Principles of Software Construction: Objects, Design, and Concurrency

Software engineering anti-patterns

Charlie Garrod **Chris Timperley**

Carnegie Mellon University
School of Computer Science

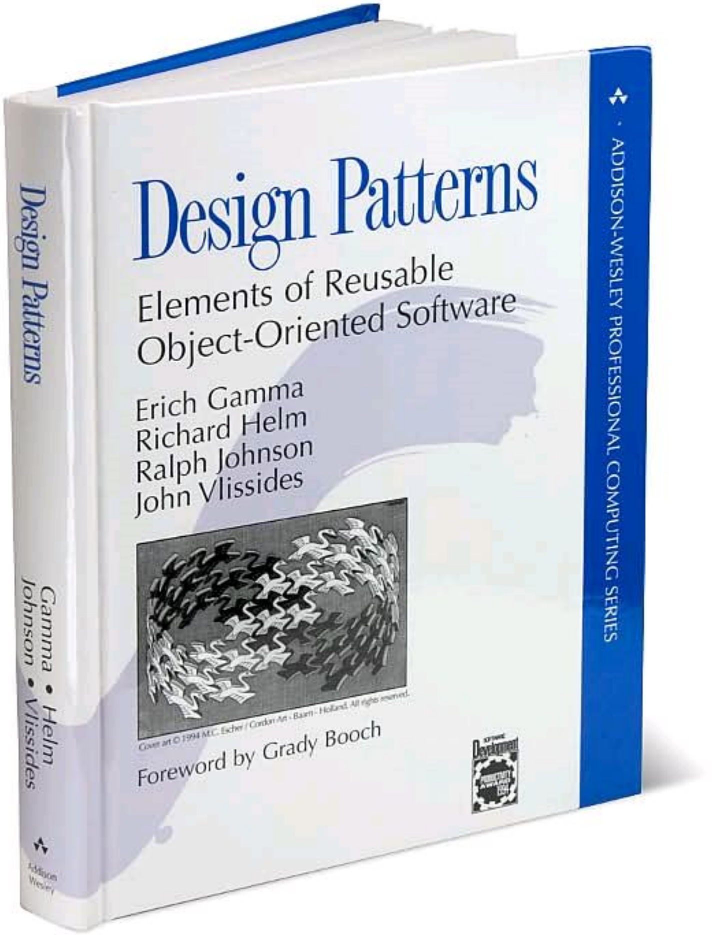
institute for SOFTWARE RESEARCH

Administrivia

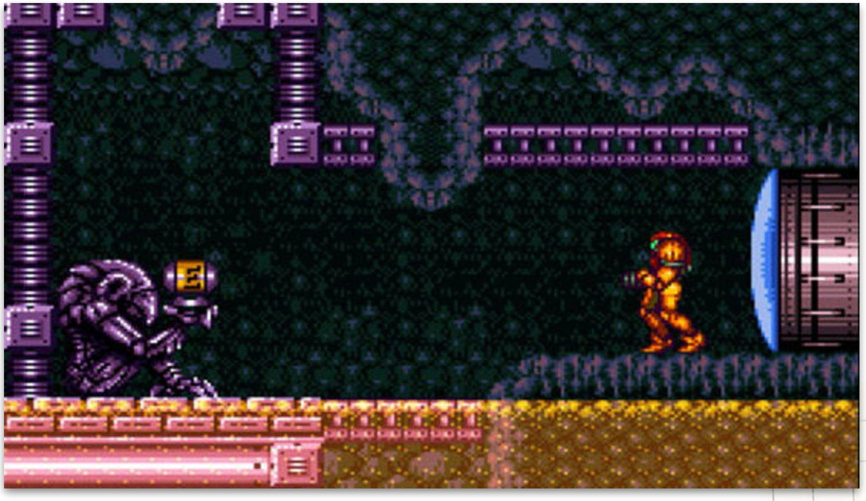
- Homework 6 due at end of Wednesday
- Final exam next Monday, 1–4 p.m. at GHC 4401 (Rashid)
  - Review session on Saturday, 12–2 p.m. at DH 1212
  - Additional office hours over the weekend (see calendar)

Last week: A tour of the "Gang of Four" patterns

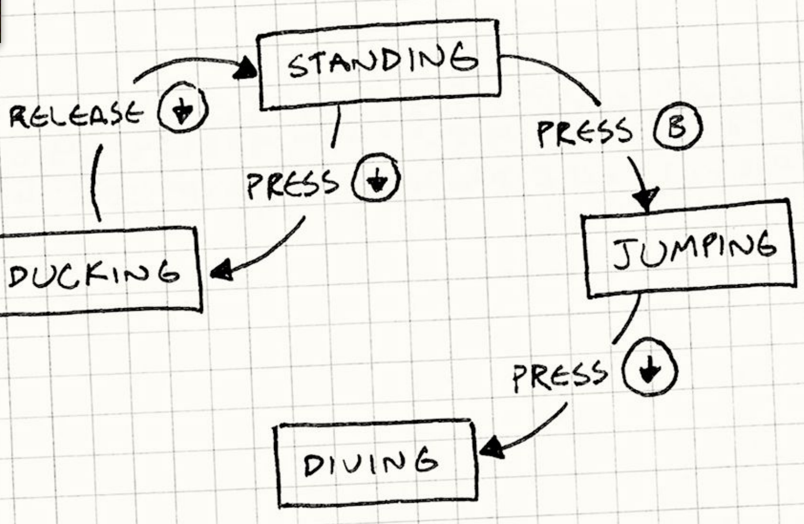
1. Creational Patterns
2. Structural Patterns
3. Behavioral Patterns



Problem: An object should behave differently based upon its internal state.

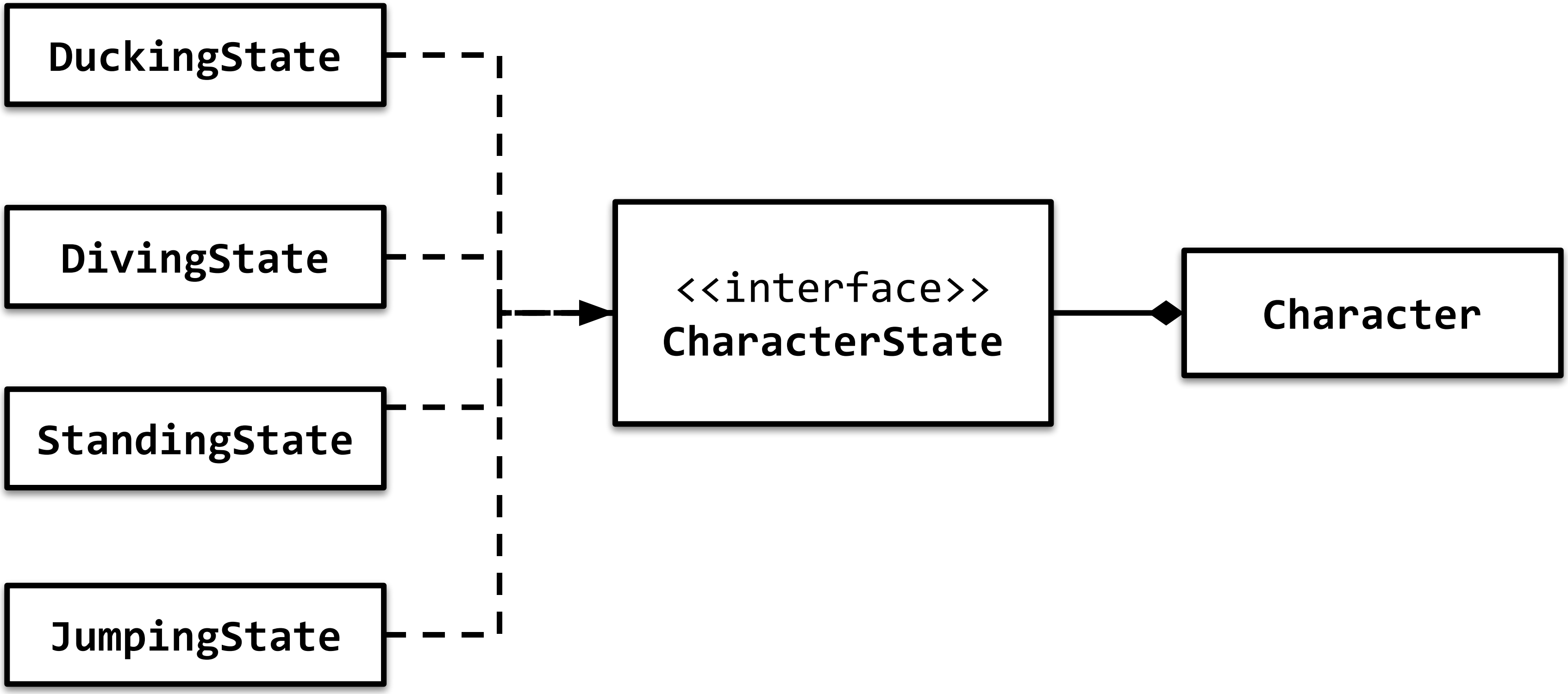


```
public class GameCharacter {
  …
  public void handleInput(Input input) {
    ...
  }
  …
}
```



https://gameprogrammingpatterns.com/state.html

Solution: Delegate behavior to a State object!



8. State

- Intent: allow an object to alter its behavior when internal state changes. “Object will appear to change class.”
- Use case: TCP Connection, Game AI
- Key type: *State* (Object delegates to state!)
- JDK: none that I’m aware of, but…
  - Works *great* in Java
  - Use enums as states
  - Use `AtomicReference<State>` to store it

Wrap-Up

- You now know *most* of the Gang of Four patterns
- Definitions can be vague
- Coverage is incomplete
- But they're extremely valuable
  - They gave us a vocabulary
  - And a way of thinking about software
- Look for patterns as you read and write software
  - GoF, non-GoF, and undiscovered

Today

- Software quality
- Technical debt
- Anti-patterns
- Code smells

Is it worth writing high-quality software?

OR

Writing and shipping new features.

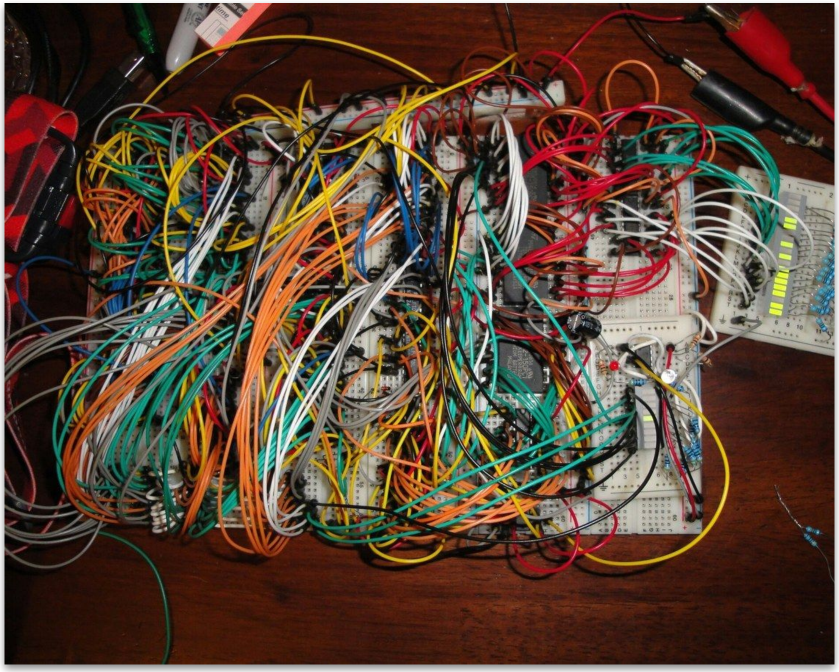
Polishing existing code and improving quality.

https://www.fedex.com/content/dam/fedex/us-united-states/FedEx-Office/images/2018/Q4/brown_boxes_stack_tile_1706294410.png
https://sierraclub.typepad.com/.a/6a00d83451b96069e20120a5b520b0970c-400wi

https://martinfowler.com/articles/is-quality-worth-cost.html

What is software quality?

Internal quality



- Is the code well structured?
- Is the code understandable?
- How well tested is the code?

