Refactoring
CSC 4700 Software Engineering

Lecture 4

Based on Fowler “Refactoring” and UWaterloo slides

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 improve the code in some significant way. For example:
    • Reducing near-duplicate code
    • Improved cohesion, lessened coupling
    • Improved parameterization, understandability, maintainability, flexibility, abstraction, efficiency, etc...

Some advice from Fowler

• “When should I refactor? How often? How much time should I dedicate to it?”
  • It’s not something you should dedicate two weeks for every six months …
  • … rather, you should do it as you develop!
  • Refactor when you recognize a warning sign (a “bad smell”) and know what to do
  • … when you add a function or method
    • Likely it’s not an island unto itself
  • … when you fix a bug
  • Is the bug symptomatic of a design flaw?
  • … when you do a code review
    • A good excuse to re-evaluate your designs, share opinions.
The rule of three (XP)

- The first time you code a task, just do it.
- The second time you code the same idea, refine 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 your 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. Don’t worry about it. Embrace it.

Bad smells in code

- Duplicated code
  - "The #1 bad smell"
  - Same expression in two methods in the same class?
    - Make it a private auxiliary routine and parameterize it
      (Extract method)
  - Same code in two related classes?
    - Push commonalities into closest mutual ancestor and parameterize
      (Form template method)
  - Same code in two unrelated classes?
    - Ought they be related?
      - Introduce abstract parent (Extract class, Pull up method)
    - Does the code really belong to just one class?
      - Make the other class into a client (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)
Bad smells in code

- **Long method**
  - 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
    - Delegate subtasks to subobjects that "know best" (i.e., template method DP)
      - (Extract method)
      - (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".

- **Large class**
  - i.e., 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)
  - Likely, you’ll notice some unnecessary subparts that have been hiding in the forest!
  - Resist the urge to micromanage the subparts!
Bad smells in code

- **Large class**
  - Counter example:
    - Library classes often have large, fat interfaces (many methods, many parameters, lots of overloading)
    - If the many methods exist for the purpose of flexibility, that's OK in a library class.

Bad smells in code

- **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

Bad smells in code

- **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.
Bad smells in code

- **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 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]

Bad smells in code

- **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.
  - Over time, some classes develop a "God complex":
    - They acquire details/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
      - [Extract class]

Bad smells in code

- **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/field]
Bad smells in code

- **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/field, extract method)
  - **Exceptions:**
    - Visitor/iterator/strategy DP 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.

Bad smells in code

- **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 just waiting to be recognized and encapsulated.

```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.
  - 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.
    - (Preserve whole object, extract class, introduce parameter object)
Bad smells in code

• Primitive obsession
  • All subparts of an object are instances of primitive types (int, string, bool, double, etc.)
    e.g., dates, currency, SIN, tel.#, ISBN, special string values
  • Often, these small objects have interesting and non-trivial constraints that can be modeled
    e.g., 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)

Bad smells in code

• Switch statements
  • We saw this before; here’s Fowler’s example:

```java
Double getSpeed () {
  switch (_type) {
  case EUROPEAN:
    return getBaseSpeed();
  case AFRICAN:
    return getBaseSpeed() -
      getLoadFactor() * _numCoconuts;
  case NORWEGIAN_BLUE:
    return (_isNailed) ? 0
                          : getBaseSpeed(_voltage);
  }
}
```

Bad smells in code

• Switch statements
  • This is an example of a lack of understanding polymorphism
    and a lack of encapsulation.
  • Solution:
    • Redesign as a polymorphic method of PythonBird
      (Replace conditional with polymorphism, replace type code with subclasses)
Bad smells in code

- **Lazy class**
  - Classes that don’t do much that’s different from other classes.
  - If there are several sibling classes that don’t exhibit polymorphic behavioural differences, the consider just collapsing them back into the parent and add some parameters.
  - Often, lazy classes are legacies of ambitious design or a refactoring that gutted the class of interesting behaviour. (Collapse hierarchy, inline class)

Bad smells in code

- **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. (Collapse hierarchy, inline class, remove parameter)

Bad smells in code

- **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.
  - Sounds like the client already has an understanding of the structure of the object, even if it is going through appropriate intermediaries.
  - 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? (Hide delegate)
Bad smells in code

- **Middle man**
  - “All hard problems in software engineering can be solved by an extra level of indirection.”
  - OOPs pretty well all boil down to this, albeit 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.
  - An object should be more than the sum of its parts in terms of behaviours!
    (Remove middle man, replace delegation with inheritance)

Bad smells in code

- **Inappropriate intimacy**
  - Sharing of secrets between classes, esp. outside of the holy bounds of inheritance
    e.g., 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.
  - Solution:
    - Appropriate use of get/set methods
    - Rethink basic abstraction.
    - Merge classes if you discover "true love"
      (Move/extract method/field, change bidirectional association to unidirectional, hide delegates)

Bad smells in code

- **Alternative classes with different interfaces**
  - Classes/methods seem to implement the same or similar abstraction yet are otherwise unrelated.
  - This is not a knock against overloading, just haphazard design.
  - Solution:
    - Move the classes "closer" together.
      - Find a common interface, perhaps an ABC.
      - Find a common subpart and remove it.
    (Extract [super]class, move method/field, rename method)
Bad smells in code

- **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 (albeit via a level of indirection).
  - 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.
    - My own view is that data classes are quite reasonable, if used judiciously.
    - In C++, often use structs to model data classes.
  - “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.”

- **Refused bequest**
  - Subclass inherits methods/variables but doesn’t seem to use some of them.
  - In a sense, this might be a good sign:
    - The parent manages the commonalities and the child manages the differences.
    - Might want to look at typical client use to see if clients think child is a parent
    - Did clients use parent’s methods? … use parent as static type?
  - 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
(Replace inheritance with delegation)

- **Refused bequest**
  - Another perspective:
    - 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.
(Replace inheritance with delegation)