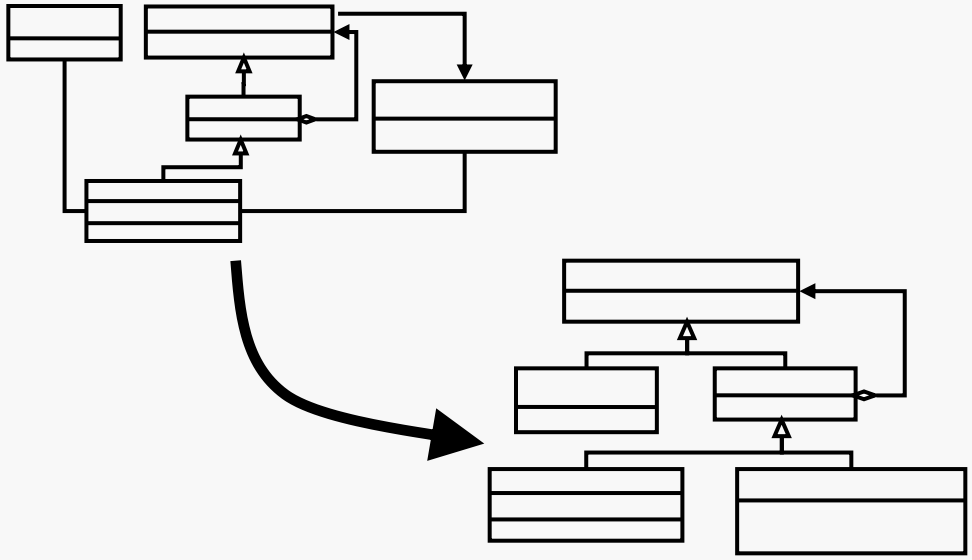
Refactoring

4010-362
Engineering of
Software Subsystems



Lehmann & Belady: Laws of Software Evolution

1. **Continuing Change** - Systems must be continually adapted else they become progressively less satisfactory.
2. **Increasing Complexity** - As a system evolves its complexity increases ***unless work is done to maintain or reduce it.***

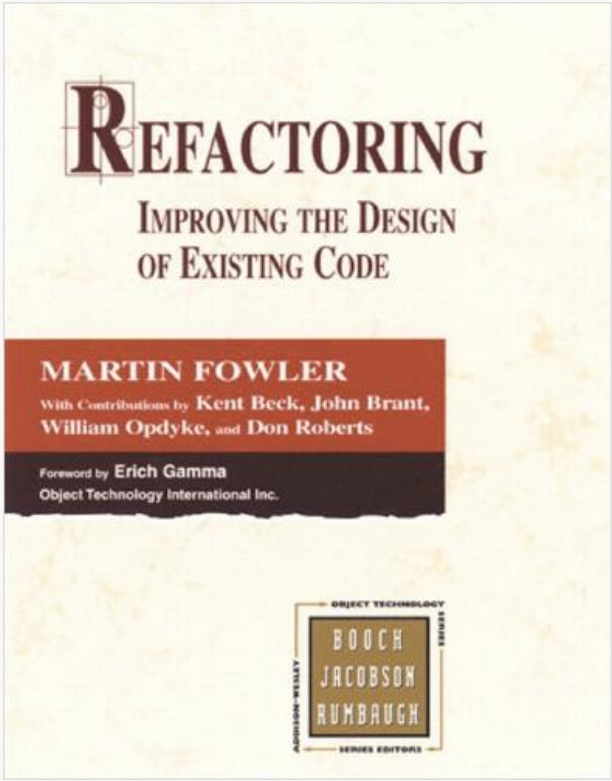
Refactoring is taking software which through natural processes has lost its original clean structure…

…and restoring a clean structure.

The definitive guide to refactoring is a book by Martin Fowler.

