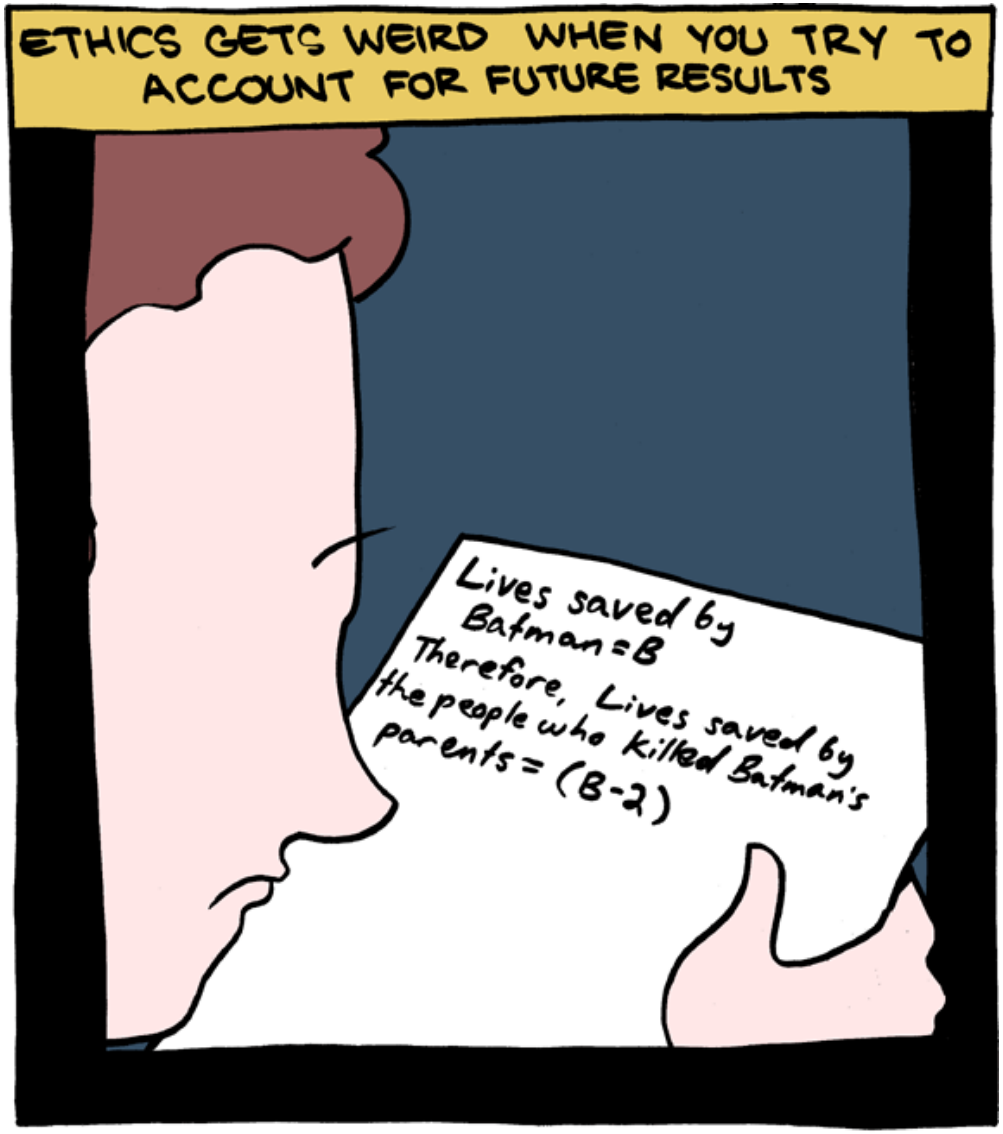
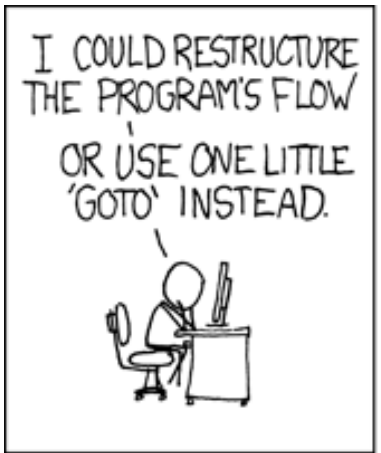
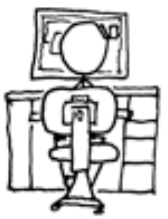
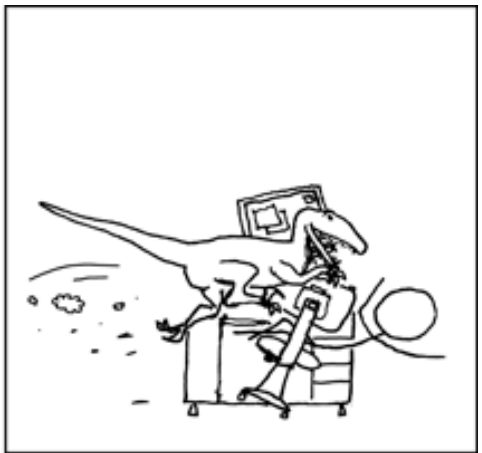
Design for Maintainability









