CS 4530
Fundamentals of Software Engineering
Lesson 11: 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…

- Define “refactoring” and give examples.
- Explain how refactoring fits into an agile development process
- Define “technical debt”
- Suggest when it may be appropriate to accrue technical debt and when it may be appropriate to retire it.

Part 1: Refactoring

Refactoring

- **Refactoring** is the process of applying transformations (refactorings) to a program, but the internal structure of the system is improved
- 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**

Example Refactoring

Consolidating duplicate conditional fragments

Original Code

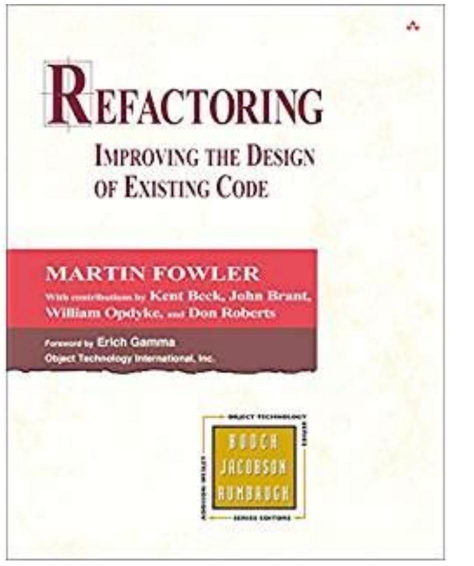
```
if (isSpecialDeal()) {
    total = price * 0.95;
    send()
} else {
    total = price * 0.98;
    send()
}
```

Refactored Code

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

Martin Fowler is the “father” of refactoring

“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
  - Gave names to each transformation
    - Helpful for team communication
    - Identified and named “bad smells” (indications that refactoring may be needed)
    - Discusses 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

Fowler gave colorful names to many of the “code smells” he identified

A complete list (with 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

“Refactoring: Improving the Design of Existing Code,” Martin Fowler, 1992

The most common refactoring is renaming

- Rename Function (124) (to rename a function)
- Rename Variable (137)
- Rename Field (244).
- People are often afraid to rename things, thinking it's not worth the trouble, but a good name can save hours of puzzled incomprehension in the future.
- Renaming is not just an exercise in changing names. When you can't think of a good name for something, it's often a sign of a deeper design malaise. Puzzling over a tricky name leads to significant improvements to your code

"Refactoring: Improving the Design of Existing Code," Martin Fowler, 1992

Luckily, VSC automates this and many other common transformations

“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

Why Refactor?

- New or anticipated requirements require a different design
- Altered design will make testing easier
- Altered design will improve maintainability
- Fix sloppiness by programmers
  - Retire or avoid technical debt

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
- A key part of TDD!
- Refactoring evolves design in increments
- Refactoring reduces technical debt
- What do you refactor?

Refactoring Benefits

- **small incremental steps** that preserve program behavior
  - Regression testing is simplified
- 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, you want to undo a step you did earlier…
  - …when you have insights for a better design
  - Having a name for what you did makes it easier to undo a step
    - (but of course there’s always git!)

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

Part 2: Technical Debt

Technical Debt is the Accumulation of Internal Problems in Project Codebase

- Internal because they don't show as user-visible failures.
- Examples:
  - Code Smells;
  - Missing tests;
  - Missing documentation;
  - Dependency on old versions of third-party systems;
  - Inefficient and/or non-scalable algorithms.

Not just code!

Technical Debts have costs (“interest” on the debt).