Refactoring: Improving the Design of Existing Code

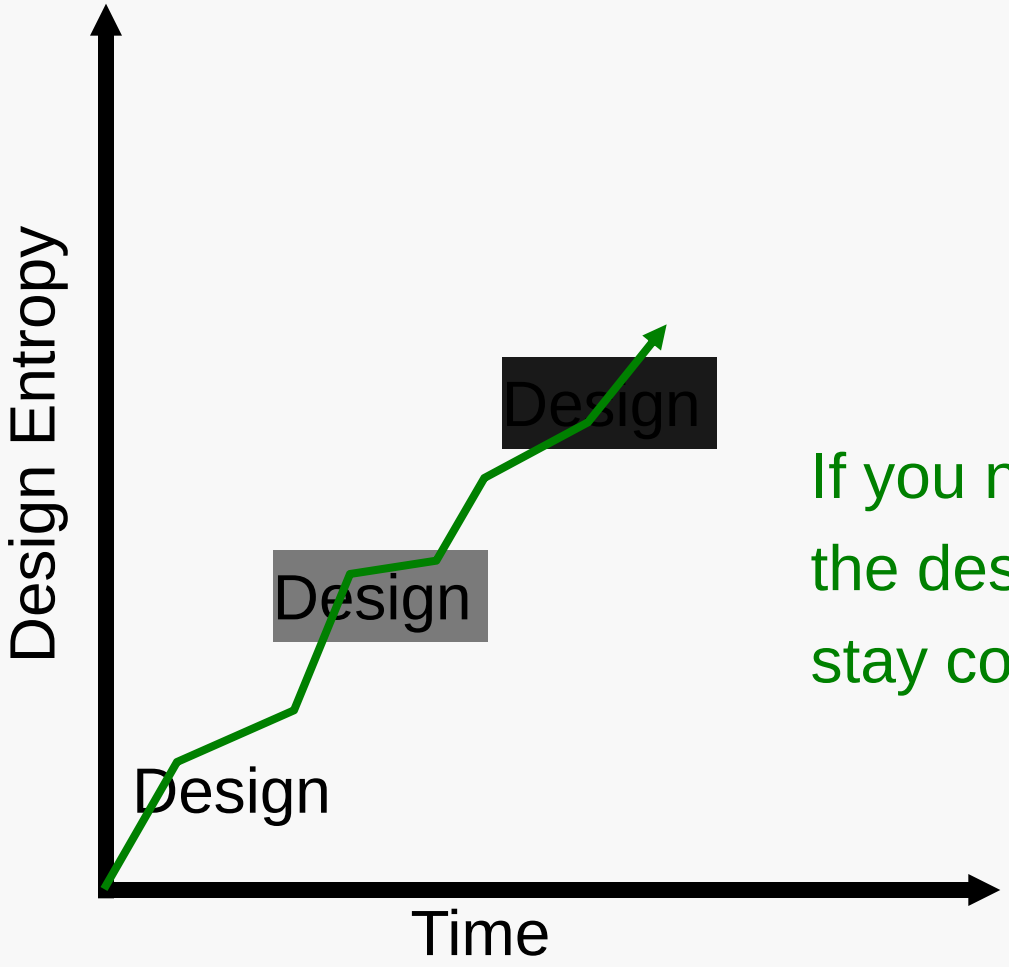
Martin Fowler, Addison-Wesley, 1999.



Refactoring should only change internal structure and not observable behavior.

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

The design entropy of a software system tends to increase over time.