The Story So Far …

- We want to deliver and support a quality software product
  - We understand the stakeholder requirements
  - We understand process and design
  - We understand quality assurance
- How should we make process and design decisions the first time …
- … if software maintenance will be the dominant activity?

One-Slide Summary

- We can invest up-front effort in **designing** software to facilitate **maintenance** activities. This reduces overall lifecycle costs.
- We will consider designing to improve comprehension, documentation, change, reuse, and testability.
  - The *metrics* used for understandability, the category of information conveyed by documentation, object-oriented principles and design patterns, and coverage are all relevant.

Analogy

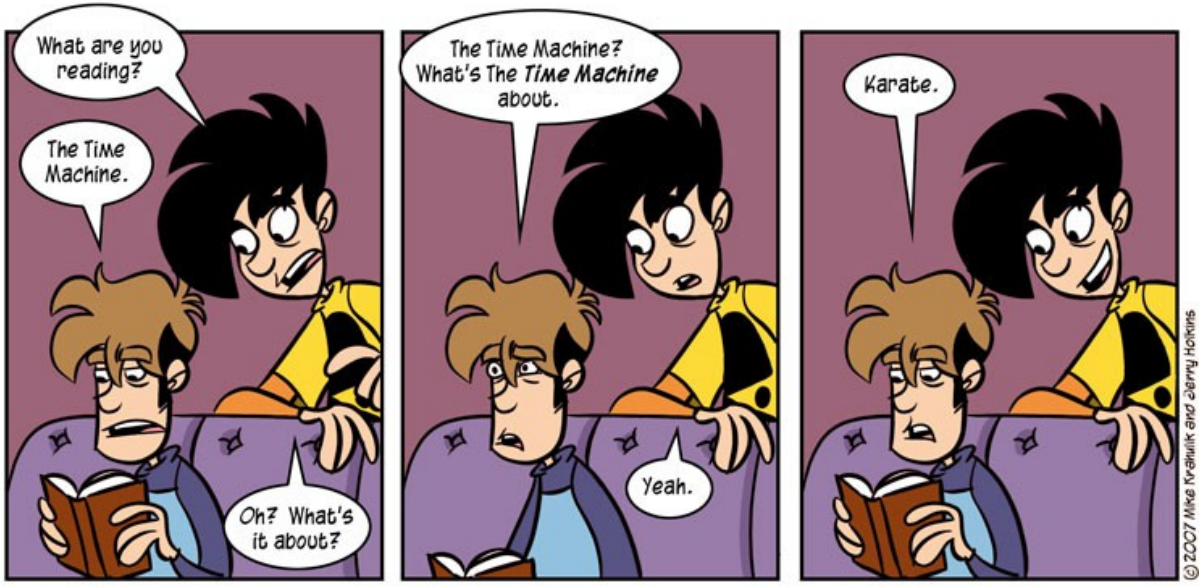
- You are playing “Civilization”
- You want to *quickly* build the Hagia Sophia
- Do you just build it now (costs 3000 production)?
- Or do you build the Forge first (costs 100 production, but then increases your production by +10%)?

Investment

- “It depends on the state of the world.”
- This is just a math problem: is T1 > T2 ?
  - T1 = 3000/production
  - T2 = (100/production) + (3000/(production*1.1))
- “To **invest** is to allocate money (or sometimes another resource, such as time) in the expectation of some benefit in the future”
- You almost always want to invest time during design to produce maintainable software!

Investment in Maintenance

- Suppose maintenance is 70% of the lifetime cost of software and the other 30% is coding and design
- Would you spend 50% more on design if that reduced the cost of maintenance by 50%?

