# CS 4530
## Fundamentals of Software Engineering
## Module 16: Refactoring and Technical Debt

**Jonathan Bell, Adeel Bhutta, 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 and help reduce technical debt
- Define “technical debt”
- Suggest when it may be appropriate to accrue technical debt and when it may be appropriate to retire it.

# Let’s discuss Refactoring first

# Refactoring

- **Refactoring** is the process of applying transformations (refactorings) to a program, and the internal structure of the system is improved
- Goals:
  - keep program readable, understandable, and maintainable
  - by eliminating small problems soon, you can avoid big troubles later
- Characteristics:
  - **behavior-preserving**: make sure the program works after each step
  - **small steps**

# Example Refactoring

## Introduce Parameter and Extract optional Parameter

**Original Code**

```
function greeter (firstName : String, lastName : String) {
        return "Hello, " + firstName + " " + lastName;
}
document.body.innerHTML = greeter("Jane","User");
```

**Refactored Code # 1**

```
function greeter (firstName : String, lastName : String, greeting = "Hello, ") {
        return greeting + firstName + " " + lastName;
}
document.body.innerHTML = greeter("Jane","User");
```

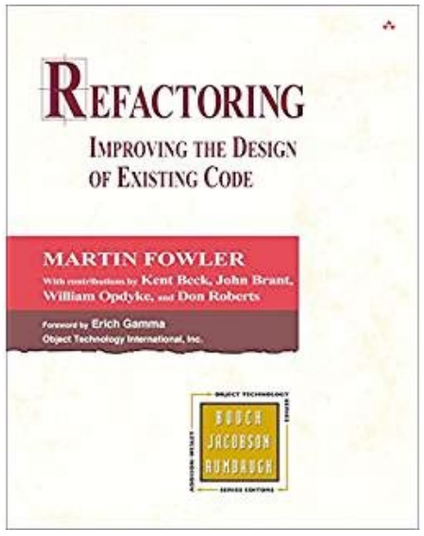
**Refactored Code # 2**

```
function greeter (firstName : String, lastName : String, greeting : String) {
        return greeting + firstName + " " + lastName;
}
document.body.innerHTML = greeter("Jane","User“,"Hello, ");
```

# 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

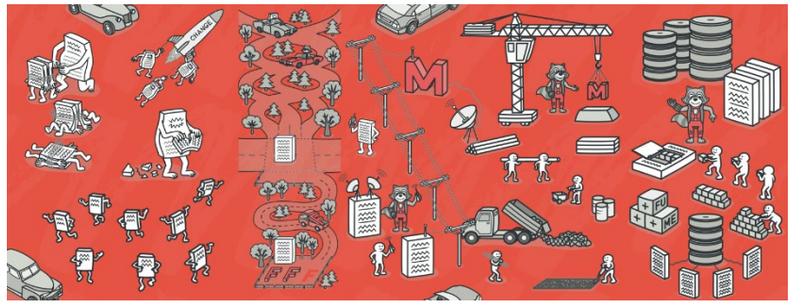
# The most common refactoring is renaming

- 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



https://refactoring.guru/

| | |
|---|---|
| **Rename** | rename variables, fields methods, classes, packages<br>provide better intuition for the renamed element’s purpose |
| **Extract Method** | extract statements into a new method<br>enables reuse; avoid cut-and-paste programming<br>improve readability |
| **Inline Method** | replace a method call with the method’s body<br>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<br>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 |

Commonly known as **Refactoring by Abstraction**

- Bad abstraction is worst than duplication (pieces of code that look the same, still represent different concepts).
  - Use “Rule of Three” – *Three strikes and you refactor*