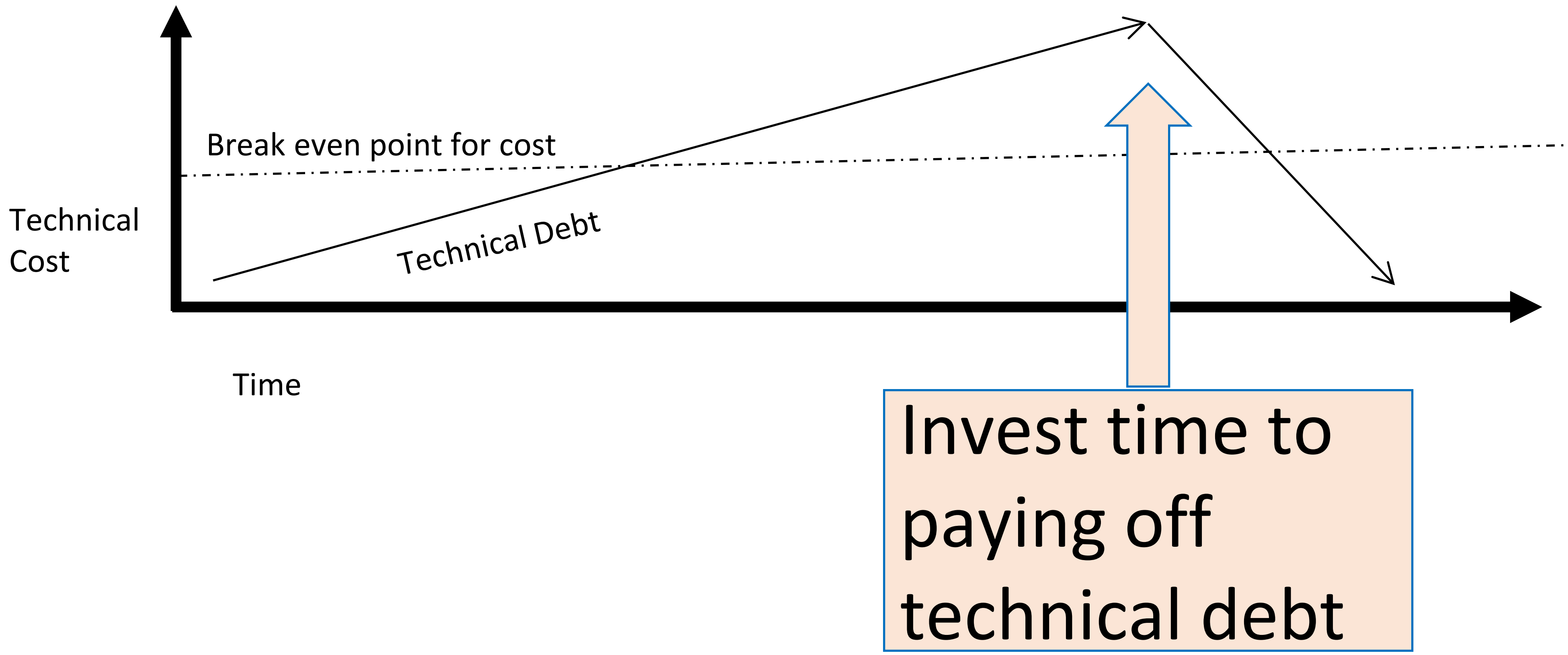
Example of Debt

- 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.