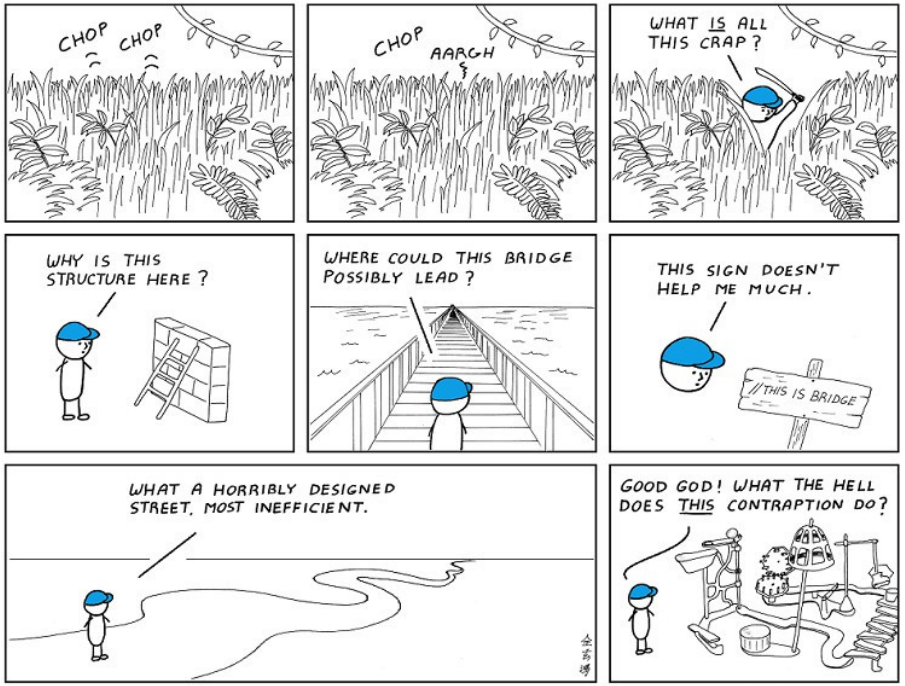


Investment in Maintenance

- Suppose maintenance is 70% of the lifetime cost of software and the other 30% is coding and design
- Would you spend 50% more on design if that reduced the cost of maintenance by 50%?
  - Cost 1 = 30 + 70
  - Cost 2 = 30*1.5 + 70*0.5
- We know the 70% number (indeed: 70-90%)
- But *can we* spend more on design to reduce maintenance costs? Yes.

Design for Maintainability

- High level plan:
- We now understand key maintenance tasks (e.g., testing, code review, etc.)
- So we should design our software to make those activities easier or more efficient
- Even if that means that coding will take longer



I hate reading other people's code.

Pride

- The first thing to change is ***you***
  - Because you likely still think of yourself as a coder
- Student coder goals: quickly produce throwaway software that runs efficiently and solves a well-specified, set-in-stone task
  - You feel good if it doesn't take you long, etc.
- You have to change your internal notion of a “good job”
  - You feel good for readable, elegant code, etc.

