CS 4530
Fundamentals of Software Engineering

Lesson 11: Code Smells, Refactoring and Technical Debt

Jonathan Bell, Adeel Bhutta, Ferdinand Vesely, Mitch Wand
Khoury College of Computer Sciences

Learning Goals

By the end of this lesson, you should be able to…

1. Some common code “smells” (anti-patterns).
2. “Refactoring”: restructuring of code to improve structure.
3. “Technical Debt”: generalization covering all internal problems in a code-base.

Refactoring

- **refactoring** is the process of applying transformations (refactorings) to a program, with the goal of improving its design
- goals:
  - keep program readable, understandable, and maintainable
  - by eliminating small problems soon, you can avoid big trouble later
- characteristics:
  - **behavior-preserving**: make sure the program works after each step
  - **small steps**

Learning Objectives for this Lesson

- By the end of this lesson, you should be able to:
  - Review several classes of code smells;
  - Describe several kinds of refactoring;
  - Identify the “technical debt” metaphor;
  - Indicate when and where technical debt is appropriate to accrue versus retire.

History of Refactoring

- refactoring is something good programmers have always done
  - Opdyke’s PhD thesis (1990): refactoring tools for Smalltalk
  - popularized by various agile development methodologies

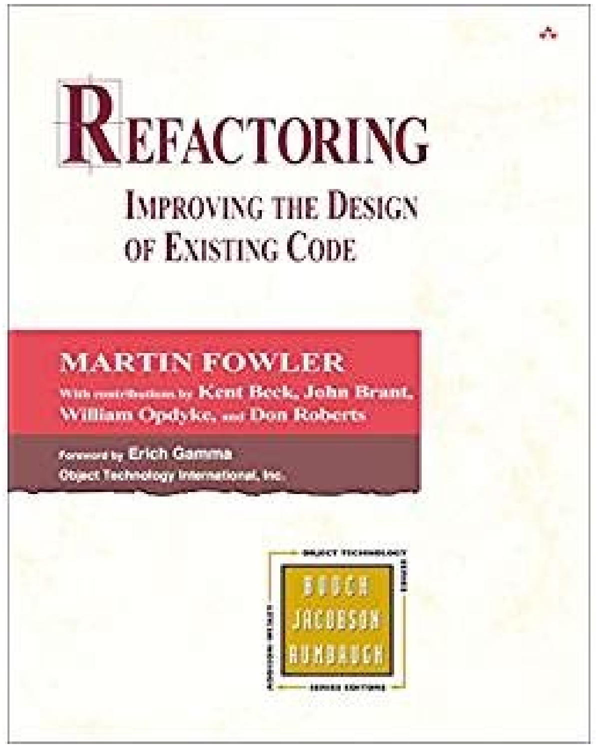
- especially popular in the context of object-oriented languages
  - OO features are well-suited to make designs flexible & reusable
  - but refactoring is not specific to OO

Refactoring

Martin Fowler

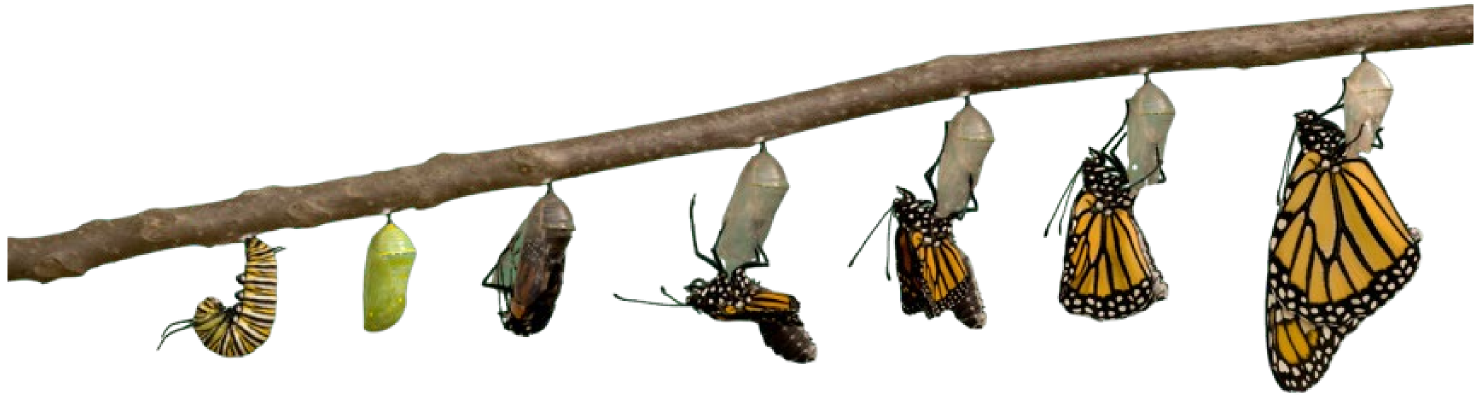
"Any fool can write code that a computer can understand. Good programmers write code that humans can understand."

