currency`, `SIN`, `tel#`, `ISBN`, `special string values`
  - `(Address, Name)`
- Often, these small objects have interesting and non-trivial constraints that can be modelled
  - `e.g., fixed number of digits/chars, check digits, special values`
Primitive obsession - Solution

- Create some “small classes” that can validate and enforce the constraints.
- This makes your system mode *strongly typed*.
Switch statements

- This is an example of a lack of understanding polymorphism and a lack of encapsulation.
- Solution?
  
  *Redesign as a polymorphic method*

- Replace conditional with polymorphism, replace type code with subclasses
Lazy class

- Classes that doesn’t do much that’s different from other classes.
- Each class you create costs money and time to maintain and understand.
- A class that is not carrying its weight should be eliminated.
- Often, lazy classes are legacies of ambitious design or a refactoring that gutted the class of interesting behaviour.
Lazy class - Solution

- If there are several sibling classes that don’t exhibit polymorphic behavioural differences, consider just collapsing them back into the parent and add some parameters.
- If you have subclasses that are not doing enough try to use "Collapse Hierarchy" and nearly useless components should be subjected to "Inline Class".
Speculative generality

- *We might need this one day ...*
  - Fair enough, but did you really need it after all?
  - Extra classes and features add to complexity.
- XP philosophy:
  - “As simple as possible but no simpler.”
  - “Rule of three”.
- Keep in mind that refactoring is an ongoing process. (If you really do need it later, you can add it back in.)
- This smell is easily detected when the only users of a class or method are test cases
Speculative generality - Solution

- If you have abstract classes that are not doing enough then use Collapse Hierarchy
- Unnecessary delegation can be removed with Inline Class
- Methods with unused parameters should be subject to Remove Parameter
- Methods named with odd abstract names should be repaired with Rename Method
Message chains

- Client asks an object which asks a subobject, which asks a subobject, ...
  - Multi-layer “drill down” may result in sub-sub-sub-objects being passed back to requesting client.
- This smell may appear as a long line of *getThis* methods, or as a sequence of temps
- Probably need to rethink abstraction ... 
  - Why is a deeply nested subpart surfacing?
  - Why is the subpart so simple that it’s useful far from home?
Message chains - Solution

- The move to use in this case is *Hide Delegate* at various points in the chain
Inappropriate intimacy

- Sharing of secrets between classes, esp. outside of the holy bounds of inheritance
  - public variables, indiscriminate definitions of 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.
Inappropriate intimacy - Solution

- Appropriate use of get/set methods
- Rethink basic abstraction.
- Merge classes if you discover “true love”
- Use Move Method and Move Field to separate the pieces to reduce the intimacy
- If the classes do have common interests, use Extract Class to put the commonality in a safe place or use Hide Delegate to let another class act as a go-between
Middle man

- “All hard problems in software engineering can be solved by an extra level of indirection.”
  - OODPs pretty well all boil down to this, in quite clever and elegant ways.
- If you notice that many of a class’s methods just turn around and beg services of delegate subobjects, the basic abstraction is probably poorly thought out.
Middle man

- An object should be more than the sum of its parts in terms of behaviours!
- Encapsulation often comes with delegation
- Sometimes delegation can go too far
Middle man - Solution

- If you find half the methods are delegated to another class it might be time to use *Remove Middle Man* and talk to the object that really knows what is going on
- If only a few methods are not doing much, use *Inline Method* to inline them into the caller
- If there is additional behavior, you can use *Replace Delegation with Inheritance* to turn the middle man into a subclass of the real object
Refused bequest – type 1

- Subclass inherits methods/variables but doesn’t seem to use some of them.
- In a sense, this might be a good sign.
- Might want to look at typical client use to see if clients think child is-a parent.
- Did the subclass inherit as a cheap pickup of functionality?
  - Fowler/Beck claim this isn’t as bad a smell as the others.
  - Might be better to use delegation.
Refused bequest – type 2

- Parent has features that are used by only some of its children.
- Typical solution is to create some more intermediate abstract classes in the hierarchy.
- Move the peculiar methods down a level.
Data class

- Class consists of (simple) data fields and simple accessor/mutator methods only.
  - Often, you’ll find that clients of this class are using get/set methods just like the micromanager anti-pattern (via a level of indirection).
- “Data classes are like children. They are OK as a starting point, but to participate as a grownup object, they need to take on some responsibility.”
Data class - Solution

- Have a look at usage patterns in the clients
- Try to abstract some commonalities of usage into methods of the data class and move some functionality over
Comments!

- XP philosophy discourages comments, in general:
  - Instead, make methods short and use long identifiers
  - Fowler is not so critical.
- In the context of refactoring, Fowler claims that long comments are often a sign of opaque, complicated, inscrutable code.
  - They aren’t against comments so much as in favour of self-evident coding practices.
  - Rather than explaining opaque code, restructure it!
Comments! - Solution

- Rule of thumb –
  - A method should be considered “long” if more than one line of comments to describe its functionality.
  - Comments are best used to document rationale (i.e., explain why you picked one approach over another.)
References

- [http://classes.seattleu.edu/computer_science/csse514/klawitter/lectures/RefactoringPrincipals.ppt](http://classes.seattleu.edu/computer_science/csse514/klawitter/lectures/RefactoringPrincipals.ppt)
- “Architectural Patterns” presentation by Mr. Samisa Abeysinghe, VP, Engineering, WSO2