Interest on Technical Debt Accrues over Time



Good Reasons to Go Into Technical Debt

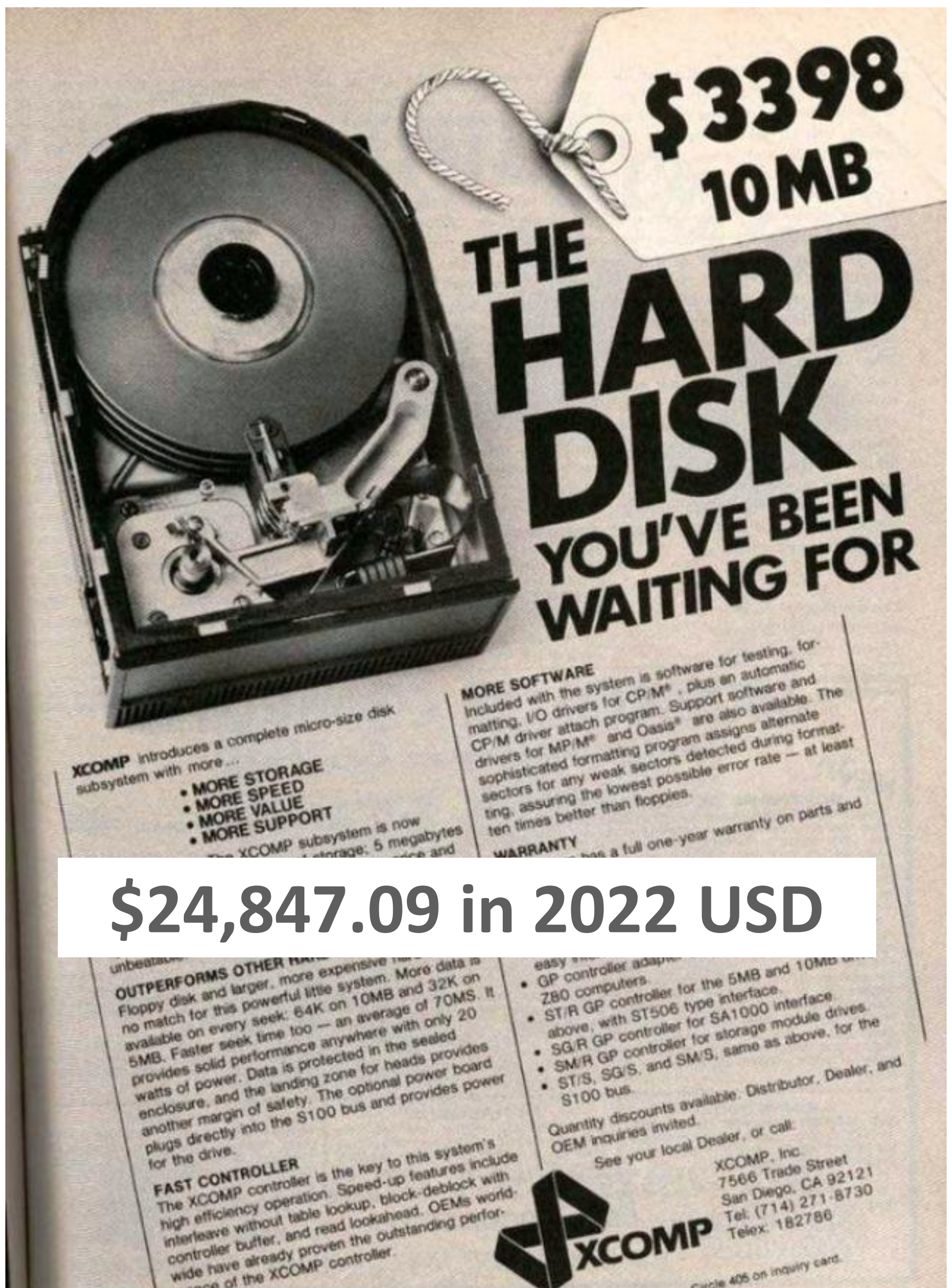
- Prototyping:
  - If code will be discarded, or drastically rewritten, don’t waste time perfecting it.
- Getting a product out the door:
  - Time is often crucial in a competitive environment.
- Fixing a critical failure:
  - People are waiting.
- Maybe a simple algorithm is good enough:
  - “Premature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

Architectural Technical Debt is Most Expensive

- Total cost of ownership generally higher than implementation-level issues; harder to get out of choices of:
  - Language
  - Middleware frameworks
  - Deployment pipeline
- Consider: What are the quality attributes that our software needs to ultimately satisfy, and how do these architectural decisions reflect those attributes?

The Y2K bug is an example of architectural technical debt

- How many digits does it take to store a year?