https://understandlegacycode.com/blog/refactoring-rule-of-three/

# Why Refactor?

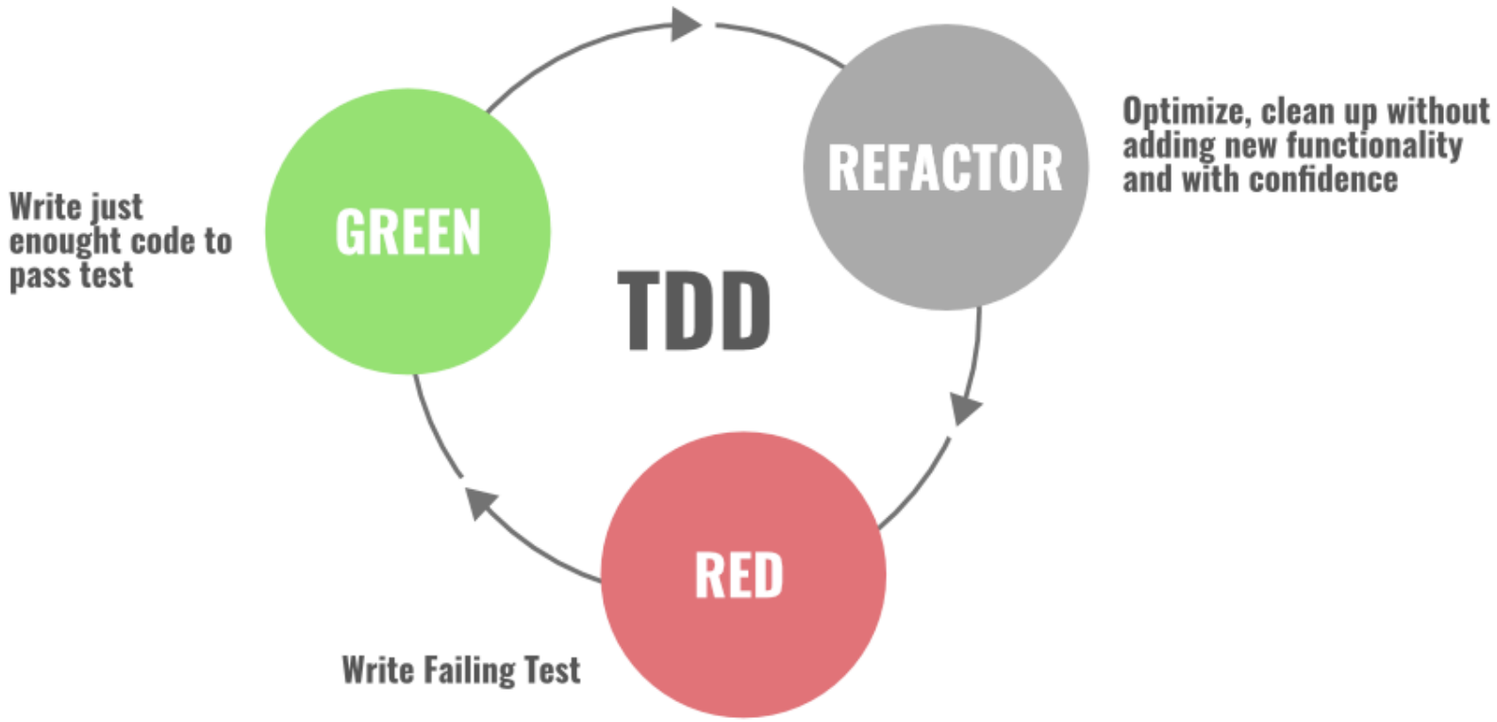
- New or anticipated requirements require a different design
- Altered design will make testing easier
- Altered design will improve maintainability
- Fix sloppiness by programmers
  - Retire or avoid technical debt

# 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 with TDD




- **RED**: The first step starts with writing the failing “red-test”. You stop and check what needs to be developed.
- **Green**: In the second step, you write the simplest enough code and get the development pass “green” testing.
- **Refactor**: In the final and third step, you focus on improving and enhancing your code keeping your test green.