External quality

- Does the software crash?
- Does the software meet its requirements?
- Is the UI well designed?

https://bugfender.com/wp-content/uploads/2018/06/01-App-crash.jpg
https://exceptionnotfound.net/content/images/2018/08/messy-circuit.jpg

Is it worth writing high-quality software?

OR

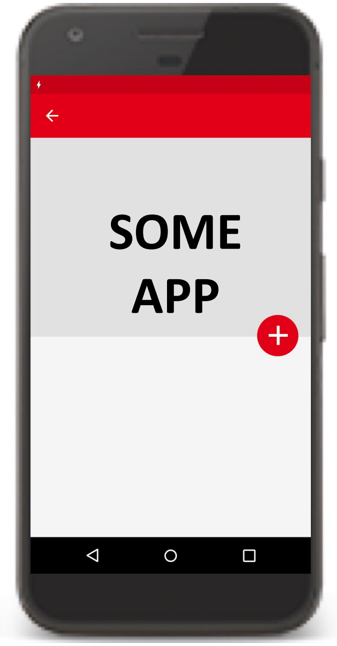
Writing and shipping new features.

Polishing existing code and improving **internal quality.**

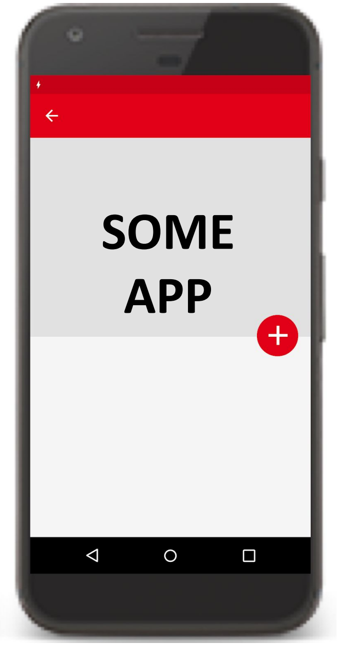
https://www.fedex.com/content/dam/fedex/us-united-states/FedEx-Office/images/2018/Q4/brown_boxes_stack_tile_1706294410.png
https://sierraclub.typepad.com/.a/6a00d83451b96069e20120a5b520b0970c-400wi

Which is better value to the customer?

Horrifying code



OR



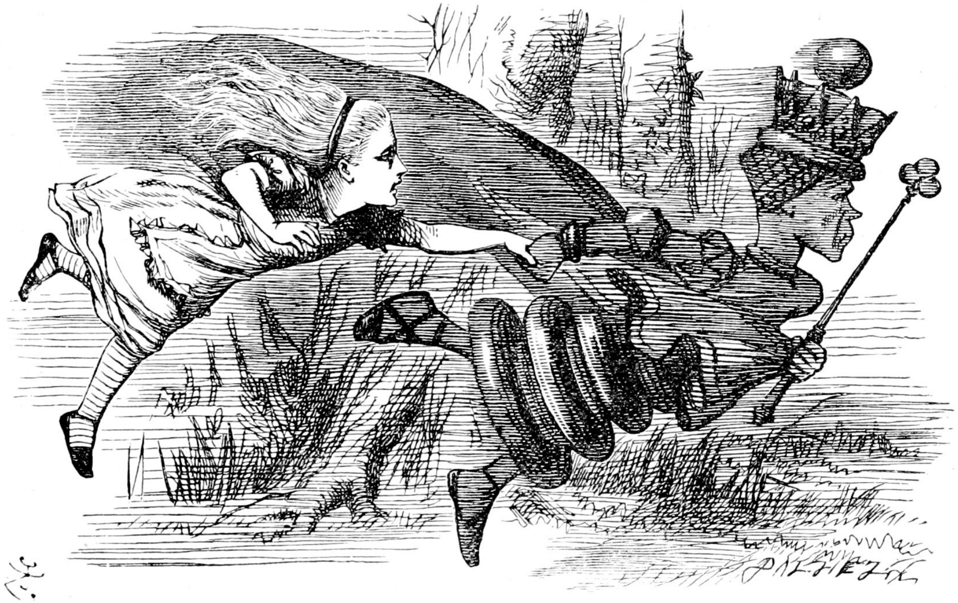
Beautiful code

$6

$10

Software entropy

“As an evolving program is continually changed, its complexity, reflecting deteriorating structure, increases unless work is done to maintain or reduce it”
Meir Manny Lehman



“Now, here, you see, it takes all the running you can do just to keep in the same place. If you want to get somewhere else, you must run at least twice as fast!”
Through the Looking Glass

https://phys.org/news/2013-06-red-queen.html

Aside: Software decay (a.k.a. “bit rot”)

Even if your software doesn’t change, it’s going to break over time due to changes in its environment.

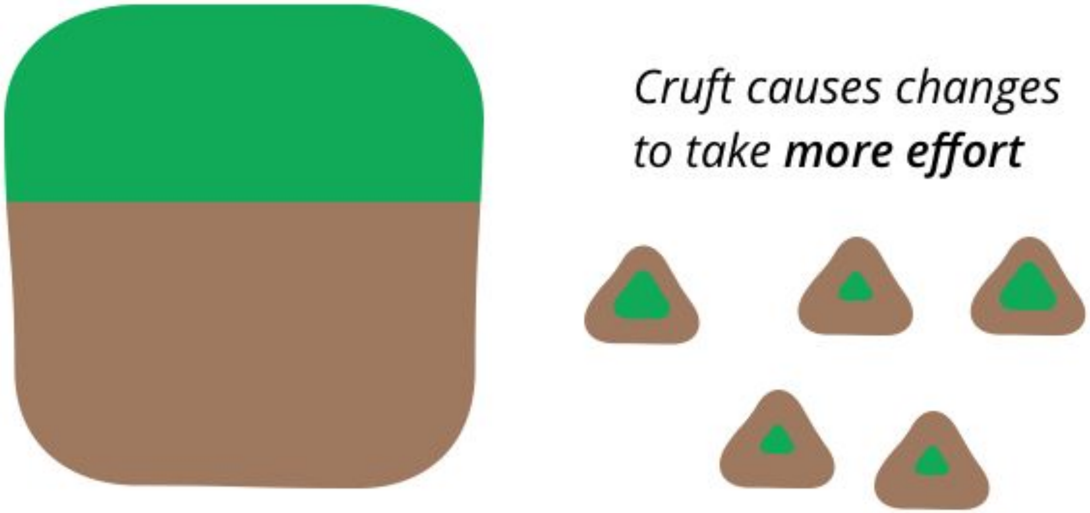
http://absfreepic.com/absolutely_free_photos/small_photos/old-cars-in-forest-4272x2848_75303.jpg

What’s happening here?

Technical debt

*Any software system has a certain amount of **essential** complexity required to do its job…*

*… but most systems contain **cruft** that makes it harder to understand.*

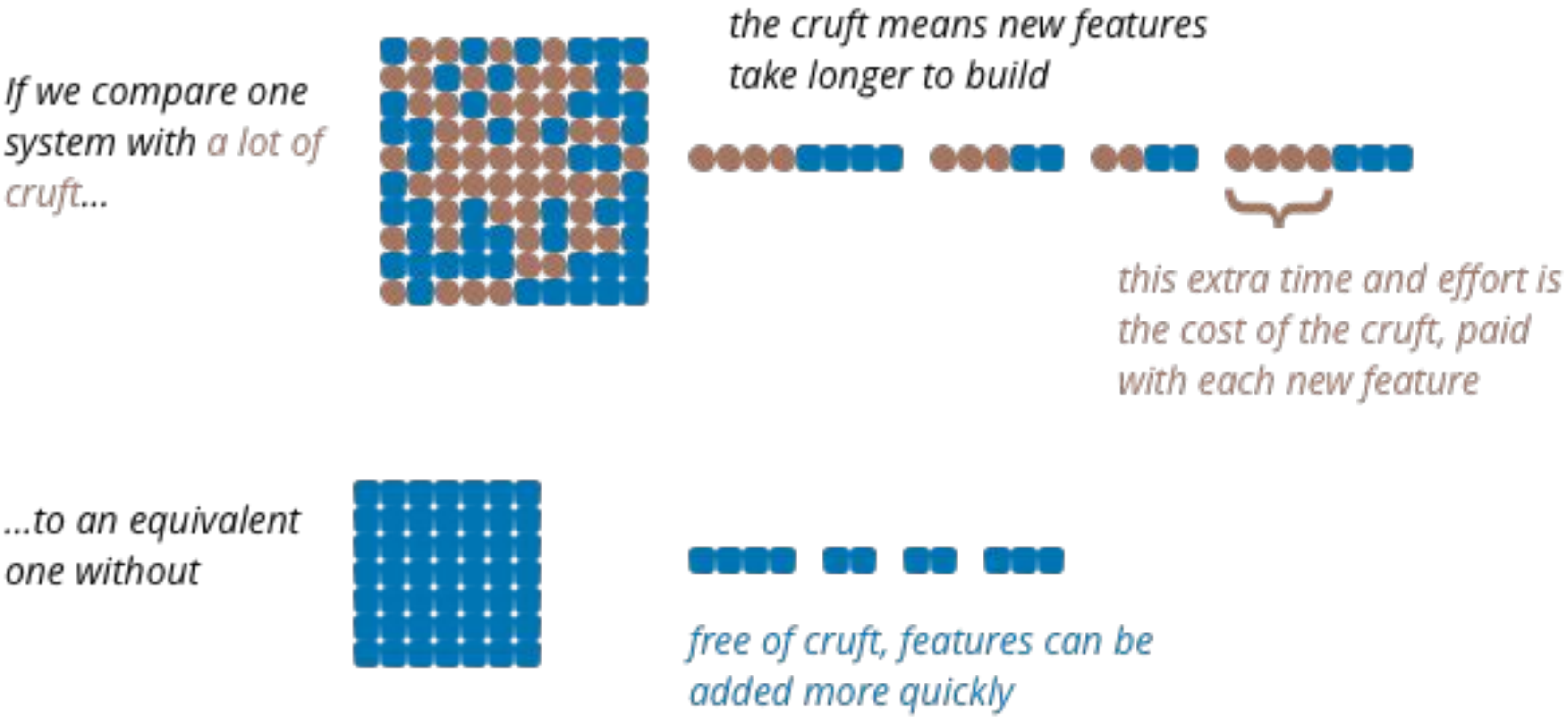


*Cruft causes changes to take **more effort***

The technical debt metaphor treats the cruft as a debt, whose interest payments are the extra effort these changes require.

https://martinfowler.com/bliki/TechnicalDebt.html

Internal quality makes it easier to add features



If we compare one system with a lot of cruft...