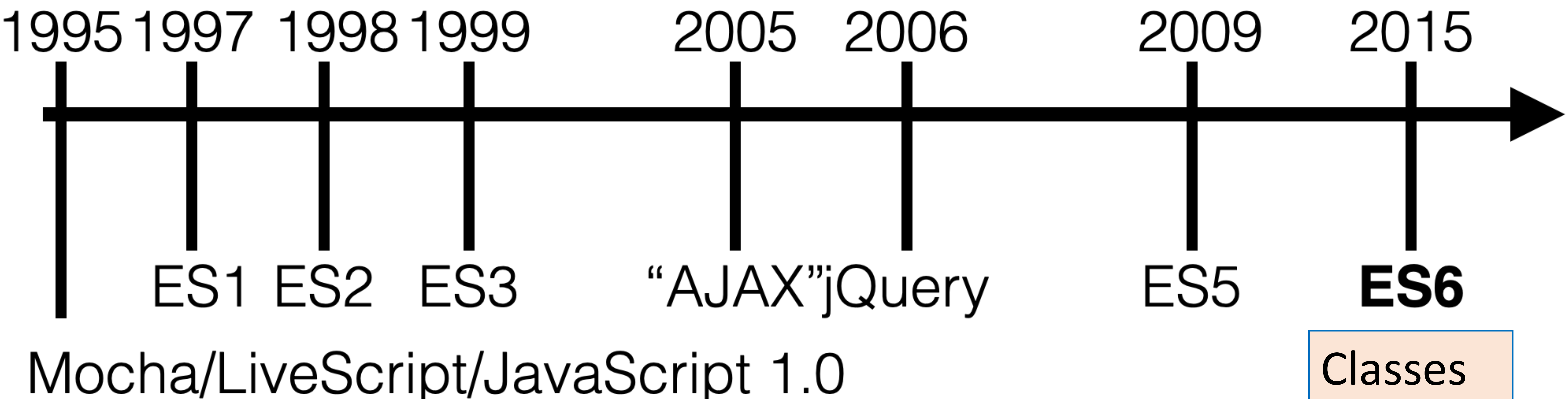


$24,847.09 in 2022 USD

"I just never imagined anyone would be using these systems 10 years later, let alone 20."

Philippe Kruchten, Robert Nord, Ipek Ozkaya:
"Managing Technical Debt: Reducing Friction in Software Development"

Evolving Languages bring Technical Debt



Classes
Promises

PLUS:
2016: ES7 (Array.includes)
2017: ES8 (Async/Await)
2018: ES9 (rest/spread operator, async iterators)

https://www.fastcompany.com/3028778/why-facebook-invented-a-new-php-derived-language-called-hack