# 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

# It brings us to 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).

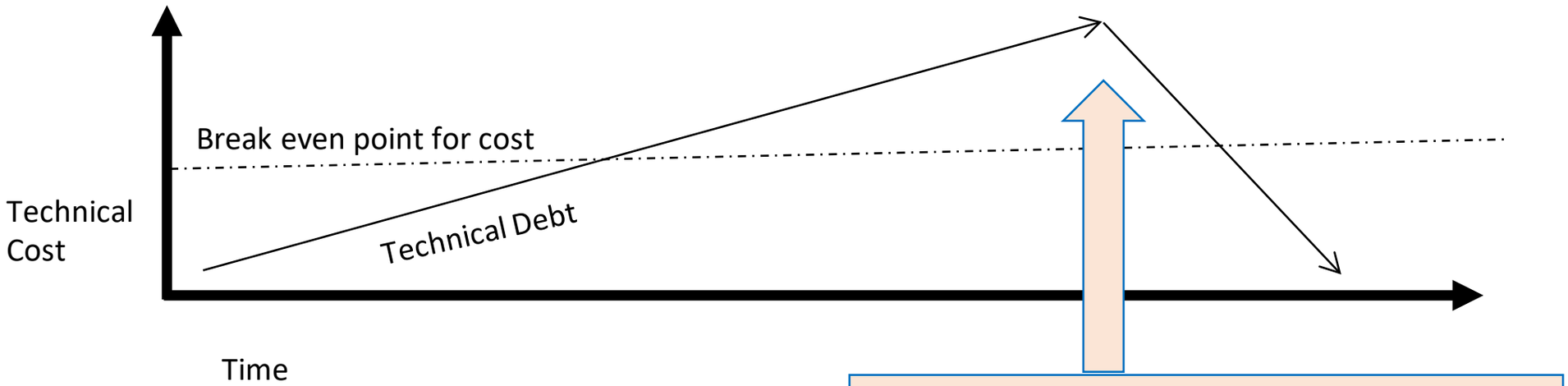
**Example of Debt**

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

**Example of Cost**

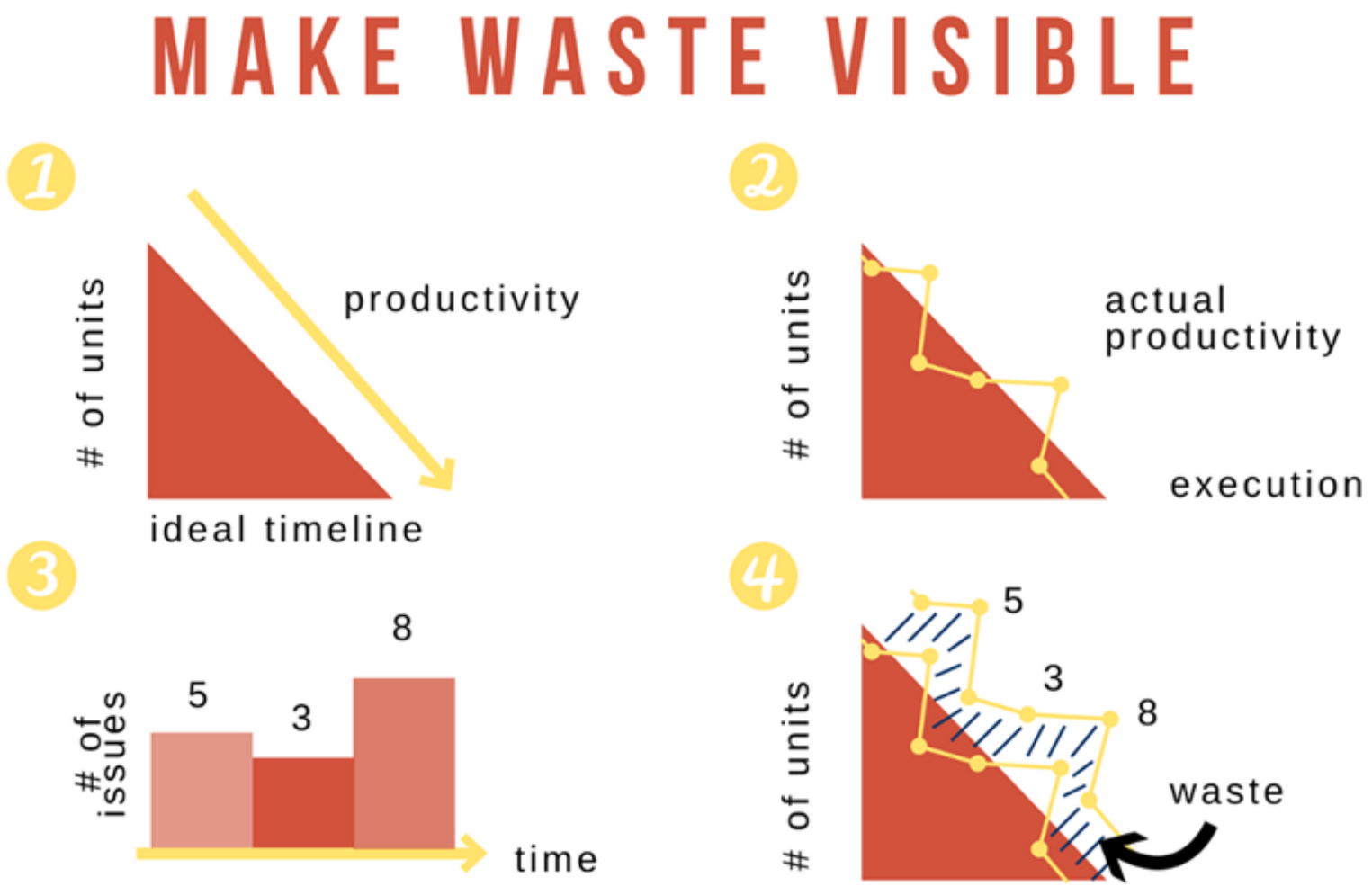
- "Smelly" code is less flexible;
- Need to revert breaking change;
- Can't figure out how to use;
- May have take over maintenance of old system;
- Lose potential customers.