Design for Code Comprehension

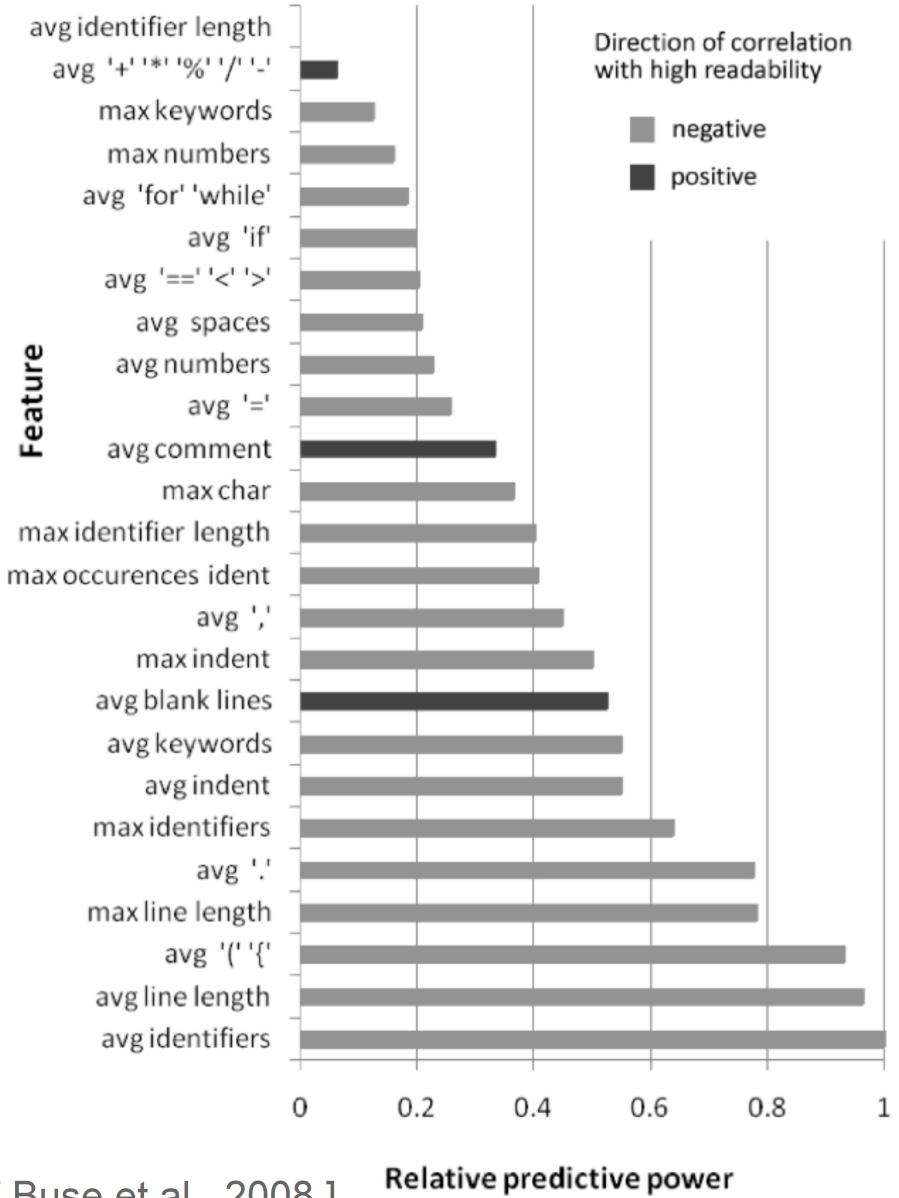
- Code Inspection and Code Review are critical maintenance activities
- We consider improving readability and documentation to aid code comprehension
- We distinguish between **essential** complexity, which follows from the problem statement
  - e.g., sorting requires N log(N) time
- and **accidental** readability, which can be more directly controlled by software engineers

Readability

- **Readability** is a human judgment of how easy a text is to understand
- Commonly desired and mandated in software
  - DOD MIL-M-38784B requires “10th grade reading level or easier”
- So how can we improve code readability?
  - It seems subjective
- Plan: ask many humans, model their average notion of readability, relate to code features
  - Use measurement plus machine learning

Learning a Metric for Code Readability

- Avoid long lines
- Avoid having many different identifiers (variables) in the same region of code
- Do include comments
- Fully blank lines may matter more than indention



[Buse et al., 2008]

Descriptive vs. Prescriptive

- **Descriptive** modeling is a mathematical process that describes [current] real-world events and the relationships between factors *correlated* with them
- A **prescriptive** (or **normative**) model evaluates alternative solutions to answer the question "What is going on?" and suggests what *ought* to be done or how things *should* work [in the future] according to an assumption or standard

Revenge of Perverse Incentive

- We can apply readability metrics automatically to code
- But because they are descriptive, this can lead to **perverse incentives**
- It may be true that existing code with a few more blank lines is more readable
- So what if we just insert a blank line between every line of code?
  - That would maximize the metric, but …
- So use them, but not blindly

Comments and Documentation

- Appeal from a developer on a mailing list:
  - “Going forward, could I ask you to be more descriptive in your commit messages? Ideally should state what you've changed and also why (unless it's obvious) … I know you're busy and this takes more time, but it will help anyone who looks through the log ...”

“WHY DID I STOP DOCUMENTING MY CODE? NO COMMENT.”