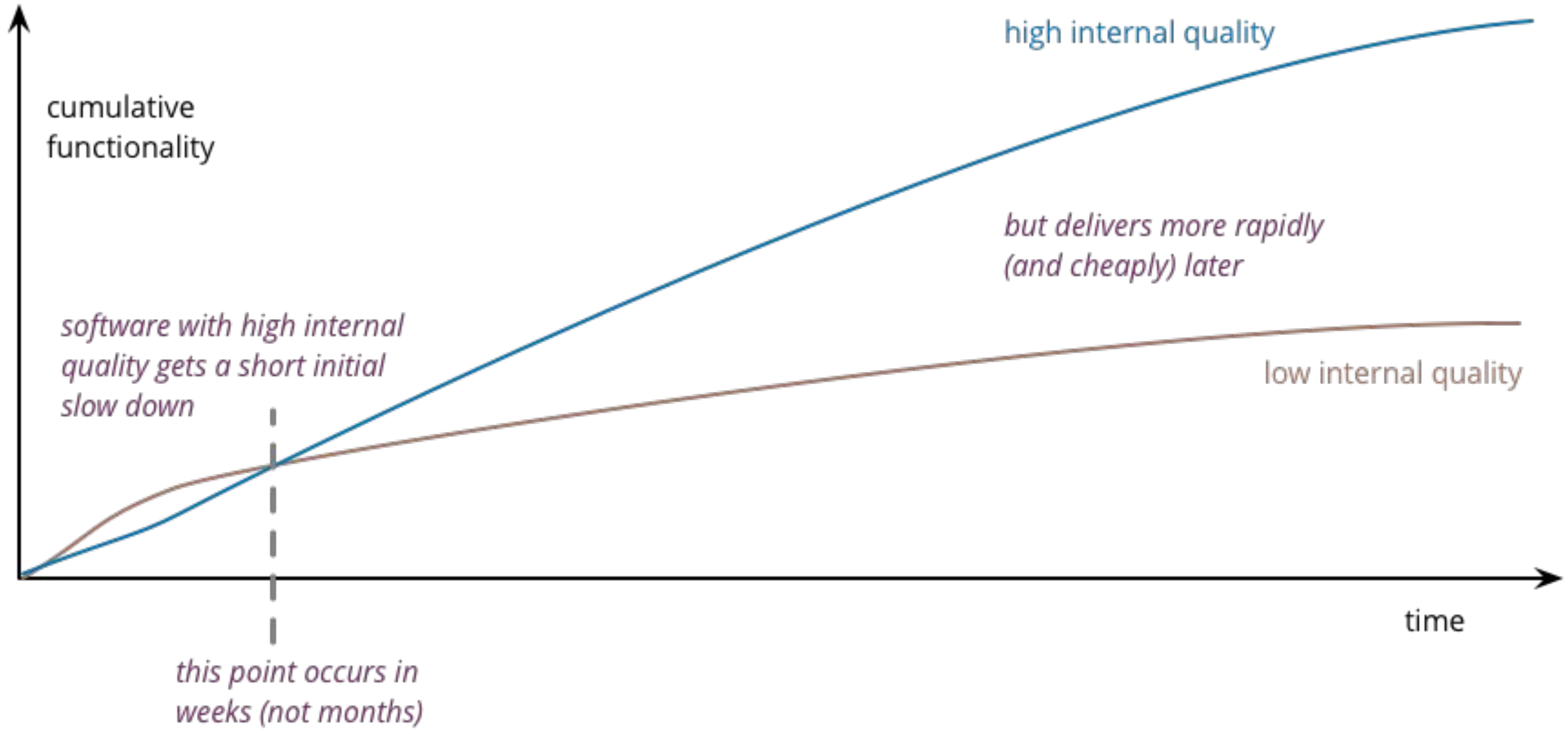
the cruft means new features take longer to build

this extra time and effort is the cost of the cruft, paid with each new feature

...to an equivalent one without

free of cruft, features can be added more quickly

High internal quality pays off over time



TL;DR: High-quality software is cheaper to produce

Today

- Software quality
- Technical debt
- Anti-patterns
- Code smells

What causes technical debt?

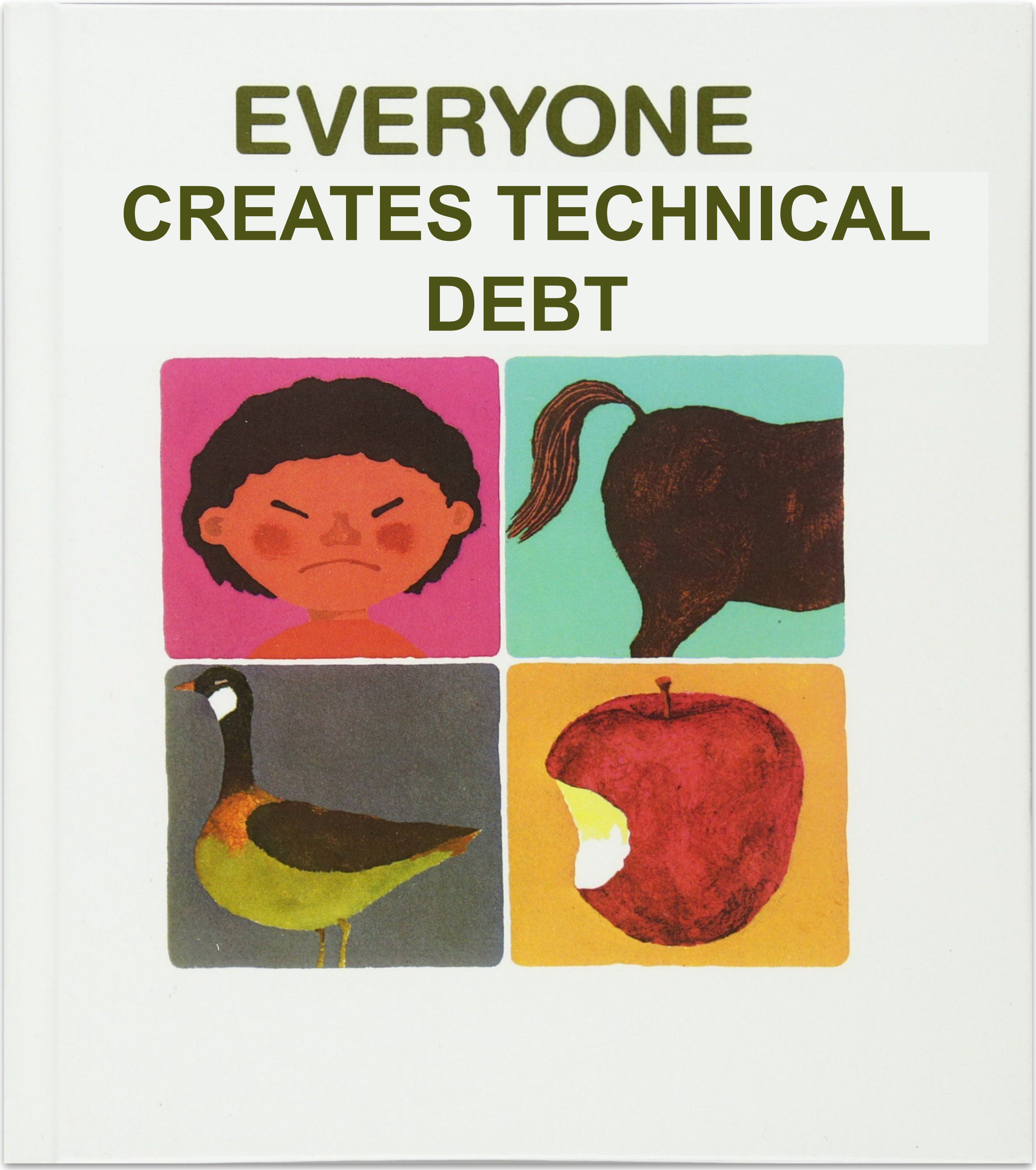
- Tightly-coupled components
- Poorly-specified requirements
- Business pressure
- Lack of process
- Lack of documentation
- Lack of a test suite
- Lack of knowledge
- Lack of ownership
- Delayed refactoring
- Multiple, long-lived development branches
- …

Types of Technical Debt

	Reckless	**Prudent**
Deliberate	*“We don’t have time for design”*	*“We must ship now and deal with consequences (later)”*
Inadvertent	*“What’s layering?”*	*“Now we know how we should have done it”*

https://martinfowler.com/bliki/TechnicalDebtQuadrant.html

EVERYONE CREATES TECHNICAL DEBT



https://www.amazon.com/Everyone-Turtleback-School-Library-Binding/dp/0613685725

Too much technical debt

- Bad code can be demoralising
- Conversations with the client become awkward
- Team infighting
- Atrophied skills
- Turnover and attrition

THIS IS FINE.
I'M OKAY WITH THE EVENTS THAT ARE UNFOLDING CURRENTLY.
THAT'S OKAY, THINGS ARE GOING TO BE OKAY.

https://www.theverge.com/2016/5/5/11592622/this-is-fine-meme-comic

https://daedtech.com/human-cost-tech-debt/

When should we reduce technical debt?

Dealing with technical debt: Fixing broken windows

https://phys.org/news/2019-05-evidence-broken-windows-theory-neighborhood.html

Alternative: Putting out fires is expensive!

https://internetofbusiness.com/how-fog-computing-is-enabling-smart-firefighting

Analogy: Cleaning your dryer

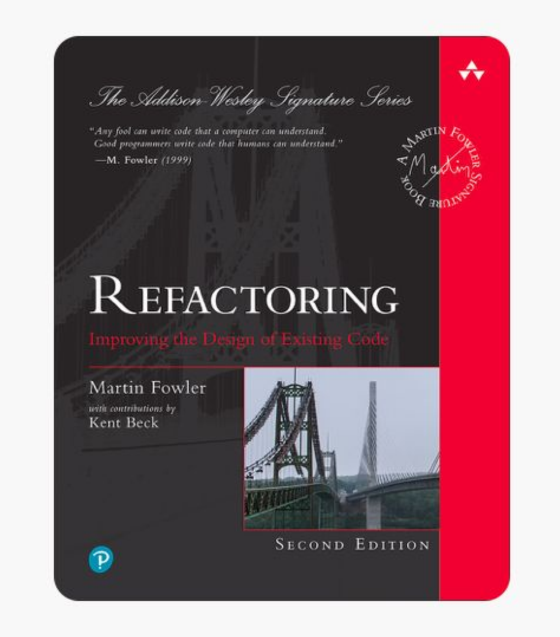
https://www.squeegeepros.com/files/71AC337A-6FC3-46A6-A68F-12551AD60EA9--E1F09494-816E-48D5-A812-7A327D17098F/dryer-lint-dryer-fire.jpg?nc=05232019092309

How should we reduce technical debt?

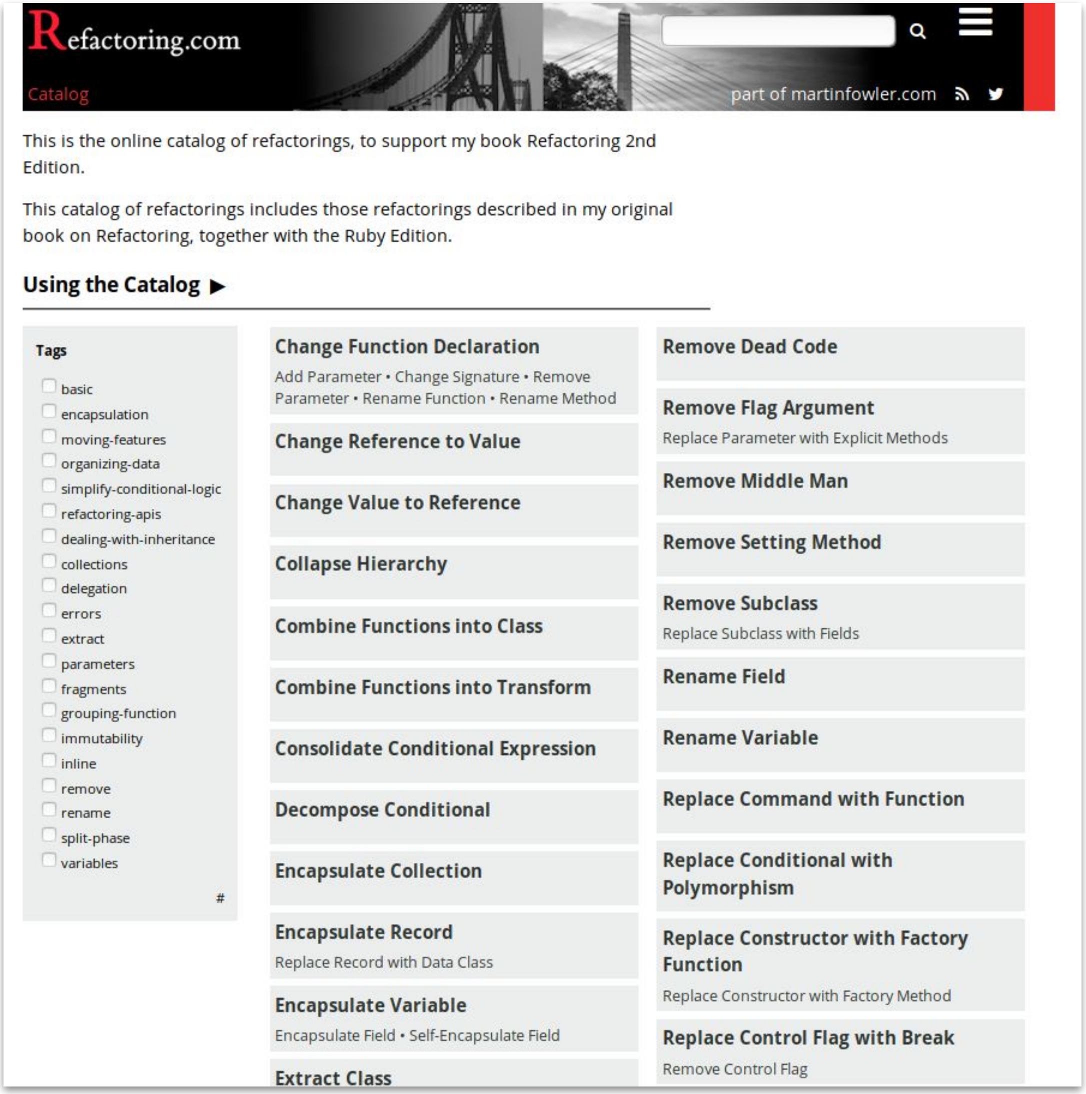
Refactoring

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.”

