SWEN 262

Engineering of Software Subsystems

The Laws of Software Evolution

- Beginning in 1974, Manny Lehman and Laszlo Belady began documenting the *laws of software evolution*.
- There are 8 laws in total, but the first two are as follows:
  - **Continuing Change** - Systems must be continually adapted else they become progressively less satisfactory.
  - **Increasing Complexity** - As a system evolves, its complexity increases unless work is done to maintain or reduce it.

In other words, over time any software system must change to add new improvements (i.e. features) or it will become out of date and/or unusable.

At the same time, introducing change to a software system also makes it more complex.

The more complex software is, the harder it is to understand and maintain.

That is unless the engineers make a specific effort to maintain or reduce complexity in some way...

Refactoring

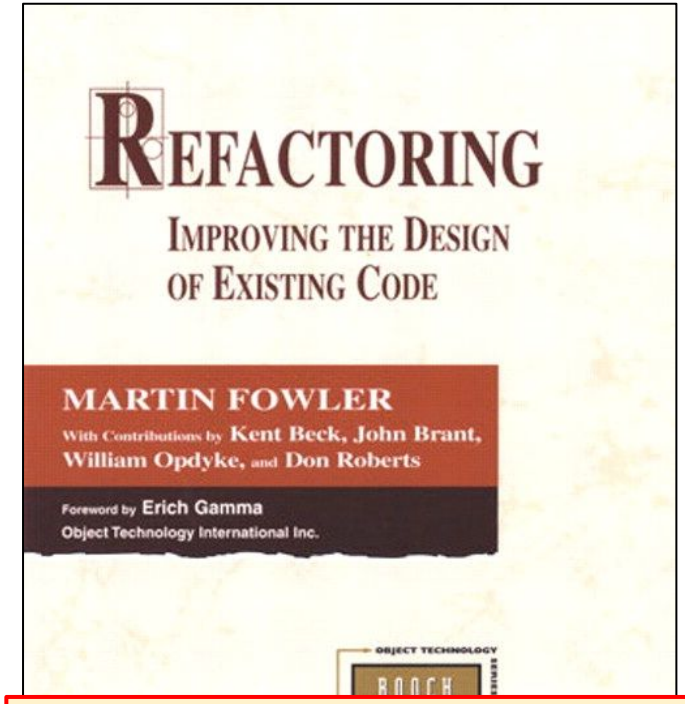
Refactoring is taking software, which through natural processes has lost its original clean structure...

Refactoring

...and restoring a clean structure.

The Fowler Book

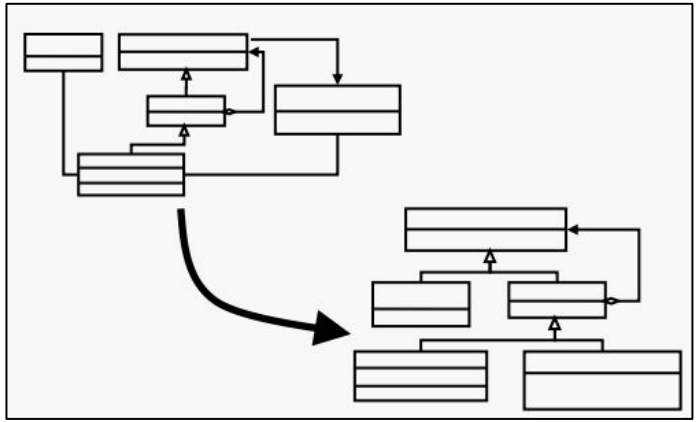
- The definitive guide to refactoring is a book by Martin Fowler.
- *Refactoring: Improving the Design of Existing Code*
  - Martin Fowler, Addison-Wesley, 1999
- The book contains more than 70 recipes for refactoring.
  - Each “recipe” contains a set of *refactoring steps* that should be completed in order to implement a specific refactoring.
  - In this way, *Refactoring* is a sort of cookbook for cleaning up legacy code.



Fowler’s refactoring.com site has a catalog of refactorings as well as other useful resources.