Fowler's book



- presents a **catalogue of refactorings**, similar to the catalogue of design patterns in the GoF book
  - catalogues "bad smells" - indications that refactoring may be needed
  - explains when and how to apply refactorings
- many of Fowler's refactorings are the inverse of another refactoring
  - often there is not a unique "best" solution
  - discussion of the tradeoffs

Why Refactor?



- requirements have changed, and a different design is needed
- design needs to be more flexible (so new features can be added)
  - design patterns are often a target for refactoring
- address sloppiness by programmers

Example Refactoring

Consolidating duplicate conditional fragments

Original Code

```
if (isSpecialDeal()) {
    total = price * 0.95;
    send()
} else {
    total = price * 0.98;
    send()
}
```

Refactored Code

```
if (isSpecialDeal()) {
    total = price * 0.95;
} else {
    total = price * 0.98;
}
send()
```

Observations

- **small incremental steps** that preserve program behavior

- most steps are so simple that they can be **automated**
  - automation limited in complex cases

- refactoring does not always proceed “in a straight line”
  - sometimes, undo a step you did earlier…
  - …when you have insights for a better design

When to refactor?

Refactoring is incremental redesign

- Acknowledge that it will be difficult to get design right the first time
- When adding new functionality, fixing a bug, doing code review, or any time
- Refactoring evolves design in increments
- Refactoring reduces technical debt
- What do you refactor?

Code Smells

Mysterious Name

“We may fantasize about being International Men of Mystery, but our code needs to be mundane and clear”

- Martin Fowler on “Mysterious Name”

Code Smells

Shotgun Surgery

> “When the changes are all over the place, they are hard to find, and it’s easy to miss an important change.”

- Martin Fowler on “Shotgun Surgery”

Code Smells

A complete list (links to book!)

- Mysterious Name
- Duplicated Code
- Long Function
- Long Parameter List
- Global Data
- Mutable Data
- Divergent Change
- Shotgun Surgery
- Feature Envy
- Data Clumps
- Primitive Obsession
- Repeated Switches
- Loops
- Lazy Element
- Speculative Generality
- Temporary Field
- Message Chains
- Middle Man
- Insider Trading
- Large Class
- Alternative Classes with Different Interfaces
- Data Class
- Refused Bequest