Refactoring (verb): “to restructure software by applying **a series of refactorings without changing its observable behavior.”**


The Addison-Wesley Signature Series
“Any fool can write code that a computer can understand. Good programmers write code that humans can understand.”
—M. Fowler (1999)
A Martin Fowler Signature Book
Martin
REFACTORING
Improving the Design of Existing Code
Martin Fowler
with contributions by
Kent Beck
SECOND EDITION

Refactorings



This is the online catalog of refactorings, to support my book Refactoring 2nd Edition.

This catalog of refactorings includes those refactorings described in my original book on Refactoring, together with the Ruby Edition.

Using the Catalog ▶

Tags

- □ basic
- □ encapsulation
- □ moving-features
- □ organizing-data
- □ simplify-conditional-logic
- □ refactoring-apis
- □ dealing-with-inheritance
- □ collections
- □ delegation
- □ errors
- □ extract
- □ parameters
- □ fragments
- □ grouping-function
- □ immutability
- □ inline
- □ remove
- □ rename
- □ split-phase
- □ variables

#

Change Function Declaration
Add Parameter • Change Signature • Remove Parameter • Rename Function • Rename Method

Change Reference to Value

Change Value to Reference

Collapse Hierarchy

Combine Functions into Class

Combine Functions into Transform

Consolidate Conditional Expression

Decompose Conditional

Encapsulate Collection

Encapsulate Record
Replace Record with Data Class

Encapsulate Variable
Encapsulate Field • Self-Encapsulate Field

Extract Class

Remove Dead Code

Remove Flag Argument
Replace Parameter with Explicit Methods

Remove Middle Man

Remove Setting Method

Remove Subclass
Replace Subclass with Fields

Rename Field

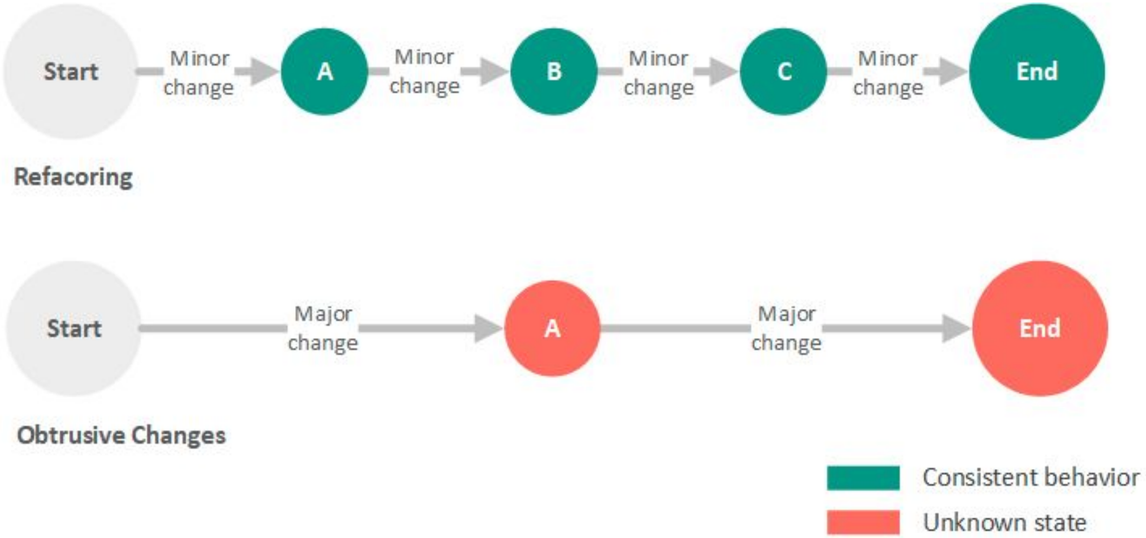
Rename Variable

Replace Command with Function

Replace Conditional with Polymorphism

Replace Constructor with Factory Function
Replace Constructor with Factory Method

Replace Control Flag with Break
Remove Control Flag



Start → Minor change → A → Minor change → B → Minor change → C → Minor change → End

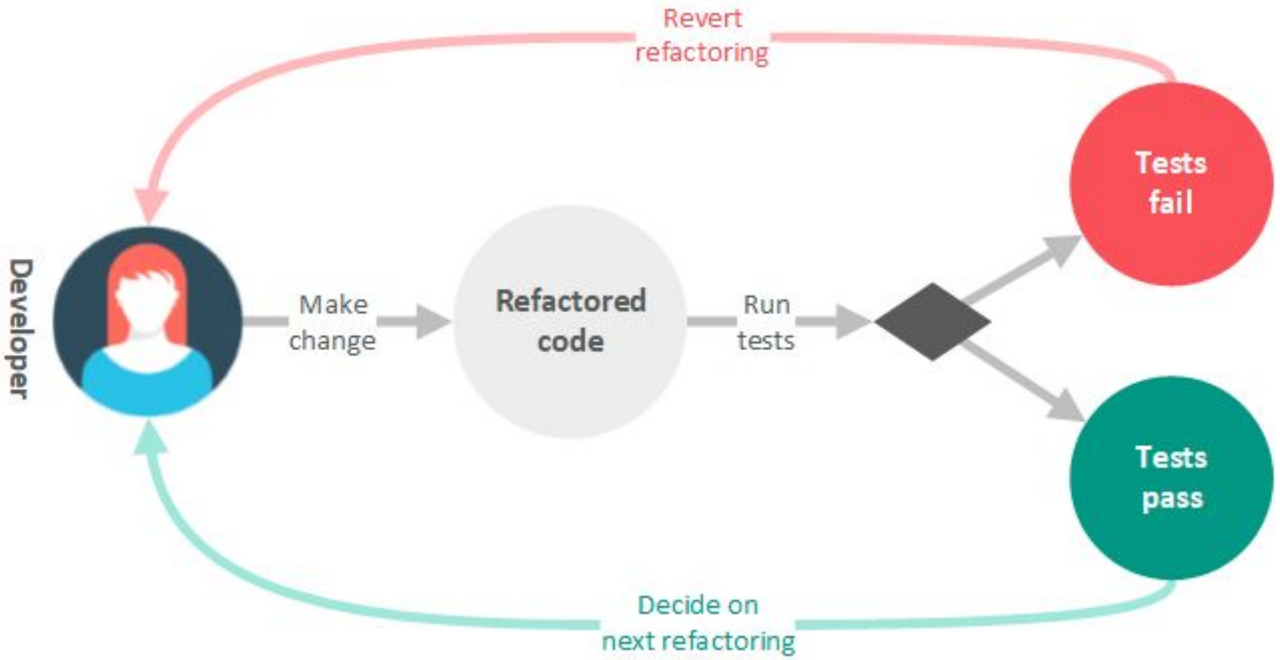
Refacoring

Start → Major change → A → Major change → End

Obtrusive Changes

🟩 Consistent behavior

🟥 Unknown state



Developer

Make change

Refactored code

Run tests

Tests fail

Tests pass

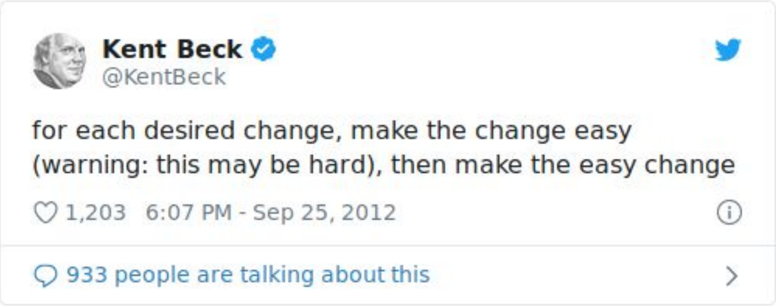
Revert refactoring

Decide on next refactoring

https://dzone.com/articles/what-is-refactoring

When should we refactor?

- TDD Refactoring
- Litter-Pickup Refactoring
- Comprehension Refactoring
- Preparatory Refactoring
- Planned Refactoring
- Long-Term Refactoring



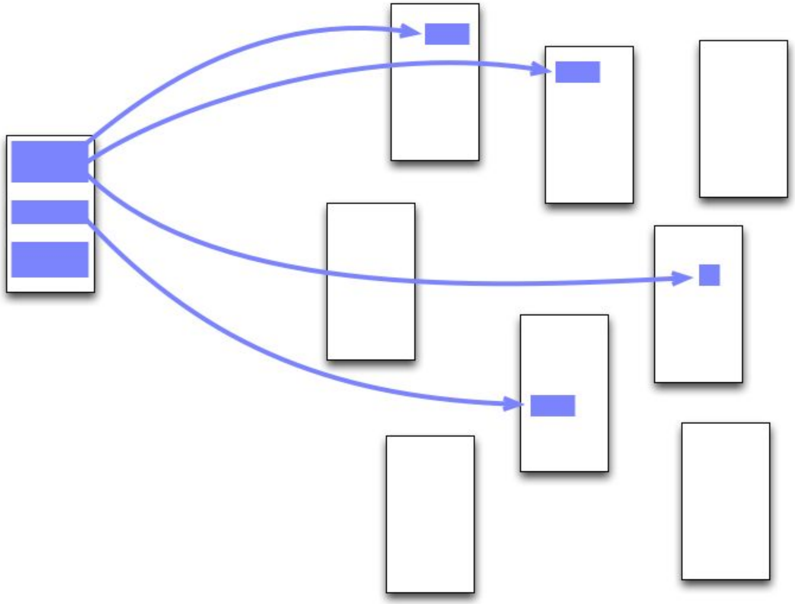
Kent Beck
@KentBeck

for each desired change, make the change easy (warning: this may be hard), then make the easy change

1,203 6:07 PM - Sep 25, 2012

933 people are talking about this

Preparatory Refactoring



Opportunistic Refactoring

https://martinfowler.com/articles/preparatory-refactoring-example.html
https://martinfowler.com/bliki/OpportunisticRefactoring.html

Today

- Software quality
- Technical debt
- Anti-patterns
- Code smells

Anti-patterns

- “Anti”-pattern
- Describe things that you should **AVOID**
  - Anti-patterns cover *programming, design,* and *process*
- Often have memorable names

> “Some repeated pattern of action, process or structure that initially appears to be beneficial, but **ultimately produces more bad consequences than beneficial results**, …”
>
> *Anti Patterns: refactoring software, architectures, and projects in crisis*

https://pbs.twimg.com/profile_images/632821853627678720/zPKK7jql_400x400.png

http://wiki3.cosc.canterbury.ac.nz/index.php/Antipatterns

There are lots of anti-patterns! Here’s a few...