What vs. Why

- We can make a distinction between documentation that summarizes **what** the code does (or what happened in a commit)
  - e.g., “Replaced a warning with an IllegalArgumentException”, “this loop sorts by task priority”, “added an array bounds check”
- And documentation that summarizes **why** the code does that (or the change was made)
  - e.g., “Fixed Bug #14235” or “management is worried about buffer overruns”

High-Quality Comments

- You should focus on adding **why** information to your documentation, comments and commit messages
- Because there is tool and process support for adding or recovering **what** information
  - For example, code inspection may reveal that a loop sorts by task priority but will not reveal that this was done because a customer required it

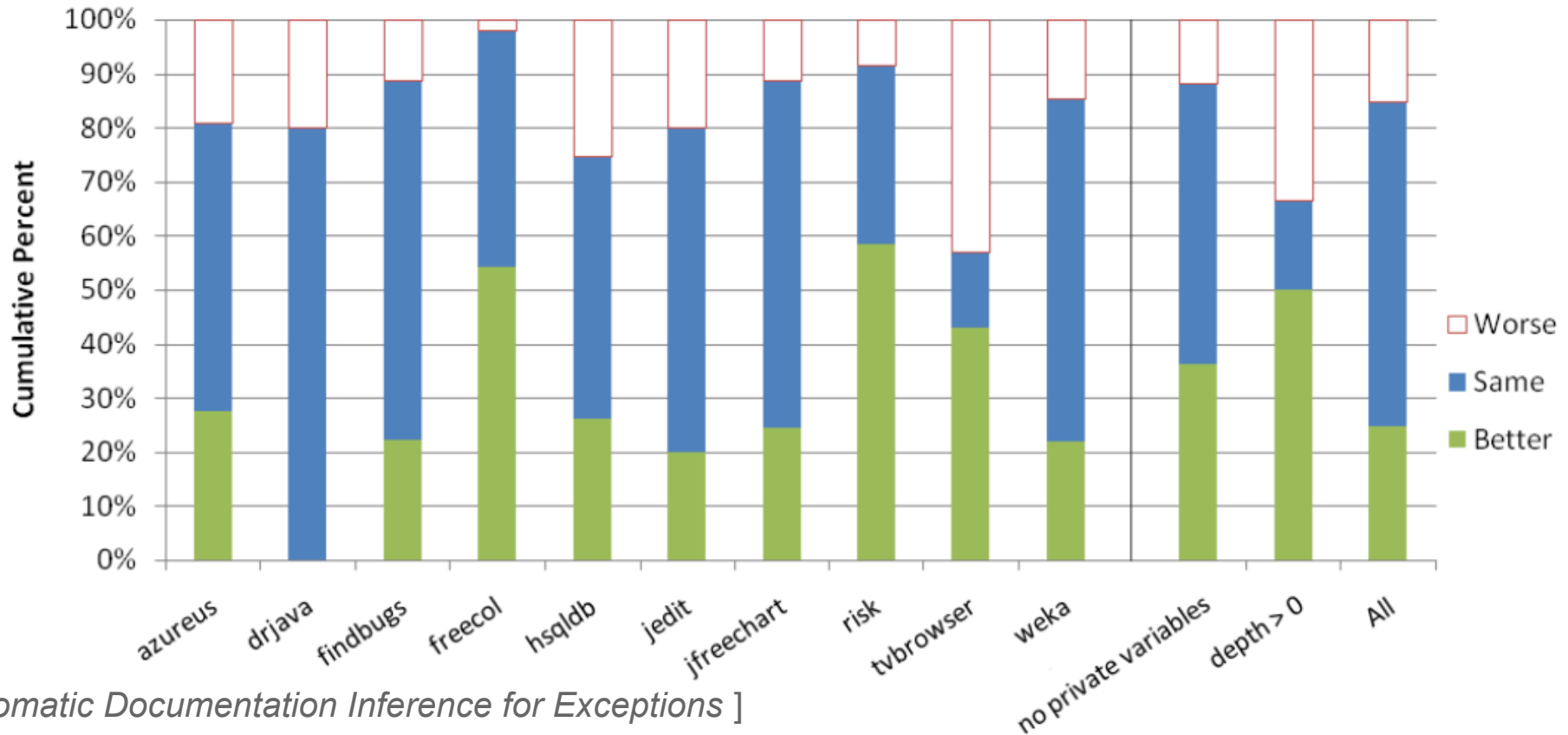
Documenting Exceptions

- Documentation for @throws information, such as **`@exception IllegalArgument if id is null or id.equals("")`** can be automatically inferred via tools
  - Same approach as test input generation
  - Gather constraints to reach the “throw” line
  - Then rewrite them in English
  - Instead of solving them
  - Explains What the code does

```
1  /**
2   * Moves this unit to america.
3   *
4   * @exception IllegalStateException
5   *            If the move is illegal.
6   */
7  public void moveToAmerica() {
8    if (!(getLocation() instanceof Europe)) {
9      throw new IllegalStateException("A unit"
10       + " can only be moved to america from"
11       + " europe.");
12   }
13   setState(TO_AMERICA);
14   // Clear the alreadyOnHighSea flag:
15   alreadyOnHighSea = false;
16 }
```