# Interest on Technical Debt Accrues over Time




Invest time to paying off technical debt
=> Refactoring

# Make Technical Debt Visible




Here are the steps:

1. Plan the ideal.
2. Track your Actual.
3. Track what you spend on waste.
4. Put it all together

- Help stakeholders visualize data (like progress, effect of debt, refactoring)

https://www.scrum.org/resources/blog/making-tech-debt-visible

# Good Reasons to Go Into Technical Debt

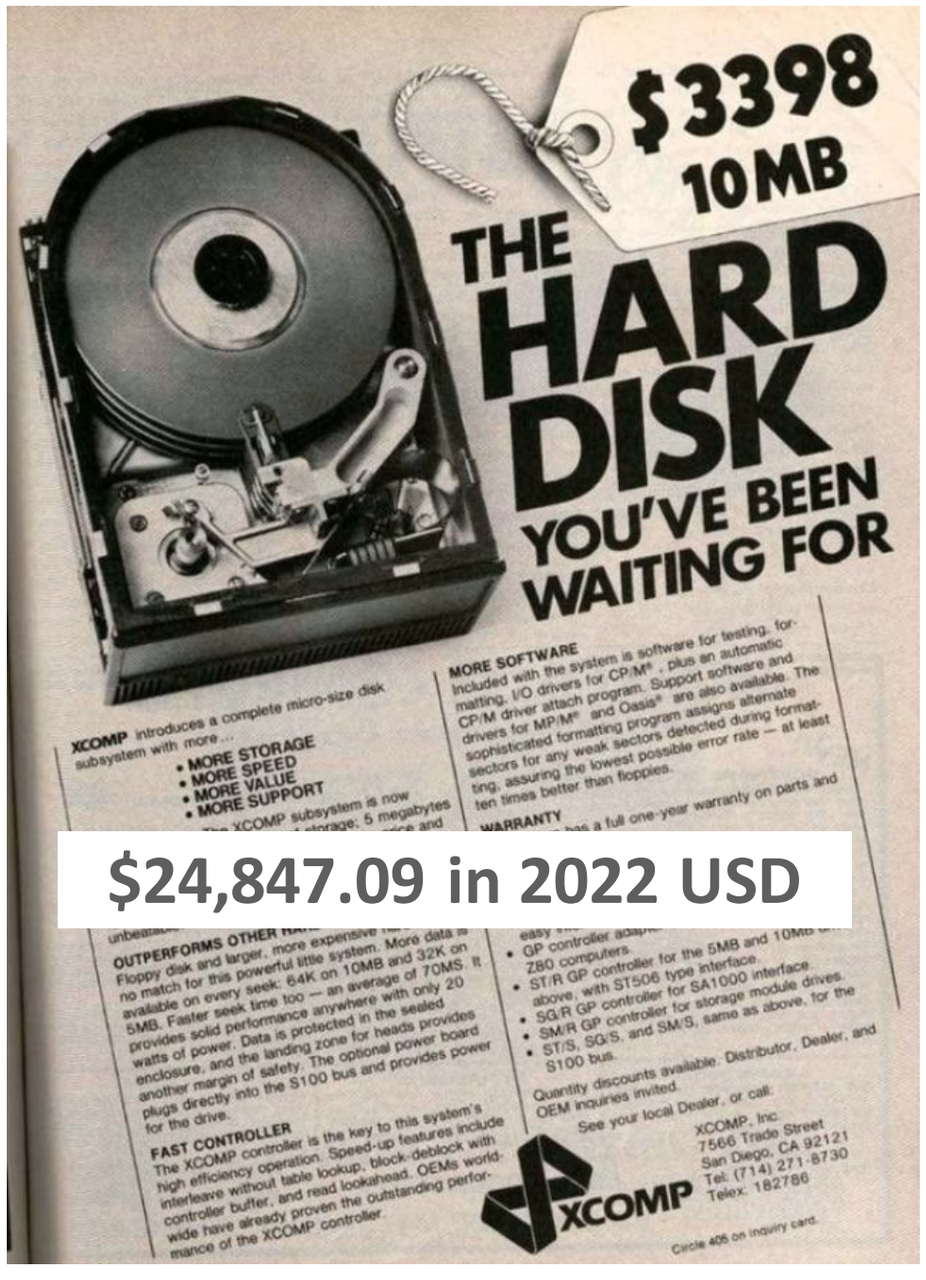
- Prototyping:
  - If code will be discarded, or drastically rewritten, don’t waste time perfecting it.
- Getting a product out the door:
  - Time is often crucial in a competitive environment.
- Fixing a critical failure:
  - People are waiting.
- Maybe a simple algorithm is good enough:
  - “Premature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