Architectural Technical Debt: Facebook

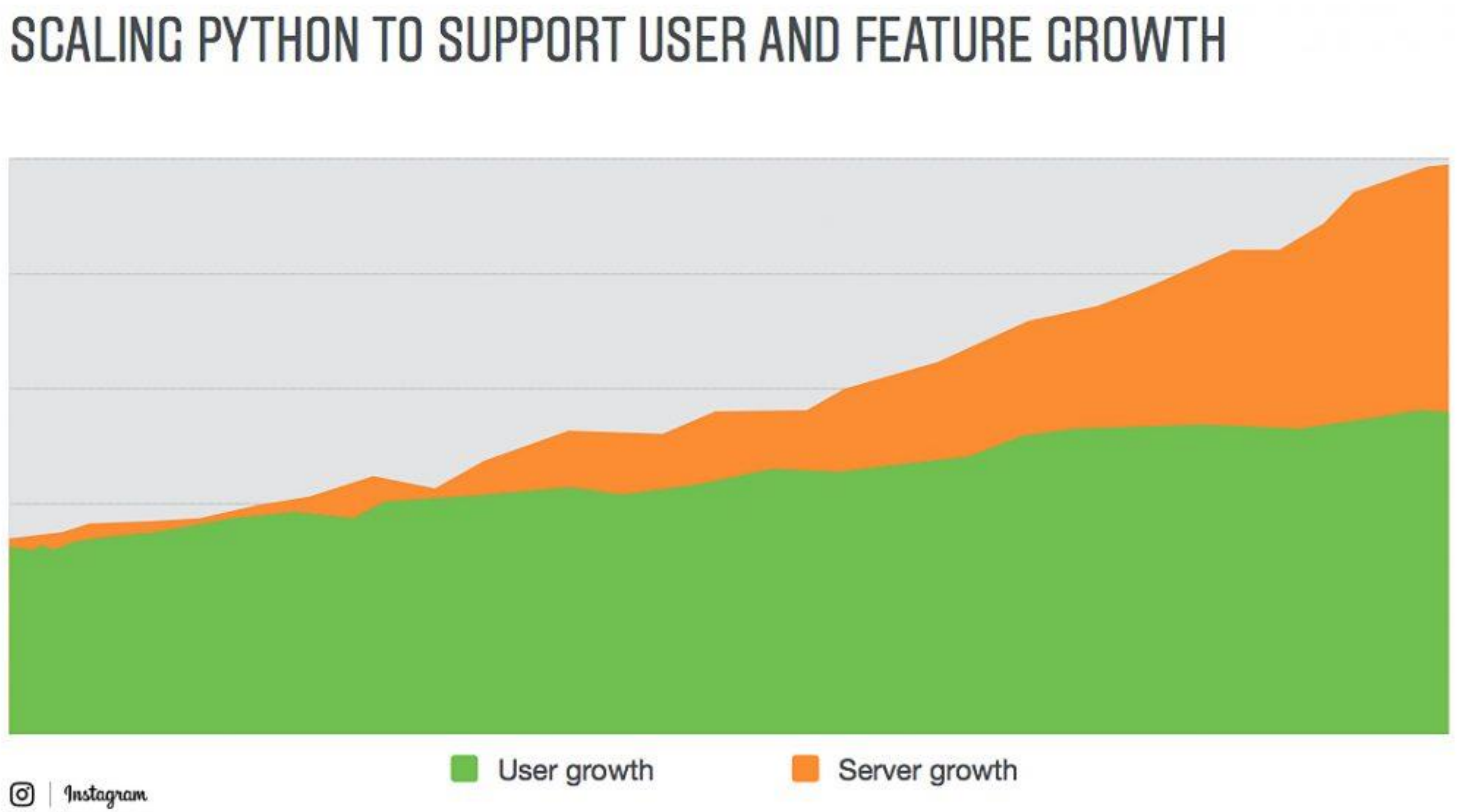
04-07-14

Why Facebook Invented A New PHP-Derived Language Called “Hack”

Instead of throwing out years of legacy code, Facebook built a new branch of the language that originally underpinned TheFacebook.com. Here’s the story behind a two-year labor of love.

https://www.fastcompany.com/3028778/why-facebook-invented-a-new-php-derived-language-called-hack

Architectural Technical Debt: Instagram

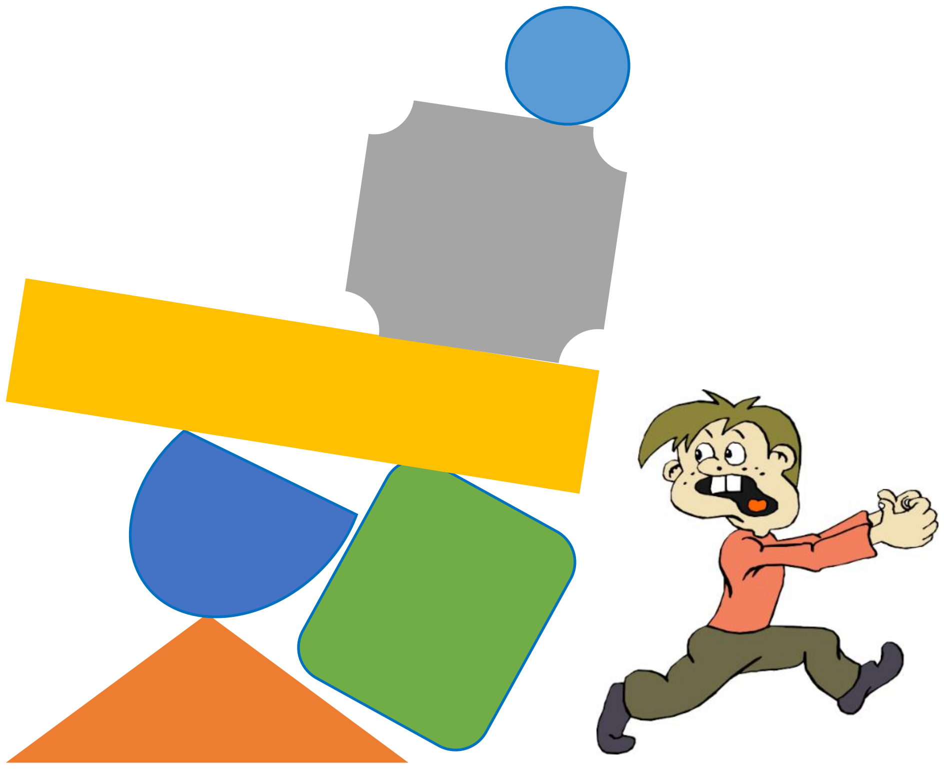


https://thenewstack.io/instagram-makes-smooth-move-python-3/

https://www.fastcompany.com/3028778/why-facebook-invented-a-new-php-derived-language-called-hack

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:
  - Define “refactoring” and give examples.
  - Explain how refactoring fits into an agile development process
  - Define “technical debt”
  - Suggest when it may be appropriate to accrue technical debt and when it may be appropriate to retire it.