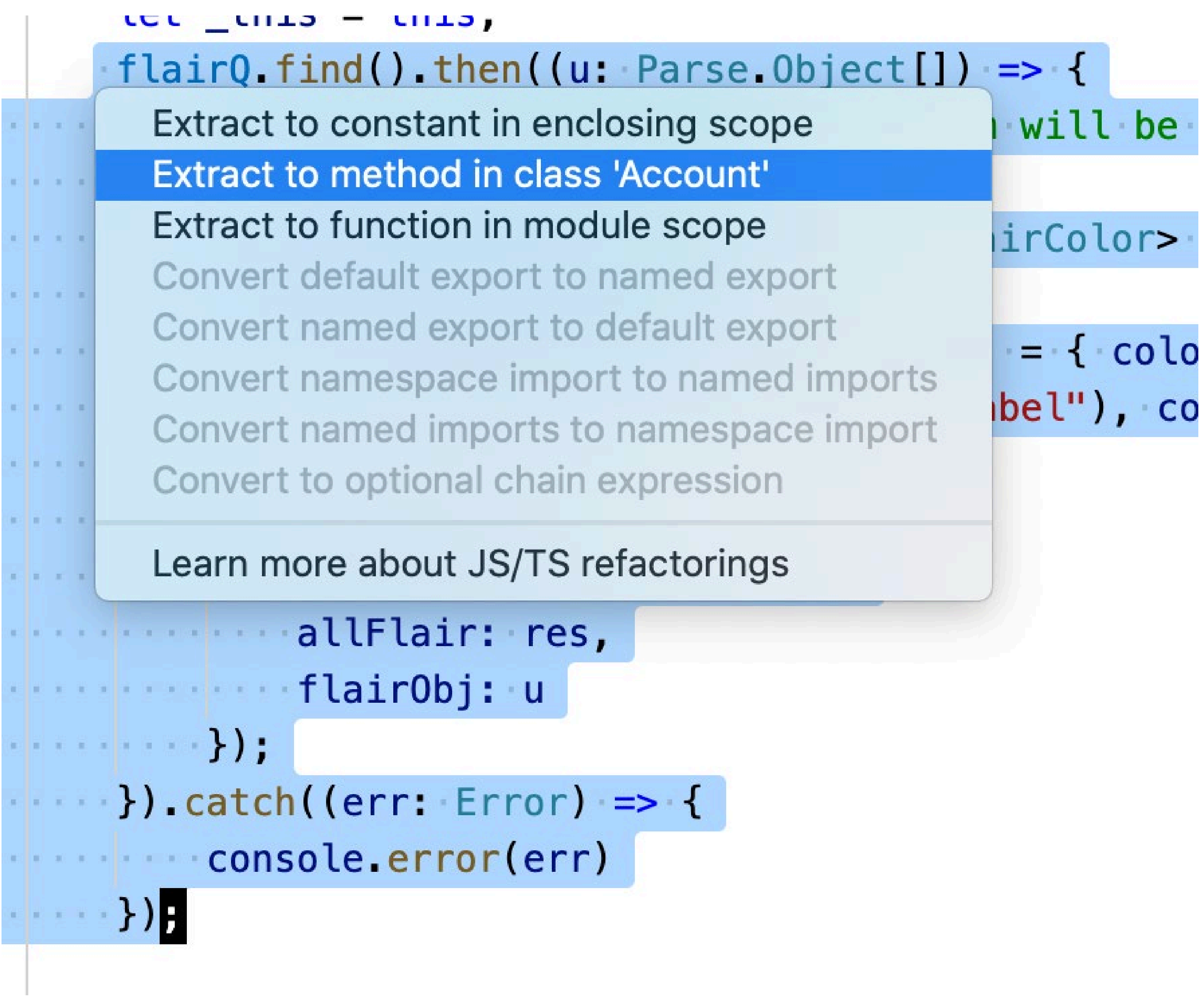
“Local” Refactorings

Rename	rename variables, fields methods, classes, packages provide better intuition for the renamed element’s purpose
Extract Method	extract statements into a new method enables reuse; avoid cut-and-paste programming improve readability
Inline Method	replace a method call with the method’s body often useful as intermediate step
Extract Local	introduce a new local variable for a designated expression
Inline Local	replace a local variable with the expression that defines its value
Change Method Signature	reorder a method’s parameters
Encapsulate Field	introduce getter/setter methods
Convert Local Variable to Field	convert local variable to field sometimes useful to enable application of Extract Method

Type-Related Refactorings

Generalize Declared Type	replace the type of a declaration with a more general type
Extract Interface	create a new interface, and update declarations to use it where possible
Pull Up Members	move methods and fields to a superclass
Infer Generic Type Arguments	infer type arguments for “raw” uses of generic types

Automated Refactorings in VSC



Refactoring Risks

- Developer time is valuable: is this the best use of time *today*?
- Despite best intentions, may not be safe
- Potential for version control conflicts

This leads us into Technical Debt

- Code smells and Refactoring are tightly coupled with Technical debt

- In software-intensive systems, technical debt consists of *design or implementation* constructs that are expedient in the short term but that set up a technical context that can make a *future change* more costly or impossible.

Technical Debt is Internal but affects maintainability and evolvability

- Technical debt is a contingent liability whose impact is limited to internal system qualities—primarily, but not only, maintainability and evolvability.
- Usual Scenarios:
  - Quick-and-Dirty if-then-else;
  - Hitting the Wall;
  - Crumbling Under the Load;
  - Death by a Thousand Cuts;
  - Tactical Investment.

Not just code!

Technical Debt is Sum of Internal Problems in Project Codebase

Example of Debt

- Code Smells;
- Missing tests;
- Missing documentation;
- Dependency on old versions of third-party systems;
- Inefficient and/or non-scalable algorithms.