Refactoring

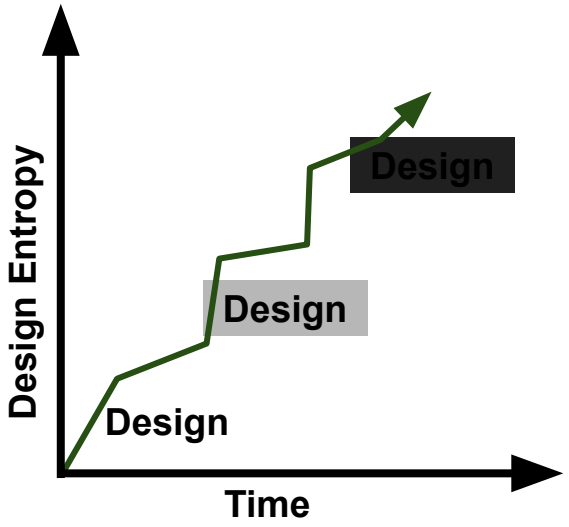
- Refactoring should only change the *internal structure* and not the *observable behavior* of a system.
- This bears repeating: refactoring should change the *internal structure* and not the *observable behavior* of a system.
  - This includes the user interface!
- Remember: adding new features to a system *increases* its complexity and makes the system more difficult to understand and maintain.
- The goal of refactoring is to *reduce* complexity.



Refactoring (noun): a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior.

Design Entropy

- The *design entropy* of a software system tends to increase over time.
  - entropy (noun): a process of degradation or running down to a trend to disorder.
  - also: chaos, disorganization, randomness.
- As the code is modified (e.g. to add new features, fix bugs, etc.) it moves farther and farther away from its original design.



If you no longer can see the design, how can you stay consistent to it?

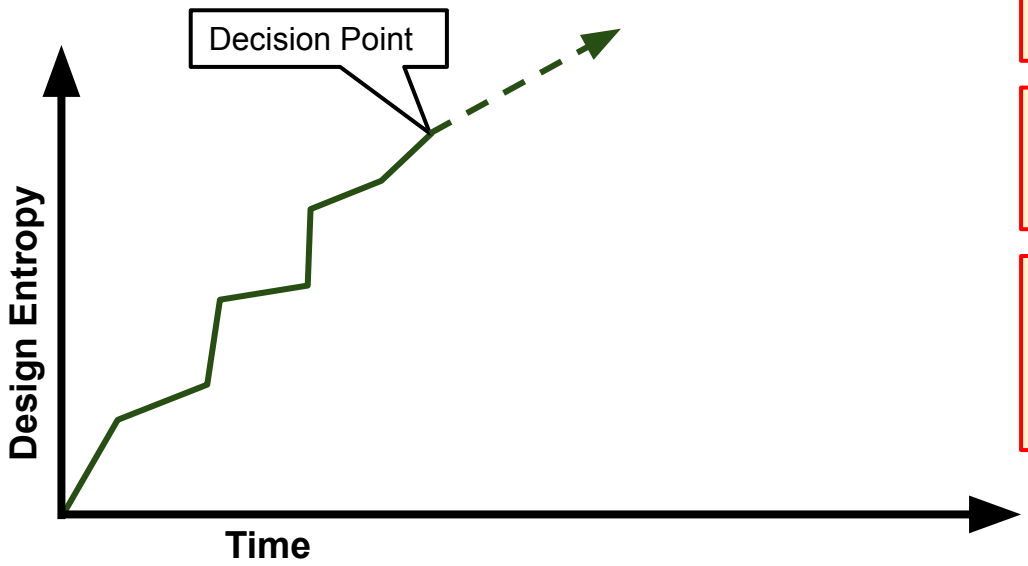
What if the original design is no longer adequate?

Design Entropy

- The entropy will increase because of the typical development death spiral.
  - Good design up front.
  - Local modifications alter the framework.
    - Small changes add up.
  - Short-term goals win out over structure maintenance.
    - Fix bugs.
    - Meet deadlines.
  - Engineering sinks into hacking.
    - Must...code...faster!
  - Integrity and structure fade (entropy).

Refactoring

- A refactoring activity can remove some of that design randomness.