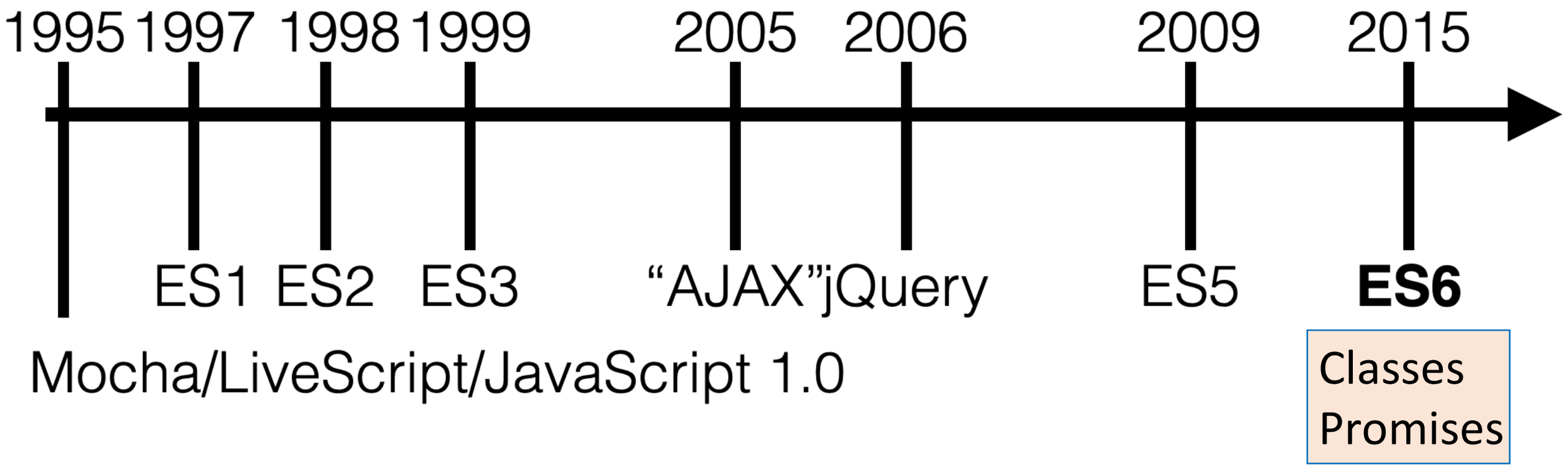



**$24,847.09 in 2022 USD**

"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




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.