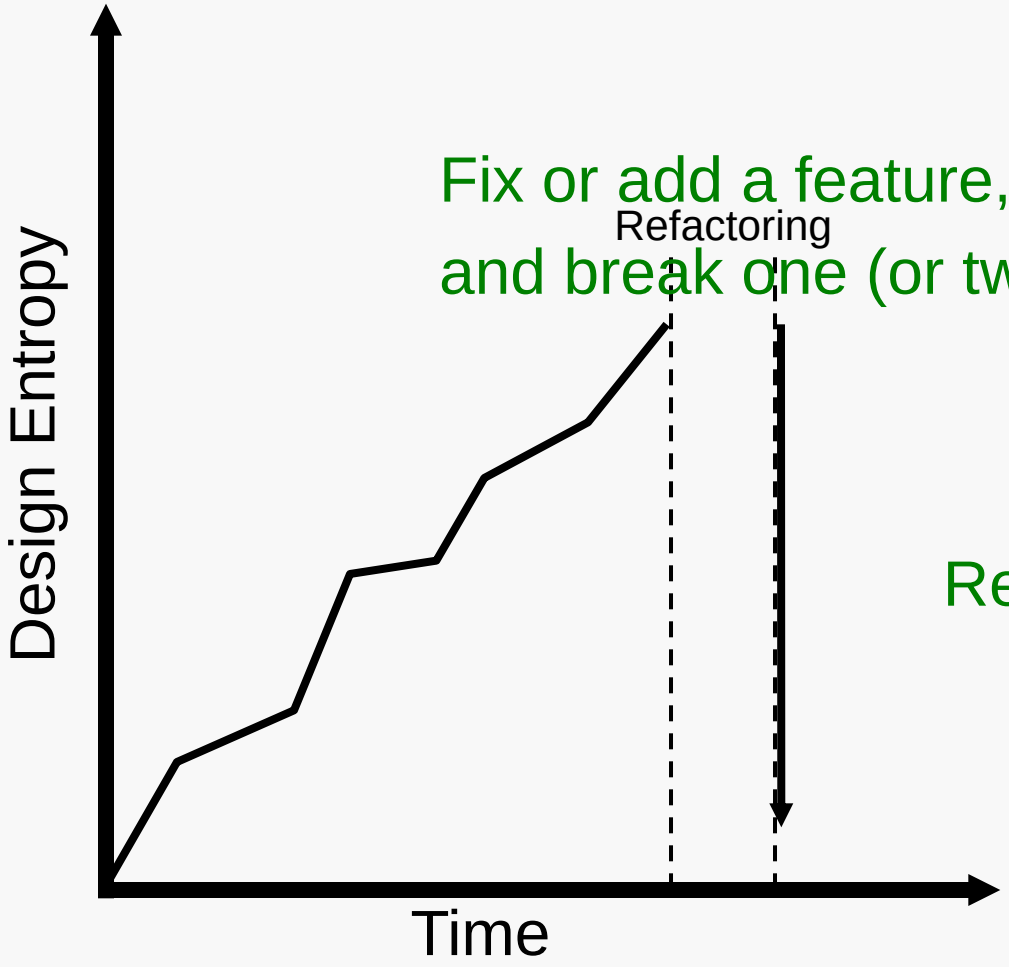


If you no longer can see the design, how can you stay consistent to it?

The entropy will increase because of the typical development death spiral.

- Good design up front
- Local modifications alter the framework
- Short-term goals win out over structure maintenance
- Engineering sinks into hacking
- Integrity and structure fade (entropy)

A refactoring activity can remove some of that design randomness.



Fix or add a feature,
and break one (or two)!

Refactor or Redo?

It is usually hard to counter, “If it ain’t broke, don’t fix it.”

- Generally improves product quality
- Pay today to ease work tomorrow.
- May actually accelerate today’s work

Ward Cunningham's Code Debt Metaphor

Shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite [refactor – ed.]. 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 debt load of an unconsolidated implementation, object-oriented or otherwise.

Refactoring does not work well as an end task because there never is any time to do it.

- Refactoring should be a continuous code improvement activity:
  - ***If it will make adding a new feature easier.***
  - ***If it will aid with debugging.***
  - ***If it fills a design hole.***
  - ***As a result of code inspection.***
  - ***If it simply makes the code easier to understand.***

- Do the right thing now before doing the right thing will take too much time…and be too risky!

While we are on the subject of doing the

Right Thing …

Code inspections have been found to be the most effective technique for early defect detection.

- Spreads design and implementation knowledge through team
- Helps mentor less experienced developers
- New eyes see things “old” eyes are not seeing
- Did your team do any code inspections on your implementation?
  - ***Next time when you can not find that bug, inspect don’t debug!***

Some complain that all this patterns stuff makes the code run slower.

- Refactored code *may* run slower
  - ***Do you notice?***
  - ***Do you care?***
- Ways to write fast code
  - ***Strict time budgets → hard real-time***
  - ***Constant attention → optimize always (!?)***
    - Write for speed – any and all “parlor” tricks
    - Obscures intentions
    - Harder to upgrade code later
    - Often does not help (80/20 rule)
  - ***Performance profiling – the intelligent engineer’s guide.***
- Make it work. Make it right. Make it fast.

If It Stinks, Change It.

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

- Duplicated code
- Long methods
- Large classes
- 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
- Inappropriate intimacy
- Data classes
- Verbose comments

What can we do with the type code?

- If type does not effect behavior of object but type is shared
  - ***Replace type code with class***
  - ***This allows type checking where data is shared***
- If type effects behavior of object
  - ***But never changes after instantiation***
    - Replace type code with subclasses
  - ***Is modified after instantiation***
    - Replace type code with state or strategy, as appropriate

```
if(type == TYPE_A) {
    code for TYPE_A …
}
else if(type == TYPE_B) {
    code for TYPE_B …
}
else if(type == TYPE_X) {
    code for TYPE_C …
}
else {
    code for unknown type …
}
```

Martin Fowler's book is a cookbook for getting rid of smells using common refactoring operations.

www.refactoring.com

- Extract method
- Inline method
- Replace temp with query
- Replace method with method object
- Substitute algorithm
- Extract class
- Hide delegate
- Remove middle man
- Replace type code with class
- Replace type code with state/strategy
- Replace type code with subclasses
- Introduce parameter object
- Replace inheritance with delegation
- **Plus 70 others**