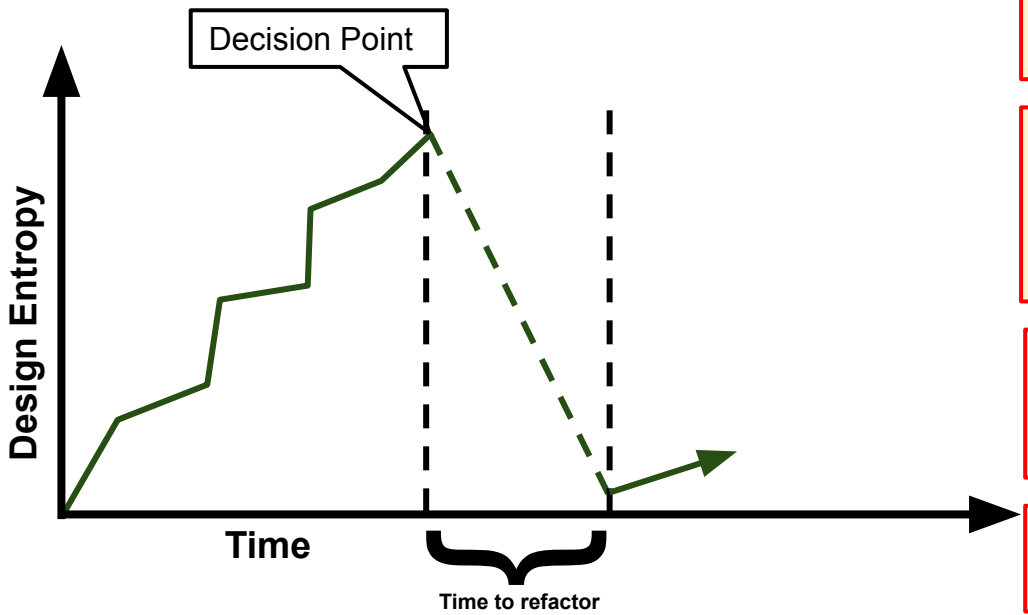
When adding a new feature, you arrive at a decision point.

Option 1: Business as usual. Hack the new feature into the system and increase the entropy.

The system moves farther from the original design, and you risk breaking some of the other features by ***introducing new bugs***.

Refactoring

- A refactoring activity can remove some of that design randomness.



Option 2: Refactor the existing code to a design into which the new feature will integrate more smoothly.

Note that the entropy in the system decreases with the refactoring, but the design has still changed from its original structure!

It's important to consider that *refactoring takes time*. It is not free. Features will take longer to deliver.

This is why many engineers make the excuse not to refactor...

If it ain't broke...

- It can be difficult to counter the "If it ain't broke, don't fix it!" mentality.
- Sure, the design may be:
  - Ugly.
  - Difficult to understand.
  - Difficult to maintain.
  - Difficult to modify.
  - Difficult to debug.
- But! It mostly works, and refactoring is *dangerous* and *takes time*.
  - Significant modifications to the design pose a risk that everything will break.
  - Time is money.

Refactoring

- But code that can't be maintained, debugged, or modified without serious risk ***is*** broken.
- In general, refactoring...
  - Improves the quality of the product.
  - Pays today to ease work tomorrow.
  - May actually accelerate today's work!

But *time is money*. How can spending time today save time later?

Good question! Let's talk about code debt...

Code Debt

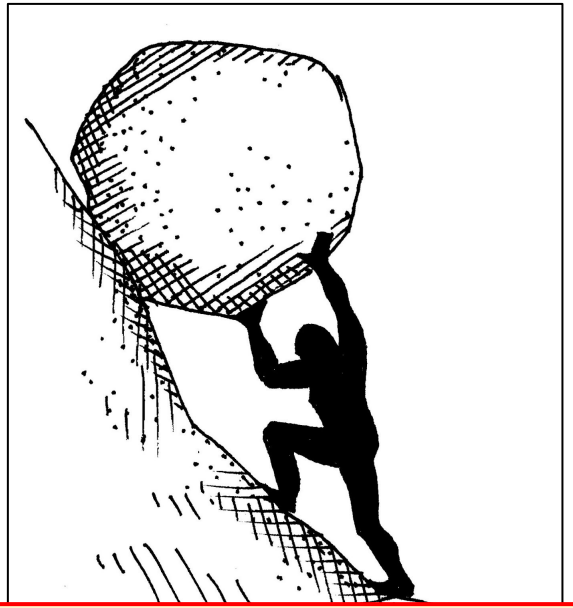
Ward Cunningham used *debt* as a metaphor for software development:

> *"Shipping first time code is a little like going into debt. A little debt speeds development so long as it is paid back promptly. Objects make the cost of this transaction tolerable.*
>
> *"The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt.*
>
> *"Entire engineering organizations can be brought to a stand-still under the load of an unconsolidated implementation, object-oriented or otherwise."*

But what does this mean?