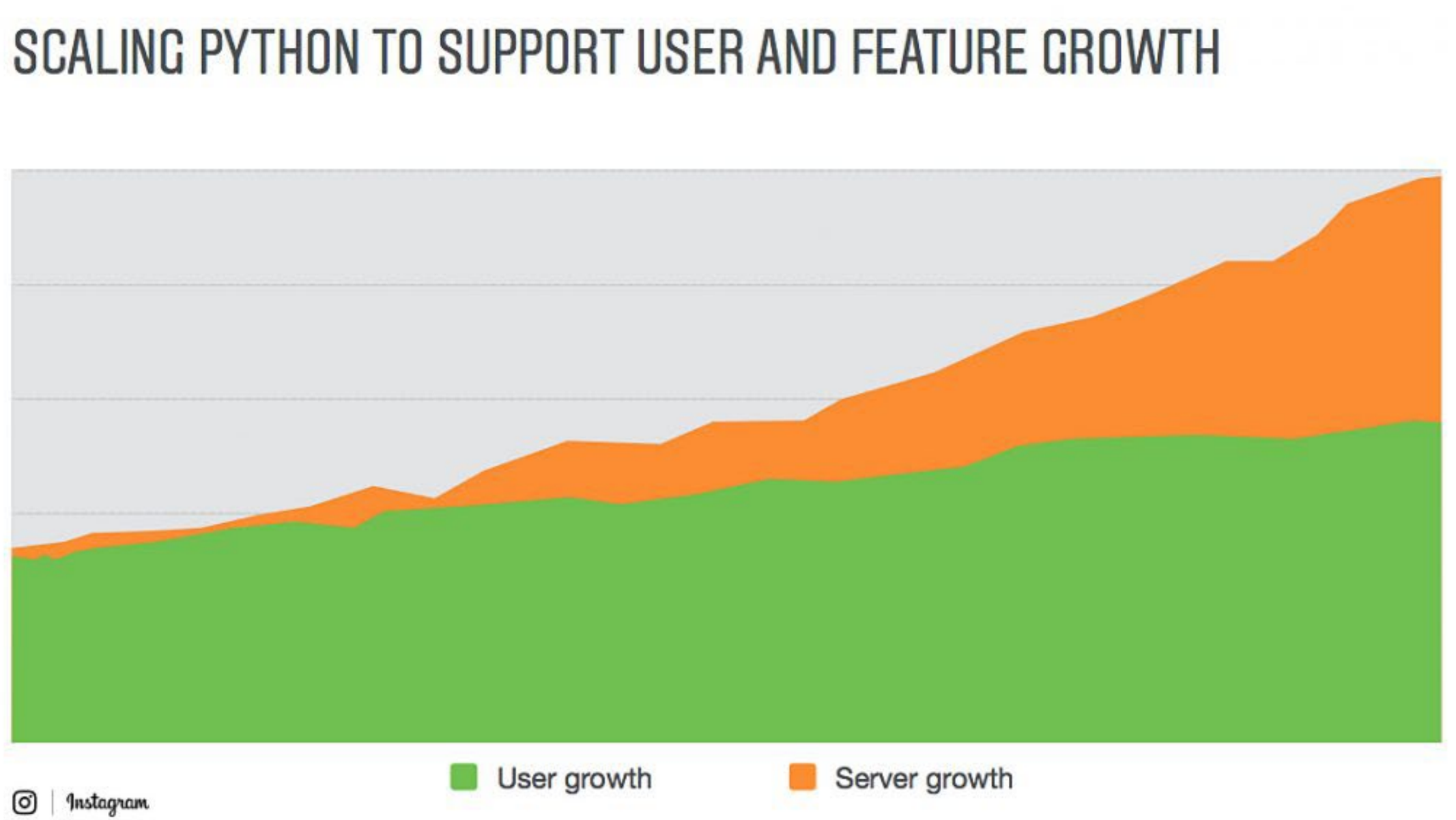

https://www.fastcompany.com/3028778/why-facebook-invented-a-new-php-derived-language-called-hack

# Architectural Technical Debt: Facebook - Hack

- Hack added new safety features.
- It uses automatic type inference (Traditional PHP is dynamically typed)
- It lets programmers specify the types of some variables in their code and uses logic to infer the rest based on how variables are used together, issuing an error if the code's logically inconsistent.
- When a file has changed, the two versions are compared to deduce what must be rechecked at a very fine-grained level: at the method level, not at the file level
- "*Hack enables us to dynamically convert our code one file at a time*" - Facebook Technical Lead HipHop VM (HHVM)

Facebook's Runtime Engine supports PHP and Hack. https://hhvm.com/

# 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

# Case Study: Instagram (Python 2 to Python 3)


- Migration was done over 10 months, all changes merges to Main branch
- Working Rule: *No Python 3, no new package*

Examples of **Refactoring**:

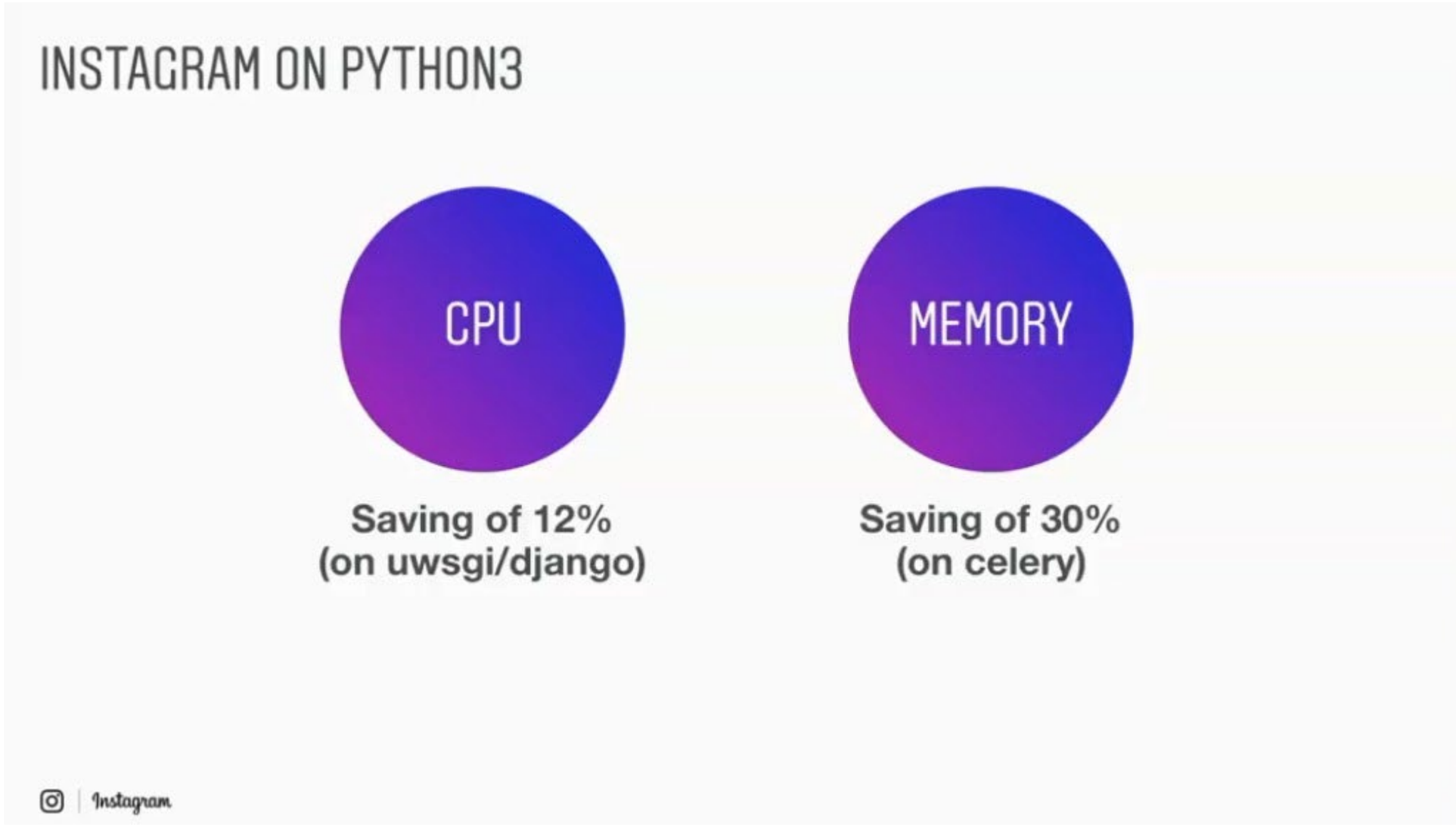
- Differences in **unicode**, **str**, **bytes**. Solved by using helper functions
- Iterator differences, such as map. Solved by converting all maps to list in Python 3
- Dictionary order is different in different Python versions, which caused differences in the dumped JSON data. Solved by forcing sorted_keys in json.dump function