Analysis paralysis
Cash cow
Design by committee
Escalation of commitment
Management by perkele
Matrix Management
Moral hazard
Mushroom management
Silos
Vendor lock-in
Death march
Groupthink
Smoke and mirrors
Software bloat
Waterfall model
Bystander apathy
Abstraction inversion
Ambiguous viewpoint
Big ball of mud
Database-as-IPC
Gold plating
Inner-platform effect
Input kludge
Interface bloat

Accidental complexity
Action at a distance
Blind faith
Boat anchor
Busy spin
Caching failure
Cargo cult programming
Coding by exception
Error hiding
Hard code
Lava flow
Loop-switch sequence
Magic numbers
Magic strings
Soft code
Spaghetti code
Copy and paste programming
Golden hammer
Improbability factor
Not Invented Here (NIH) syndrome
Premature optimization
Programming by permutation
Reinventing the wheel
Reinventing the square wheel

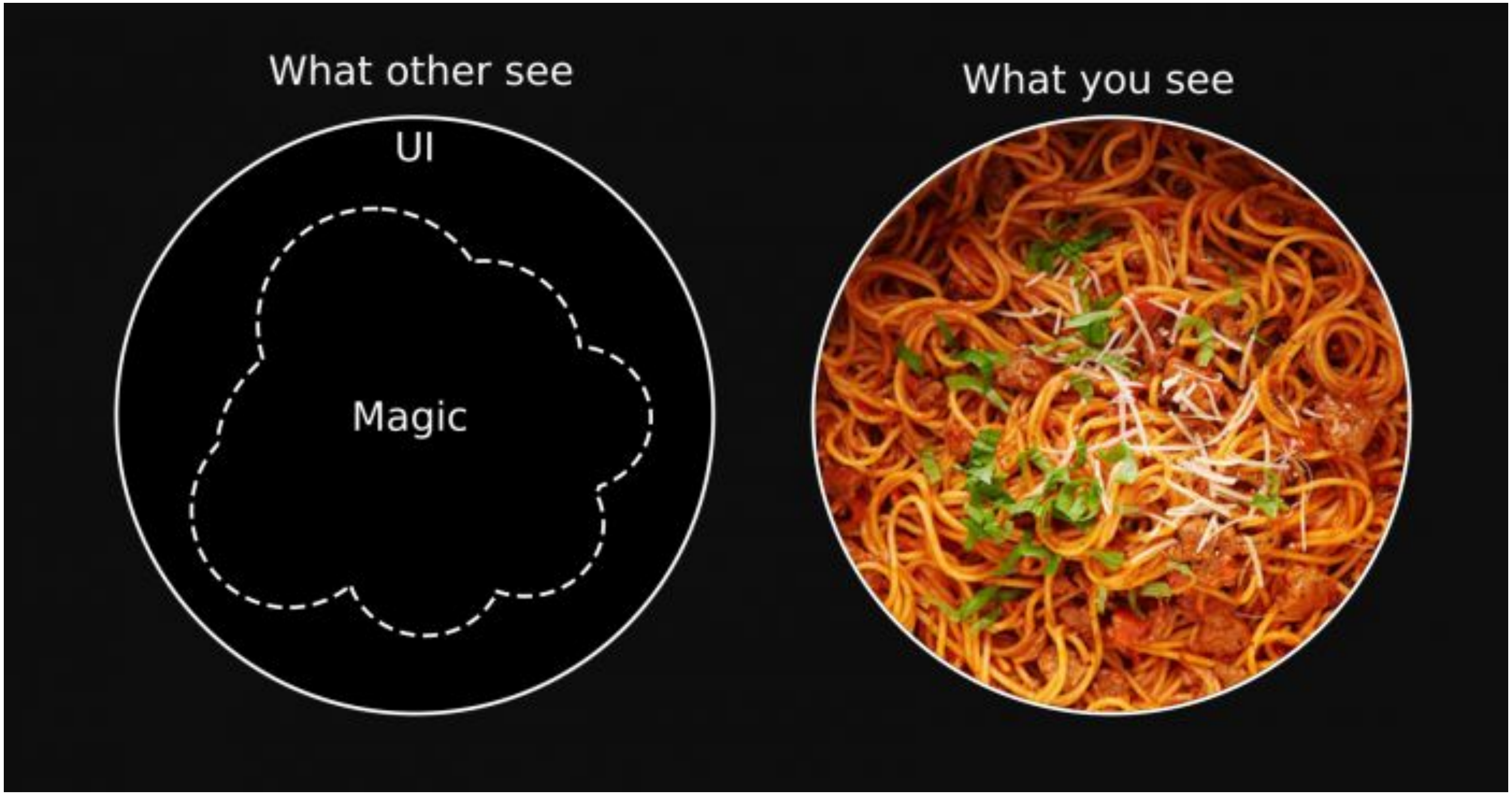
Extension conflict
JAR hell
BaseBean
Call super
Circle-ellipse problem
Circular dependency
Constant interface
God object
Object cesspool
Object orgy
Poltergeists
Sequential coupling
Yo-yo problem
Hurry up and wait
Magic pushbutton
Race hazard
Stovepipe system
Anemic Domain Model
Silver bullet
Tester Driven Development
Dependency hell
DLL hell
...

https://en.wikibooks.org/wiki/Introduction_to_Software_Engineering/Architecture/Anti-Patterns

Anti-patterns

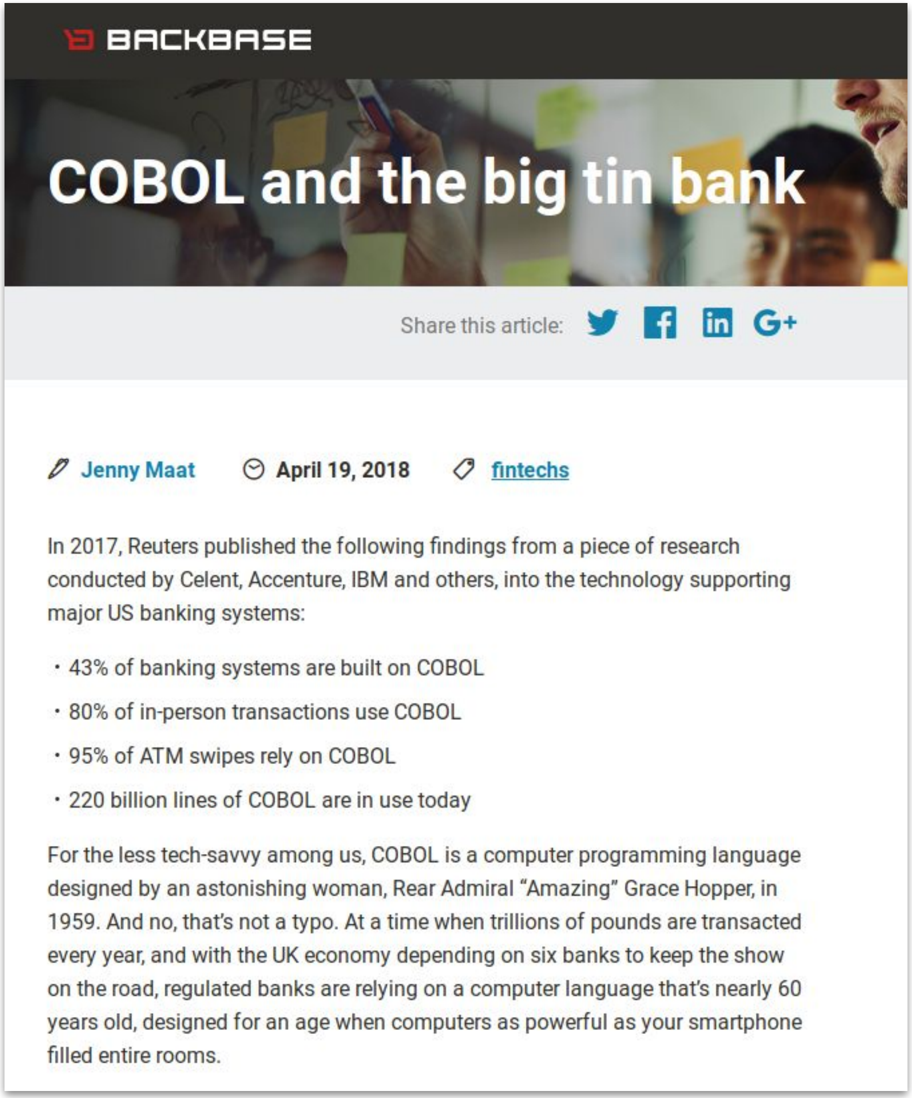
1. **Programming anti-patterns**
2. Design anti-patterns
3. Process anti-patterns

Spaghetti Code



https://i.pinimg.com/originals/c7/69/04/c76904f05d92f2b45a3bccc45a3998f2.png

Lava Flow



COBOL and the big tin bank

Jenny Maat April 19, 2018 fintechs

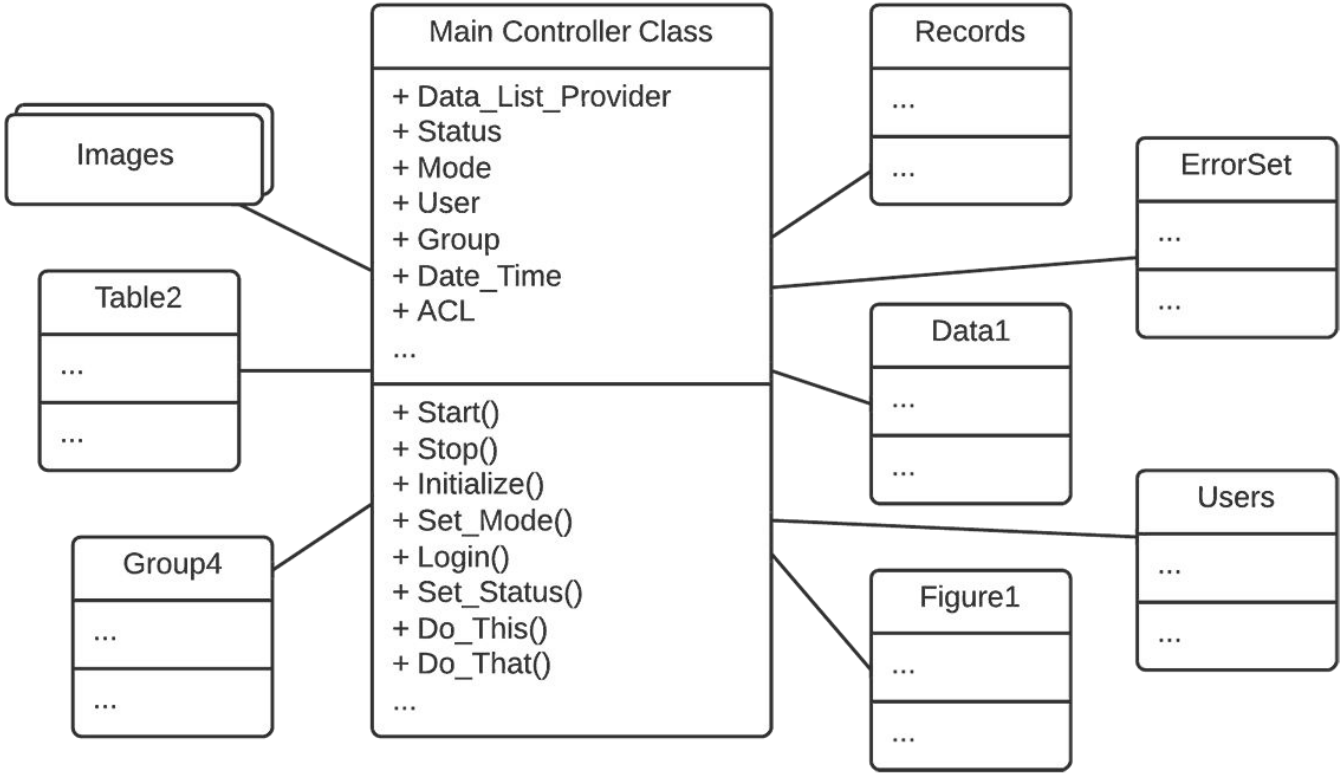
In 2017, Reuters published the following findings from a piece of research conducted by Celent, Accenture, IBM and others, into the technology supporting major US banking systems:

- 43% of banking systems are built on COBOL
- 80% of in-person transactions use COBOL
- 95% of ATM swipes rely on COBOL
- 220 billion lines of COBOL are in use today

For the less tech-savvy among us, COBOL is a computer programming language designed by an astonishing woman, Rear Admiral "Amazing" Grace Hopper, in 1959. And no, that's not a typo. At a time when trillions of pounds are transacted every year, and with the UK economy depending on six banks to keep the show on the road, regulated banks are relying on a computer language that's nearly 60 years old, designed for an age when computers as powerful as your smartphone filled entire rooms.

https://image.shutterstock.com/image-photo/wood-blocks-stack-game-copyspace-260nw-1076163038.jpg

The Blob



Main Controller Class

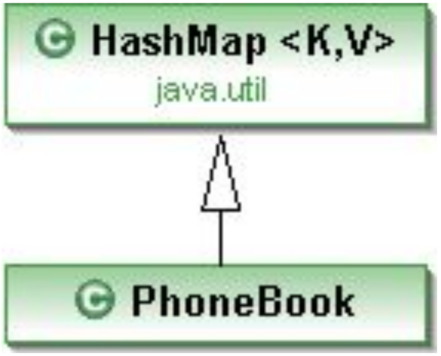
+ Data_List_Provider
+ Status
+ Mode
+ User
+ Group
+ Date_Time
+ ACL
...