Code Debt

- Taking shortcuts or risks during software development accrue a small amount of debt.
  - Hacking new features into an existing design.
  - Skipping unit testing.
  - Writing a line of code!
- Eventually, *interest* must be paid on that debt in the form of the *time* it takes to work around the problems introduced by the shortcuts.
  - Fixing bugs.
  - Deciphering inscrutable code.
  - Difficulty in adding new features.
- Some organizations end up spending *most or all* of their time *paying interest on technical debt*.



The system becomes so difficult to maintain that the organization spends all of its time fixing problems rather than introducing new features.

Refactoring

- Refactoring does not work well as an end task because there is never any time to do it.
  - Will the customer *pay* for you to spend lots of *time* to produce a product that has changed *internally* but where the observable features have remained the same?
- Refactoring may be a continuous code improvement activity *if*...
  - It will make adding a new feature easier.
  - It will make the code easier to debug.
  - It fills in a “design hole.”
  - It is done as a result of a code inspection.
  - If it simply makes the code easier to understand.

Time ***is*** money. But sometimes spending a *little* money **now** saves a *lot* of money **later**.

Code Inspections

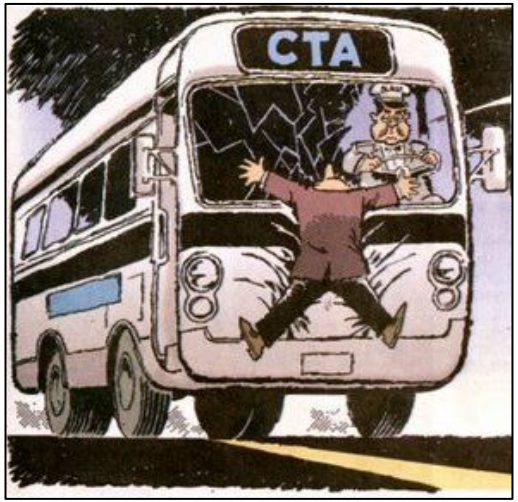
- Code inspections have been found to be the most effective technique for early defect detection.
  - Spreads design and implementation knowledge through the team.
  - Helps more experienced engineers mentor less experienced developers.
  - New eyes see things “old” eyes are not seeing.
  - Next time you can’t find a bug, inspect!

The ultimate form of code inspection is pair programming.

One developer performs a *continuous code inspection* as the other developer codes.

Bus Number

- A development team's *bus number* is the answer to the following question: "How many team members need to be hit by a bus before you lose critical knowledge about part of the system?"
  - Obviously, the worst answer to the question is "one."
  - If a single member of the team becomes unavailable, there is no one else that could quickly and easily pick up where that person left off.
- Code inspection, including pair programming, is a mechanism for increasing your bus number.
  - This helps to avoid "siloing."
  - This also helps the team work with more *agility* because any team member can take any task, even if (or especially when) the rest of the team is busy.



Team members don't need to *actually* be hit by a bus.

They could also go on vacation, be stuck in training, or leave for another job (for example).

Smells: When to Refactor

- There are many bad smells that get designed and coded into software.

- Duplicated code
- Long methods
- Long parameter lists
- Orthogonal purposes for a class
- Shotgun changes
- Feature envy
- Data clumping
- Primitive object avoidance
- Switch statements
- Type codes
- Speculative generality
- Middle man overuse
- Data classes
- Verbose comments

Not all smells are necessarily bad.

But they can be an indication of a problem in the system.

A simple rule that applies to code and diapers: *if it stinks, change it.*

Duplicated Code

- Rule of three.
  - Do something in one place, that's OK.
  - Do something in two places, hold your nose and go ahead.
  - Do something in three places… time to refactor.
- If the same code exists in two or more places, it may cause problems.
  - A bug in one place is a bug in all of them.
  - Modifications made to one need to be made to the others.
  - Code is longer (this is a smell).
- In this case, the problem can be solved using the *extract method* refactor.