PyCon 2017 Keynote Talk: https://www.youtube.com/watch?v=66XoCk79kjM

http://euccas.github.io/blog/20170616/how-instagram-moved-to-python-3.html

# Case Study: Instagram (Python 2 to Python 3)

- Feb 2017: Completely dropped Python 2

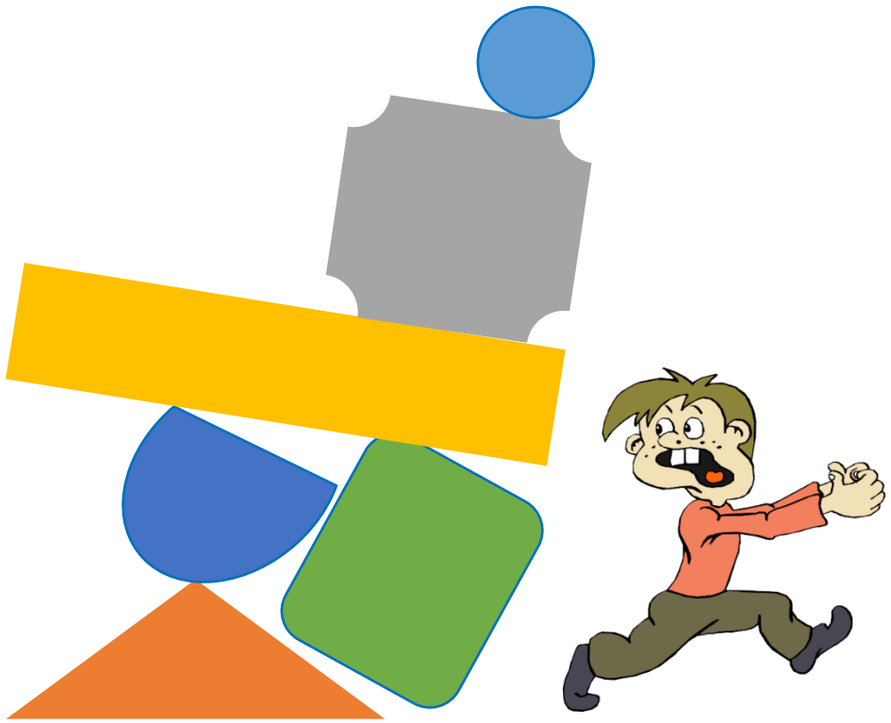



PyCon 2017 Keynote Talk: https://www.youtube.com/watch?v=66XoCk79kjM

http://euccas.github.io/blog/20170616/how-instagram-moved-to-python-3.html

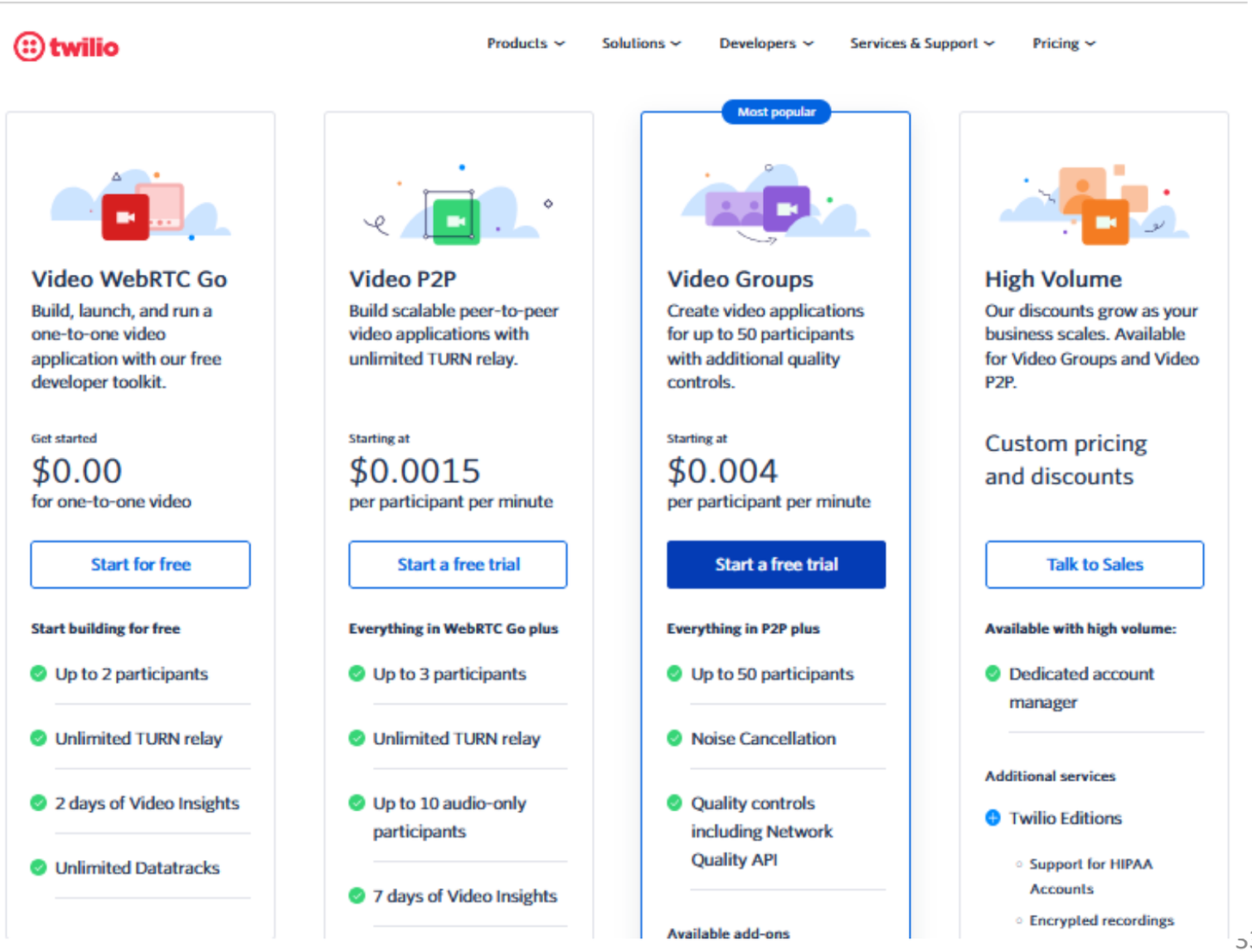
# Retire Technical Debt at Leisure

- 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.



# Twilio Programmable Video vs Amazon Chime Video conferencing service

- What if we need more than 50 people in a town?
- **Discuss** strategies for determining if/when/how to migrate to Amazon's Chime Video service?



# 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.