+ Start()
+ Stop()
+ Initialize()
+ Set_Mode()
+ Login()
+ Set_Status()
+ Do_This()
+ Do_That()
...

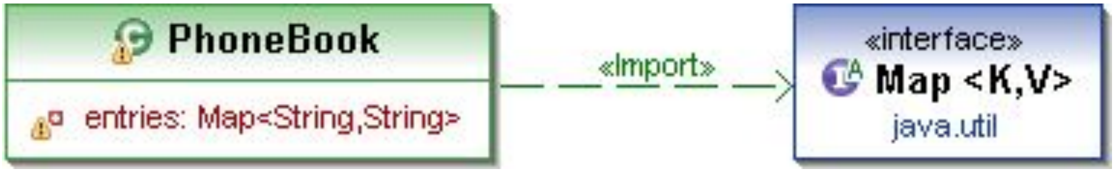
Anti-patterns

1. Programming anti-patterns
2. **Design anti-patterns**
3. Process anti-patterns

BaseBean



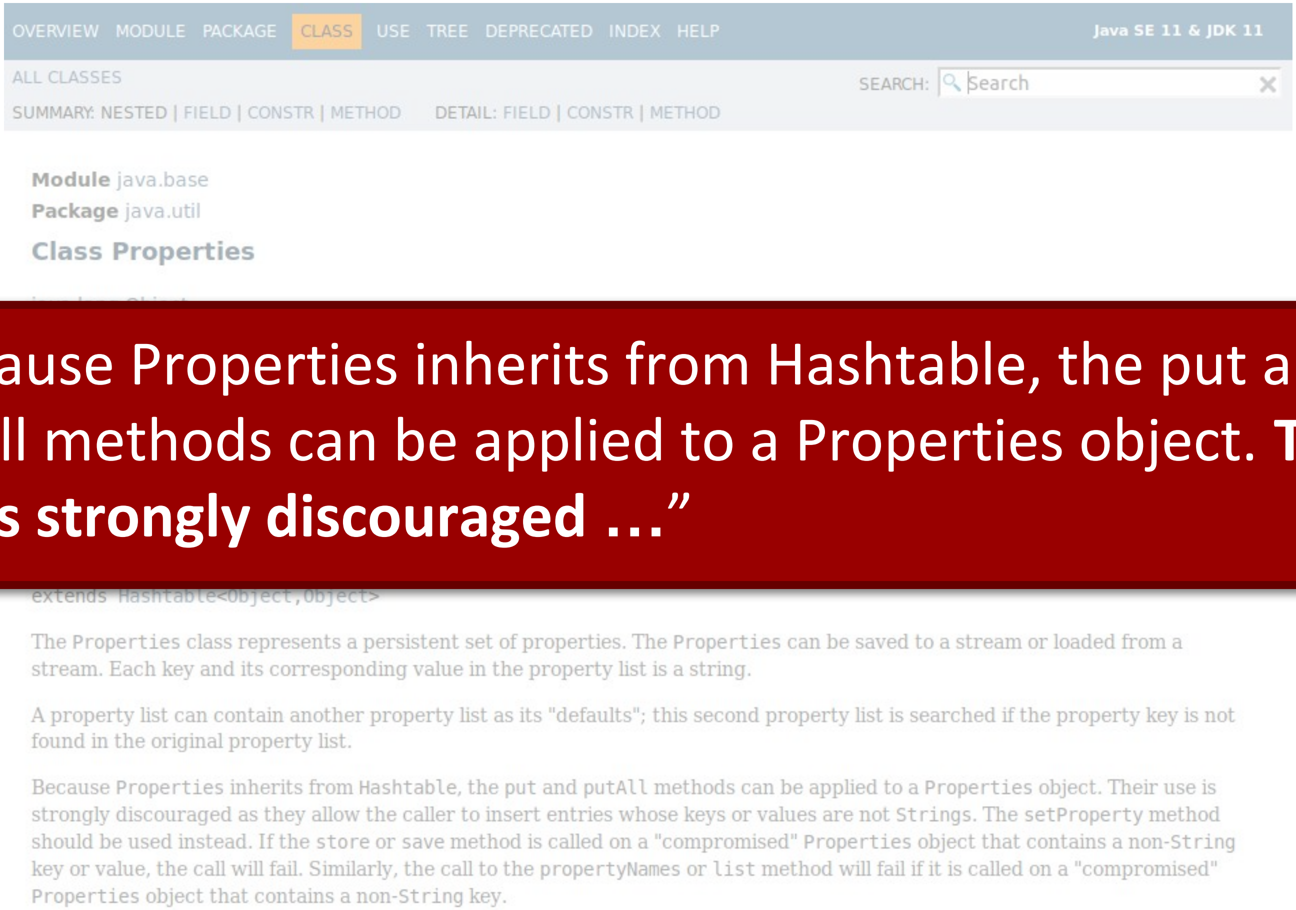
BAD



GOOD

http://wiki3.cosc.canterbury.ac.nz/index.php/BaseBean

```
public class Properties extends Hashtable<Object, Object> {
```



OVERVIEW MODULE PACKAGE CLASS USE TREE DEPRECATED INDEX HELP
Java SE 11 & JDK 11
ALL CLASSES
SEARCH: Search
SUMMARY: NESTED | FIELD | CONSTR | METHOD
DETAIL: FIELD | CONSTR | METHOD

Module java.base
Package java.util

Class Properties

`extends Hashtable<Object,Object>`

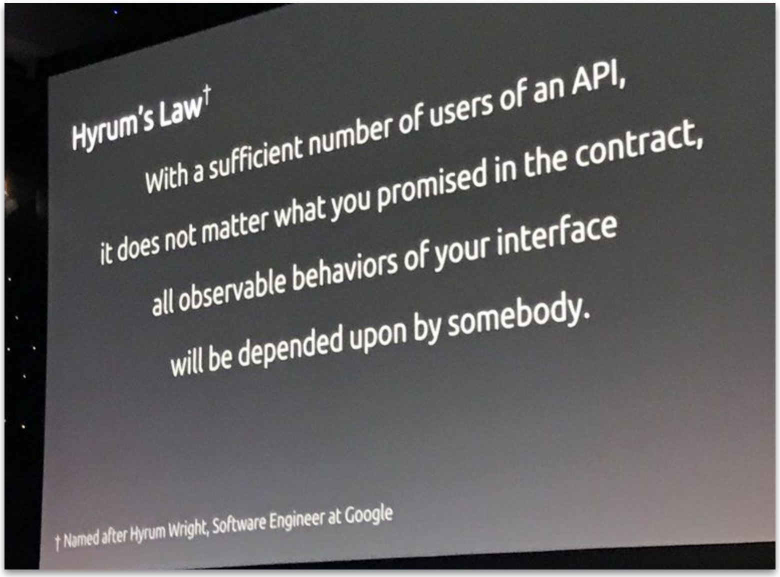
The Properties class represents a persistent set of properties. The Properties can be saved to a stream or loaded from a stream. Each key and its corresponding value in the property list is a string.

A property list can contain another property list as its "defaults"; this second property list is searched if the property key is not found in the original property list.

Because Properties inherits from Hashtable, the put and putAll methods can be applied to a Properties object. Their use is strongly discouraged as they allow the caller to insert entries whose keys or values are not Strings. The setProperty method should be used instead. If the store or save method is called on a "compromised" Properties object that contains a non-String key or value, the call will fail. Similarly, the call to the propertyNames or list method will fail if it is called on a "compromised" Properties object that contains a non-String key.

"Because Properties inherits from Hashtable, the put and putAll methods can be applied to a Properties object. **Their use is strongly discouraged …**"

Swiss-Army Knife



Hyrum's Law[†]

With a sufficient number of users of an API,
it does not matter what you promised in the contract,
all observable behaviors of your interface
will be depended upon by somebody.

† Named after Hyrum Wright, Software Engineer at Google

http://codebalance.blogspot.com/2011/08/software-architecture-antipatterns.html

Call Super

```
public class EventHandler {
  …
  public void handle(BankEvent event) {
    housekeeping(event);
  }
}

public class TransferEventHandler extends EventHandler {
  …
  public void handle(BankingEvent event) {
    super.handle(event);
    initiateTransfer(e);
  }
}
```

Danger: Easy to forget to call super!

https://martinfowler.com/bliki/CallSuper.html

Call Super

```
public class EventHandler {
  …
  public void handle(BankEvent event) {
    housekeeping(event);
    doHandle(event);
  }
  protected void doHandle(BankEvent event) { }
}

public class TransferEventHandler extends EventHandler {
  protected void doHandle(BankingEvent event) {
    initiateTransfer(e);
  }
}
```

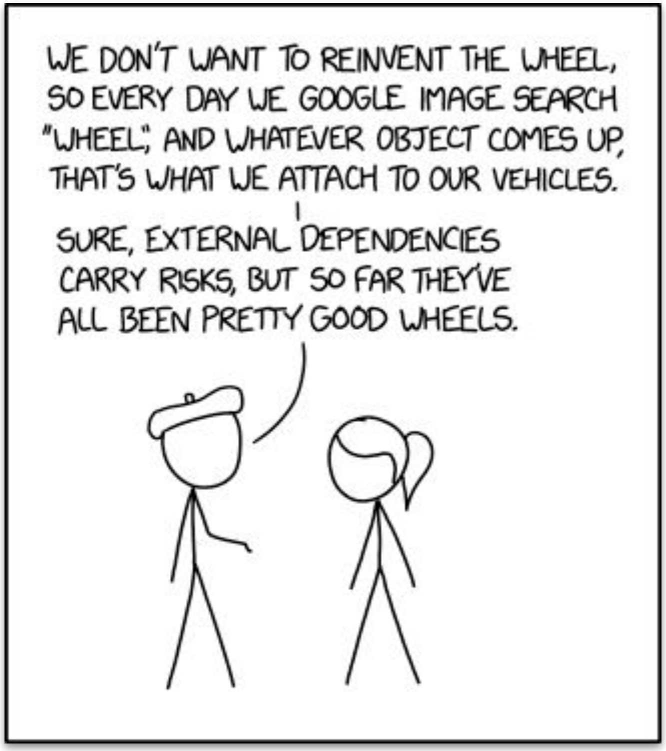
Solution: Use the template method pattern instead.