2

Some developers practice the *rule of two*.

Extract Method: Refactoring Steps

- Create a new method.
- Copy the extracted code into the method.
- Look for *local* variables on which the extracted code depends, and add them as parameters to the method.
- Replace the original code with a call to the method.
    - Be sure to pass in the required *local variables* as *parameters*.

This is an abbreviated version of the actual refactoring steps from the *Fowler* book.

See the *Extract Method* refactoring on page 110 for full details.

Extract Method: Refactoring Steps

```
public class MyClass {

      // somewhere in the code...
      for(String name:listOfNames) {
        System.out.println(name);
      }

      // somewhere else in the code...
      for(String name:listOfNames) {
        System.out.println(name);
      }









}
```

Extract Method: Refactoring Steps

```
public class MyClass {

      // somewhere in the code...
      for(String name:listOfNames) {
        System.out.println(name);
      }

      // somewhere else in the code...
      for(String name:listOfNames) {
        System.out.println(name);
      }

}
```

Identify duplicate code that exists in more than one place (usually 3, but 2 is OK, too).

(obviously this is an overly simple example, but you get the idea)

Extract Method: Refactoring Steps

```
public class MyClass {

      // somewhere in the code...
      for(String name:listOfNames) {
        System.out.println(name);
      }

      // somewhere else in the code...
      for(String name:listOfNames) {
        System.out.println(name);
      }

      public void printNames() {

      }


}
```

Create a new method with a name that captures the method's *intent*.

Extract Method: Refactoring Steps

```
public class MyClass {

      // somewhere in the code...
      for(String name:listOfNames) {
        System.out.println(name);
      }

      // somewhere else in the code...
      for(String name:listOfNames) {
        System.out.println(name);
      }

      public void printNames(List<String> listOfNames) {

     }


}
```

Look for *local variables* on which the code depends...

...and add those variables as parameters to the new method.

Extract Method: Refactoring Steps

```
public class MyClass {

      // somewhere in the code...
      for(String name:listOfNames) {
        System.out.println(name);
      }

      // somewhere else in the code...
      for(String name:listOfNames) {
        System.out.println(name);
      }

      public void printNames(List<String> listOfNames) {
        for(String name:listOfNames) {
          System.out.println(name);
        }
      }

}
```

Copy and paste the original code into the new method.

Extract Method: Refactoring Steps

```
public class MyClass {

      // somewhere in the code...
      printNames(listOfNames);

      // somewhere else in the code...
      printNames(listOfNames);




      public void printNames(List<String> listOfNames) {
        for(String name:listOfNames) {
          System.out.println(name);
        }
      }

}
```

Finally, replace the original code with a call to the new method.

Q: This is a very simple example. What other variations might need to be considered?

A: What about temporary variables? What if the method changes the value of some variable used later?

*(by the way, most modern IDEs include built-in macros to handle common refactorings like **extract method**).*

Safely refactoring

- Refactoring is often *dangerous*.
- More than adding a simple feature, refactoring involves *changing the design of the system*.
- Before refactoring, smart developers write *characterization tests*.
- A *characterization test* is a unit test that verifies the current functionality of existing software.
  - Unlike many unit tests, characterization tests are written *after* the code is already working.
- Once the code is characterized with characterization tests it should be safe to modify.
  - If the tests pass, great!
  - If the tests break, roll back the change!

Test Driven Development (TDD) states: never modify a line of code before it is under test.

This is true for ***legacy code*** that needs to be refactored as well as new code.

If the ***legacy code*** is not yet under test, it needs to be brought under test before the refactoring can begin.

This means writing unit tests to *characterize* the current functionality (which is theoretically working as intended).

Once the code is under test, the refactor can begin and the tests run to make sure the refactor didn't break the code.

Refactoring Your Design

- This semester you will be asked to evaluate your design from Release 1 of the Design Project and complete at least one major refactoring.
  - Run a metric analysis of your code to identify hot spots for a potential refactoring.
  - Apply at least one significant design pattern (any of the GoF patterns is eligible) to a part of your R1 design that doesn't currently make use of one.
  - Implement your refactored design as part of you Design Project R2.