Example of Cost

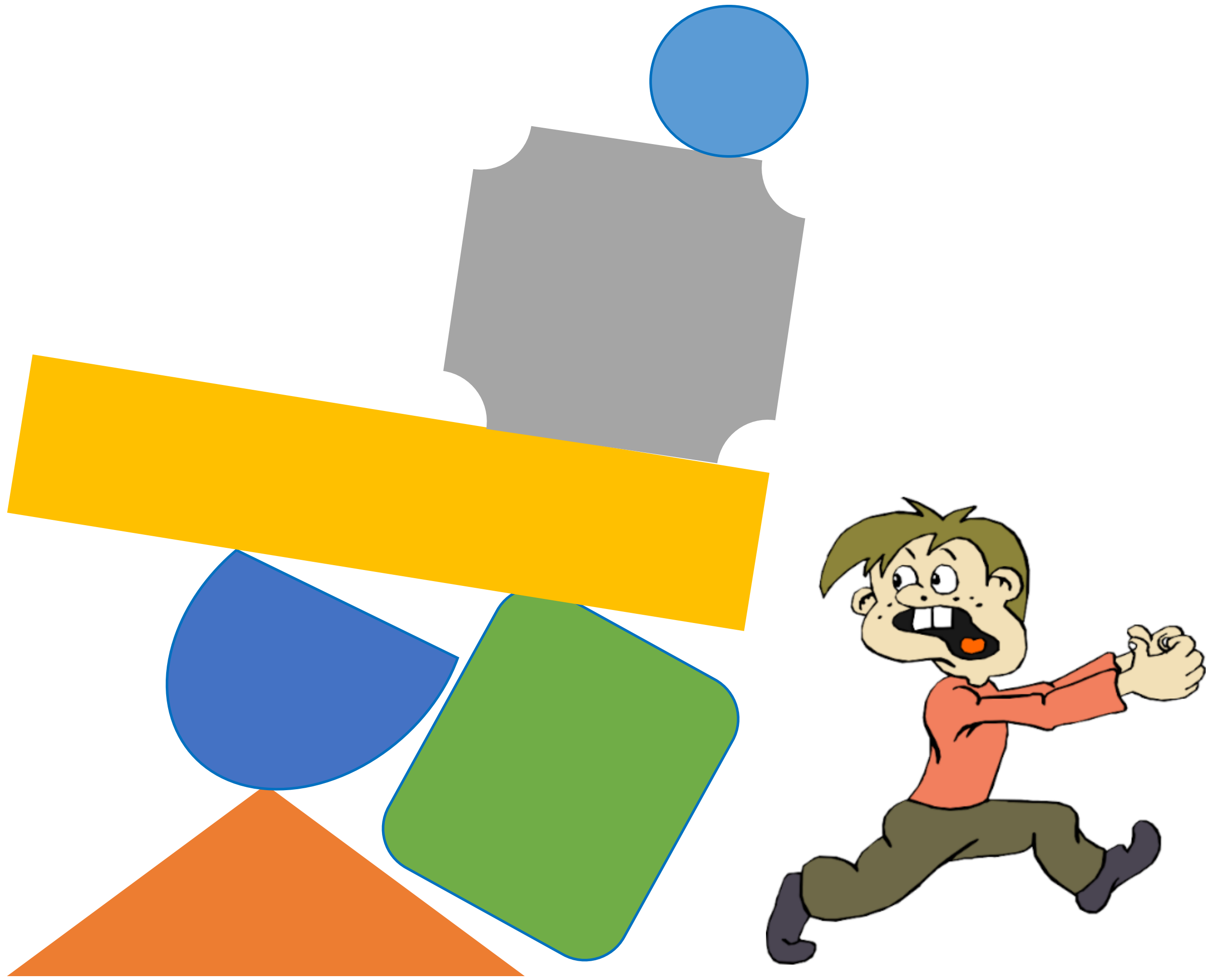
- “Smelly” code is less flexible;
- Need to revert breaking change;
- Can’t figure out how to use;
- May have take over maintenance of old system;
- Lose potential customers.

Good Reasons to Go Into Technical Debt

- Prototyping:
  - If code will be discarded, or drastically rewritten, don’t waste time perfecting it.
- Getting a product out the door:
  - Time is often crucial in a competitive environment.
- Fixing a critical failure:
  - People are waiting.
- Maybe a simple algorithm is good enough:
  - “Premature optimization is the root of all evil”
    - Tony Hoare, Donald Knuth

Retire Technical Debt at Leisure

- Set aside time to pay off technical debt:
  - Google has (had?) “20%-time” for tasks such as this.
- A new initiative can take on some technical debt:
  - Refactoring at the start of a project.
- Don’t keep on putting off!
  - When a crisis hits, it’s too late;
  - Hasty fixes to unmaintainable code multiplies problems;
  - Eventually mounting technical debt can bury the team.



Review: Learning Objectives for this Lesson

- You should now be able to:
  - Review several classes of code smells;
  - Describe several kinds of refactoring;
  - Identify the “technical debt” metaphor;
  - Indicate when and where technical debt is appropriate to accrue versus retire.