https://martinfowler.com/bliki/CallSuper.html

Anti-patterns

1. Programming anti-patterns
2. Design anti-patterns
3. **Process anti-patterns**

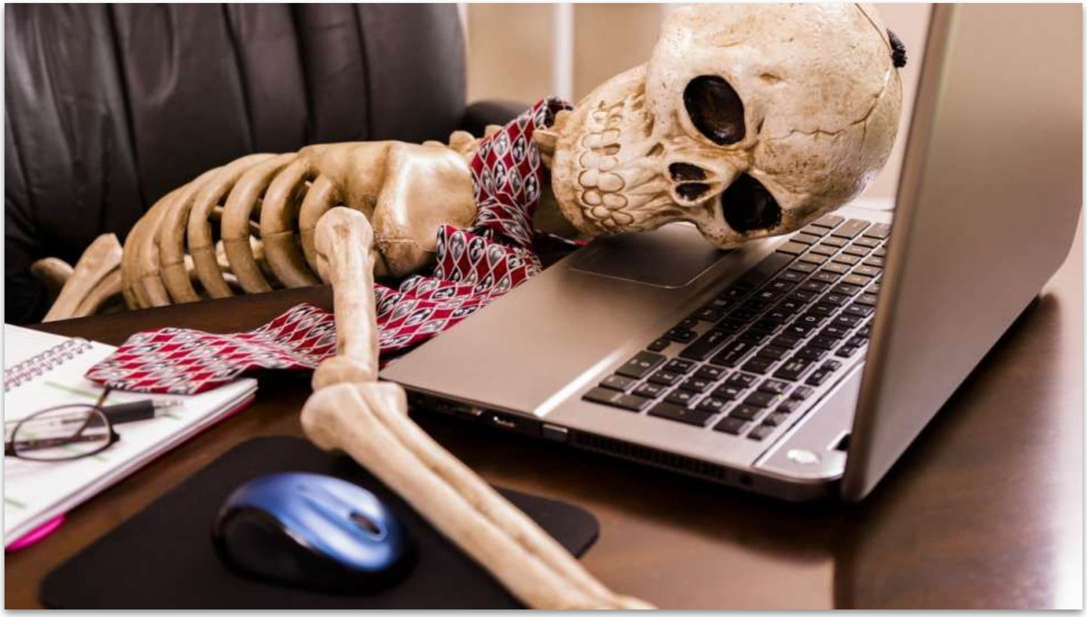
Reinventing the wheel

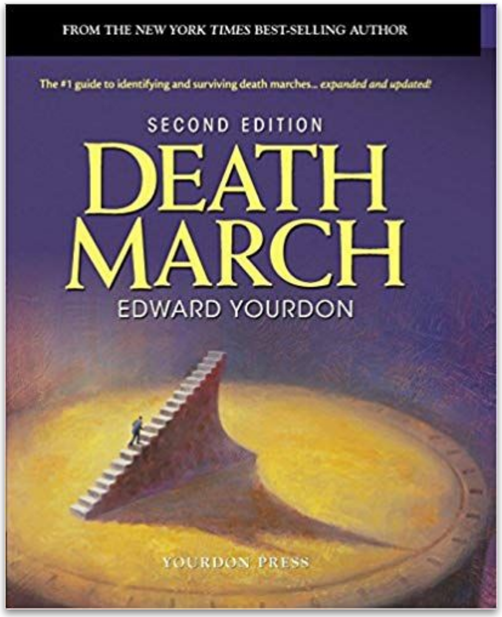

WE DON'T WANT TO REINVENT THE WHEEL, SO EVERY DAY WE GOOGLE IMAGE SEARCH "WHEEL," AND WHATEVER OBJECT COMES UP, THAT'S WHAT WE ATTACH TO OUR VEHICLES.
SURE, EXTERNAL DEPENDENCIES CARRY RISKS, BUT SO FAR THEY'VE ALL BEEN PRETTY GOOD WHEELS.

https://imgs.xkcd.com/comics/reinvent_the_wheel.png

https://exceptionnotfound.net/reinventing-the-square-wheel-the-daily-software-anti-pattern/

Death March




FROM THE NEW YORK TIMES BEST-SELLING AUTHOR
The #1 guide to identifying and surviving death marches... expanded and updated!
SECOND EDITION
DEATH MARCH
EDWARD YOURDON
YOURDON PRESS

Golden Hammer

https://images-na.ssl-images-amazon.com/images/I/81Qq22mGSYL._SL1500_.jpg

https://exceptionnotfound.net/the-golden-hammer-anti-pattern-primers/

Cargo Cult Programming

Including code in a system without understanding why that code needs to be included.

https://www.apexbpm.com/blog/cargo-cult-programming/

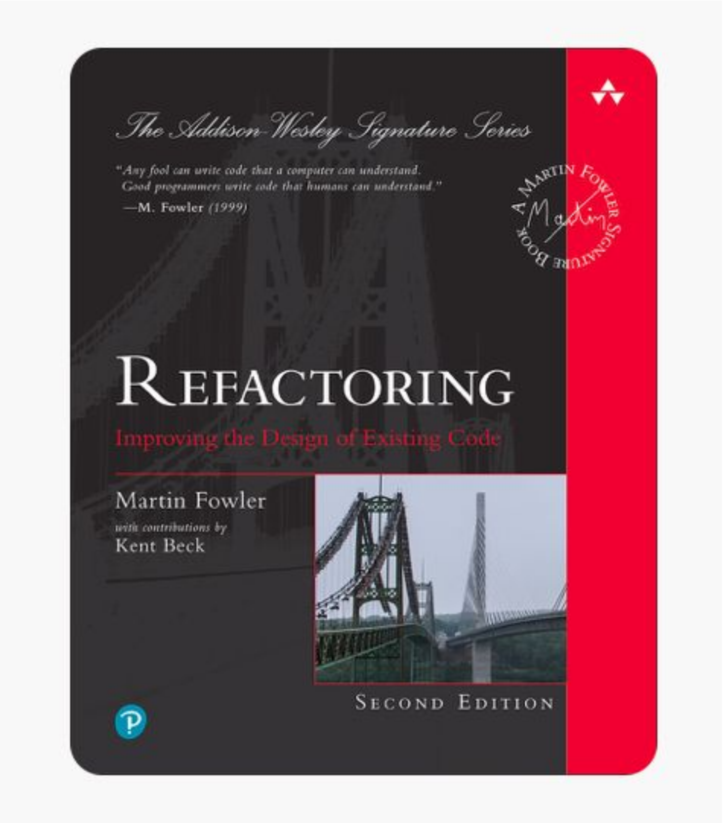
https://exceptionnotfound.net/cargo-cult-programming-the-daily-software-anti-pattern/

Today

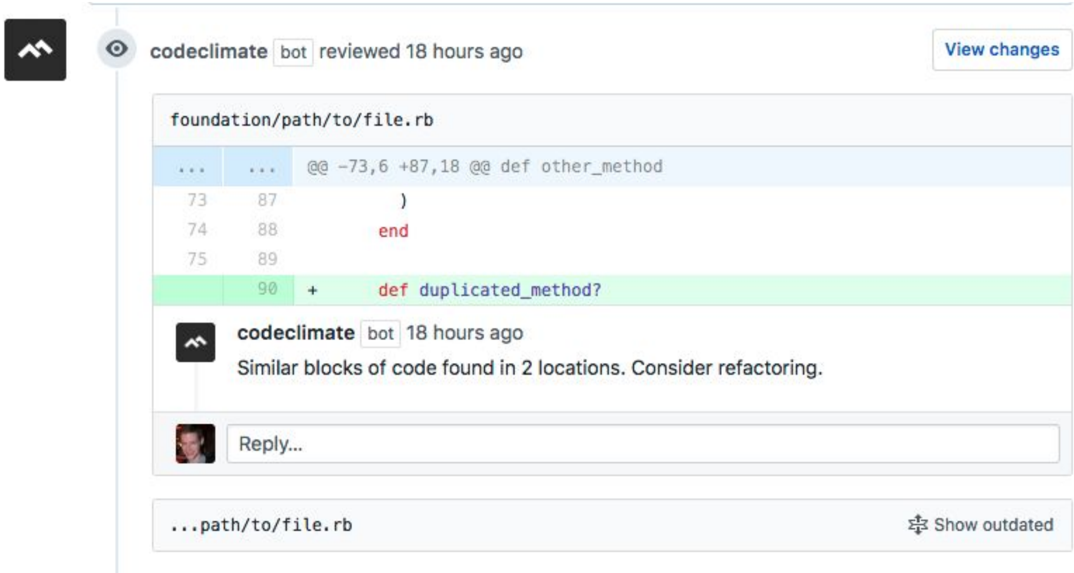
- Software quality
- Technical debt
- Anti-patterns
- Code smells

What is a code smell?

- A *code smell* is a hint that something has gone wrong somewhere in your code.
- A smell is *sniffable*, or something that is quick to spot.
- A smell doesn’t *always* indicate a problem.


The Addison-Wesley Signature Series
“Any fool can write code that a computer can understand. Good programmers write code that humans can understand.” —M. Fowler (1999)
A Martin Fowler Signature Book
Refactoring
Improving the Design of Existing Code
Martin Fowler
with contributions by
Kent Beck
Second Edition

Smell checks can be manual or automatic



codeclimate bot reviewed 18 hours ago

foundation/path/to/file.rb

```
...   ...   @@ -73,6 +87,18 @@ def other_method
73    87             )
74    88          end
75    89
      90    +      def duplicated_method?
```

codeclimate bot 18 hours ago

Similar blocks of code found in 2 locations. Consider refactoring.

...path/to/file.rb

SpotBugs

CODE CLIMATE

coverity™
a higher code™

Travis CI

Code Smells

1. Lack of polymorphism
2. Divergent change
3. Shotgun surgery
4. Mysterious names
5. Long methods
6. Large classes
7. Primitive obsession
8. Long parameter lists
9. Data clumps
10. Duplicated code
11. Dead code
12. Stinky comments

Lack of polymorphism

```
public void doSomething(Account acct) {
  long adj = 0;
  if (acct instanceof CheckingAccount) {
    checkingAcct = (CheckingAccount) acct;
     adj = checkingAcct.getFee();
  } else if (acct instanceof SavingsAccount) {
    savingsAcct = (SavingsAccount) acct;
     adj = savingsAcct.getInterest();
  }
  …
}
```

Instead:

```
public void doSomething(Account acct) {
  long adj = acct.getMonthlyAdjustment();
  …
}
```

Long parameter lists

```
public class User {
  ...
  public User(String firstName,
              String lastName,
              int age,
              String address,
              String phone)
  {
    this.firstName = firstName;
    this.lastName = lastName;
    this.age = age;
    this.address = address;
    this.phone = phone;
  }
}
```

Code becomes had to read and maintain with many attributes!

Solution: Use a Builder to hold build instructions.

```
public class User {
  private final String firstName;
  private final String lastName;
  private final int age;
  private final String address;
  private final String phone;

  private User(UserBuilder builder) {
    this.firstName = builder.firstName;
    this.lastName = builder.lastName;
    …
  }

  public String getFirstName() { … }
  public String getLastName() { … }
  …
}
```

```
new User.Builder(“Fred”, “Rogers”)
    .age(30)
    .phone(“1234567”)
    .address(...)
    .build();
```

```
public static class Builder {
  private final String firstName;
  private final String lastName;
  private int age;
  private String address;
  private String phone;

  private UserBuilder(String firstName,
                      String lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }

  public UserBuilder age(int age) {
    this.age = age;
    return this;
  }
  public UserBuilder phone(String phone) {
    this.phone = phone;
    return this;
  }
  …
}
```

In general, you can introduce a Parameter object

Primitive obsession

Common abuses:

- Phone numbers
- Currency
- Physical units
- Email addresses
- Zip codes
- Coordinates
- Ranges

Using primitives to represent types.

- No type checking!
- Poor encapsulation

Variables represented by strings are known as *stringly-typed variables*.

Solution: Replace primitives with strongly-typed value objects

Data clumps

Whenever two or more values are gathered together, turn them into an object (e.g., database connections, coordinates).

```
public bool submitCreditCardOrder(string firstName,
                                  string lastName,
                                  string zipcode,
                                  string streetAddress1,
                                  string streetAddress2,
                                  string city,
                                  string state,
                                  string country,
                                  string phoneNumber,
                                  string creditCardNumber,
                                  int expirationMonth,
                                  int expirationYear,
                                  BigDecimal saleAmount)
{
  …
}
```

https://scotkelly.wordpress.com/2014/08/24/data-clumps-code-smell/

Data clumps

Whenever two or more values are gathered together, turn them into an object (e.g., database connections, coordinates).

```
public bool submitCreditCardOrder(ContactInformation customerInfo,
                                  CreditCard card,
                                  BigDecimal saleAmount)
{
  …
}
```

Benefits:

- Cleaner code
- Type checking and data validation
- Information hiding

Dead Code

As your software evolves, parts of the source code become *unused* or *unreachable* (e.g., if-else branches, parameters)

Solution: If you can, delete the dead code! If it's an API, deprecate the method and eventually remove.

Stinky Comments

```
// prompt the user for their name using System.out, which
// is a PrintStream class. The PrintStream class has a
// method called println, which will output the text
// passed to the console (so that the user can see it)
// and then print a newline.
System.out.println("Welcome to my program! What is your name? ");
```

```
/* set the value of the age integer to 32 */
int age = 32;

// declare double-type variables
double salePrice;
double priceWithTax;
```

```
// if (opt.equals("d"))
//   isDebug = true;
```

```
// TODO implement missing branch!

// BUG this code doesn’t actually work -- woops! :-)

// FIXME I should probably implement those features in my API
```

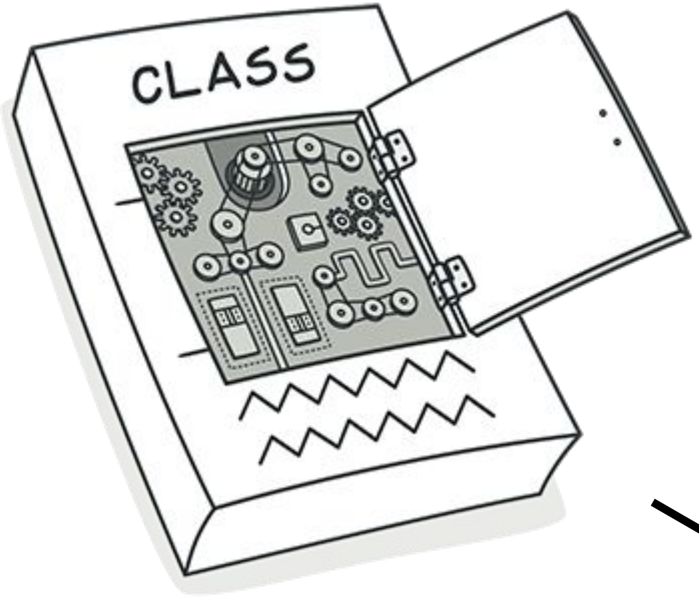
https://rmarcus.info/blog/2018/11/05/good-bad-comment.html

Duplicated Code

- Need to maintain multiple copies!
- Slows down development.
- Very easy to forget to modify a copy and to introduce a bug.
- Harms comprehension.

Solutions: *Extract Functions, Slide Statements, Pull Up Method*

Divergent change

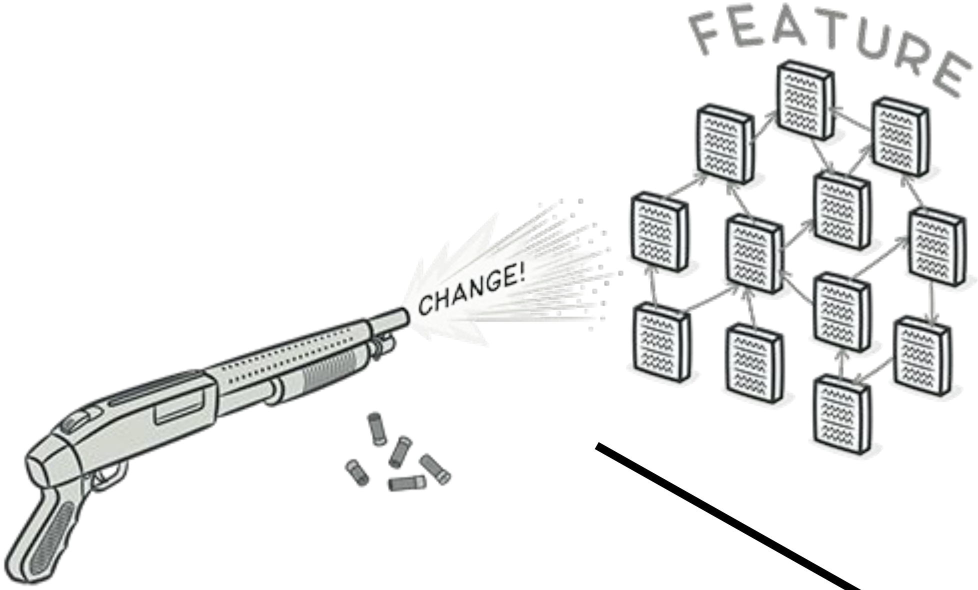


Changing a class requires additional changes to unrelated methods in that class.

Try to decompose the concerns of the class into multiple classes.

https://refactoring.guru/smells/divergent-change

The opposite smell: Shotgun surgery

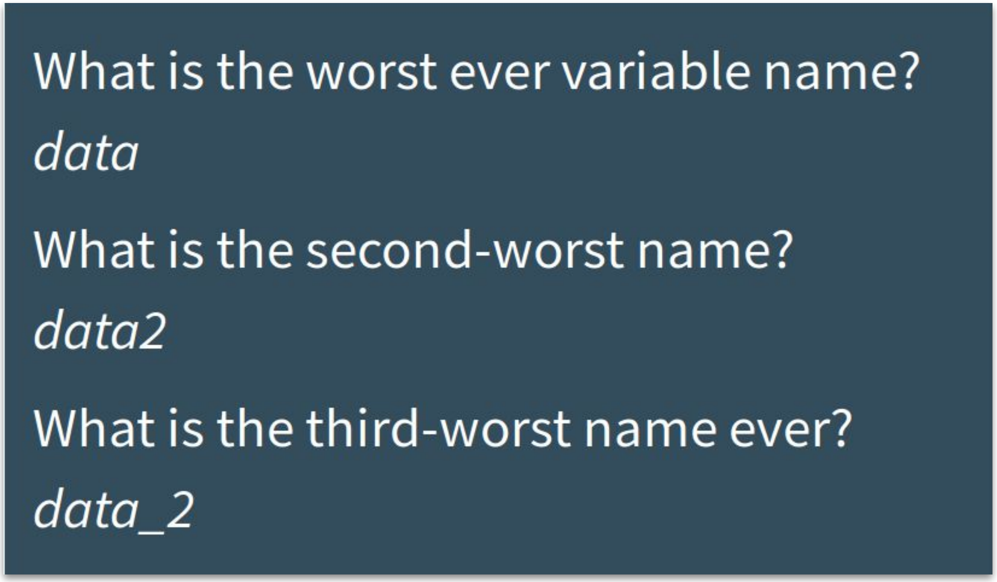


Making a change requires lots of small changes to a large number of classes.

Try to collapse methods and fields into a single class.

https://refactoring.guru/smells/shotgun-surgery

Mysterious names



What is the worst ever variable name?

data

What is the second-worst name?

data2

What is the third-worst name ever?

data_2

- Name should be concise and meaningful.
- If it’s really hard to come up with a name, you may have a deeper design problem!

Solution: Take the time to rename your methods, variables, and fields.

How to name things:

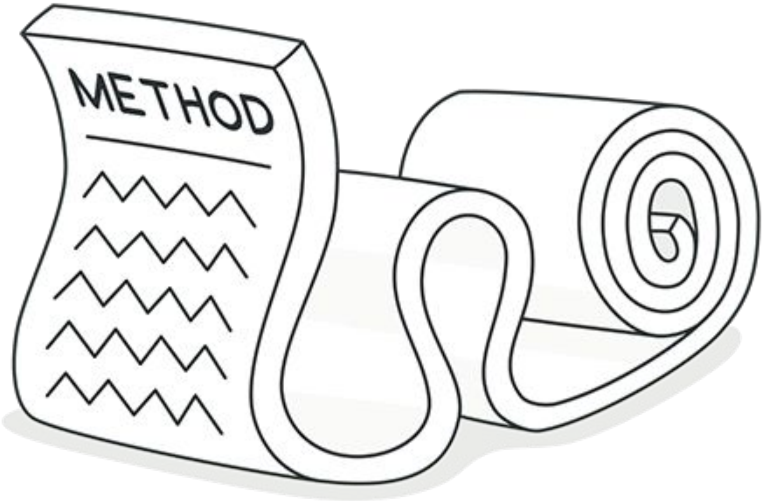
the hardest problem in programming

@PeterHilton

http://hilton.org.uk/

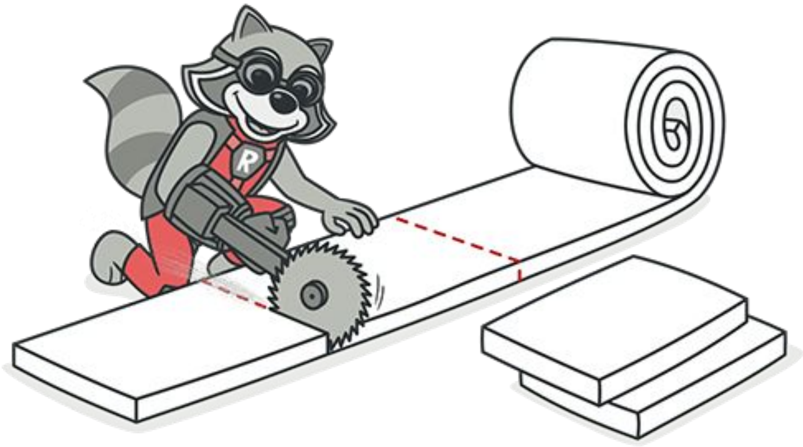
https://hilton.org.uk/blog/naming-smells

Long methods



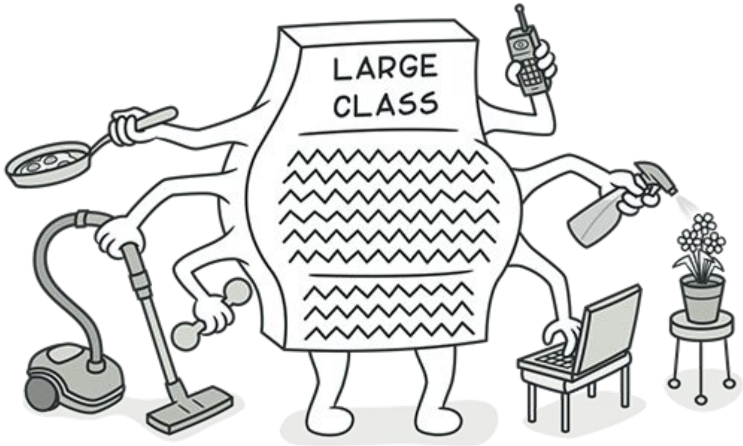
- Difficult to understand
- Hard to debug*
- Redundant code?
- Poor code?

Solution: Decompose large methods into smaller methods that capture different steps



Large classes

- *Suggests* bad OO design
- Multiple responsibilities?
- Duplicate or redundant code?



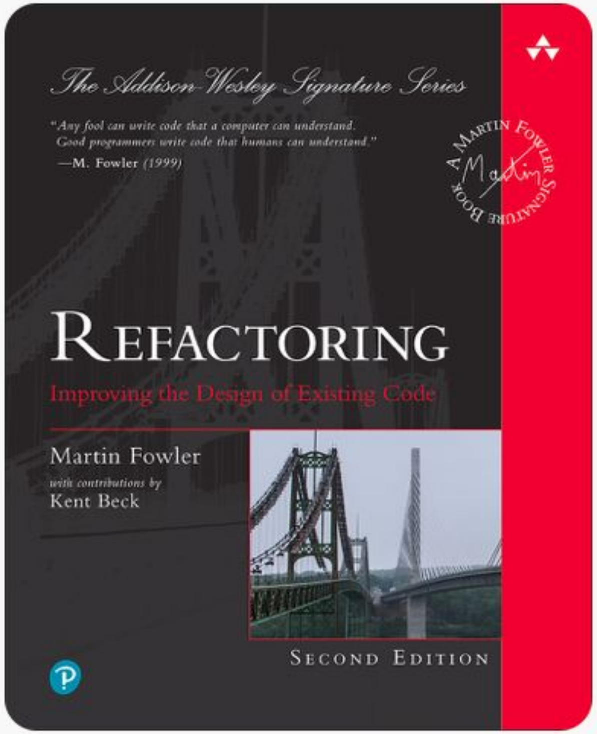
Solution: Break up class into multiple, smaller classes, each with a *single responsibility*.

https://refactoring.guru/smells/large-class

There are lots of code smells!

To learn more, check out:

- *Refactoring: Improving the Design of Existing Code* by Martin Fowler
- https://refactoring.guru


The Addison-Wesley Signature Series
"Any fool can write code that a computer can understand. Good programmers write code that humans can understand."
—M. Fowler (1999)
A Martin Fowler Signature Book
Refactoring
Improving the Design of Existing Code
Martin Fowler
with contributions by
Kent Beck
Second Edition

Summary

- Software accumulates technical debt as it evolves. Technical debt introduces *cruft* and slows down development. The longer technical debt lingers, the more problems it creates.
- Refactoring is used to continually reduce technical debt.
- Anti-patterns represent common programming, design, and process failures that should be *avoided*.
- Code smells *suggest* problems with your code and design.
- Eliminating smells via refactoring can reduce cruft.