“Why” for Exceptions

- Tools are at least as accurate as humans 85% of the time, and are better 25% of the time
  - Tools can do *What* – so have humans focus on *Why*

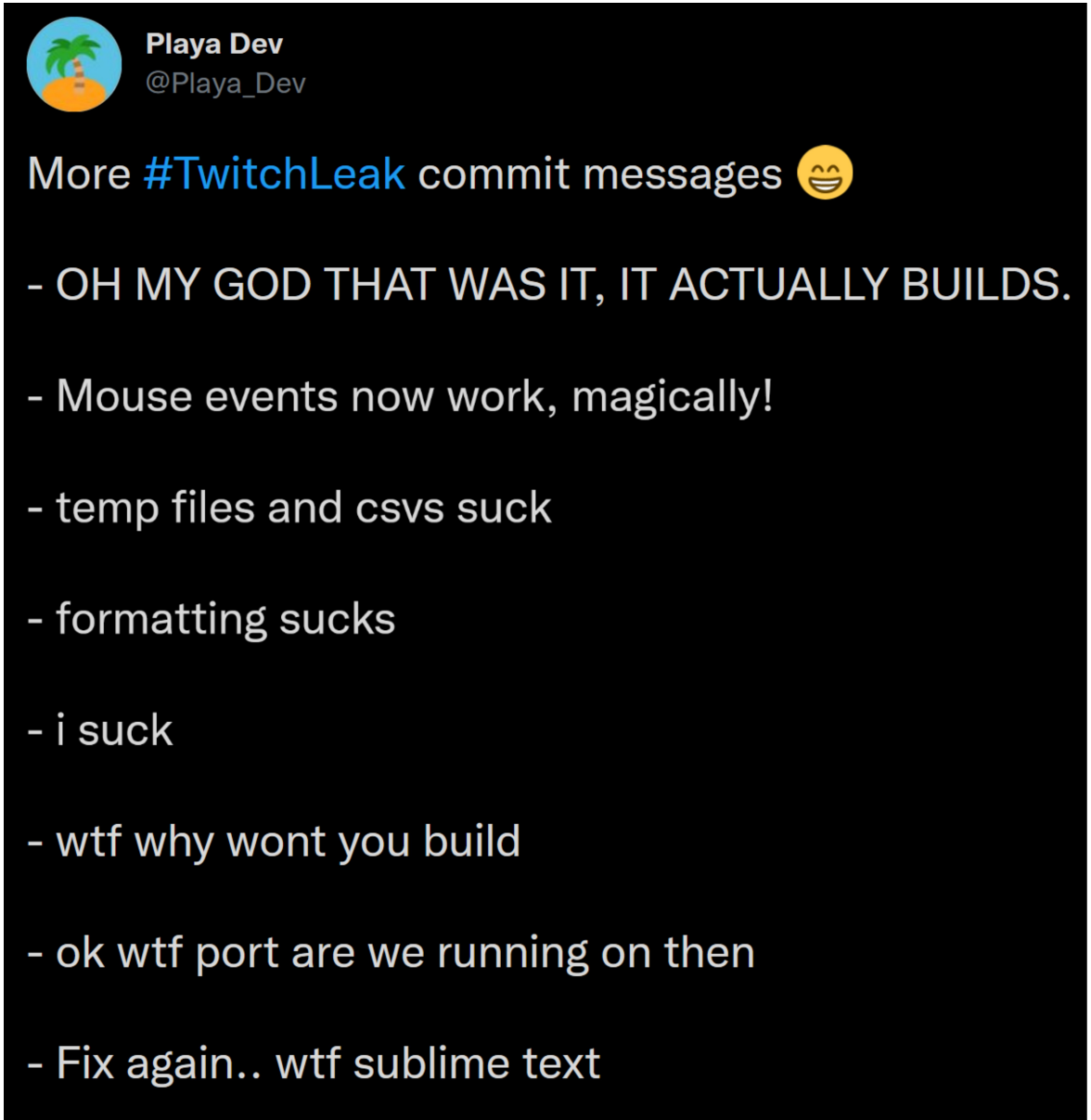


Documenting Commit Messages

- Appeal from a developer:
  - “Sorry to be a pain in the neck about this, but could we please use more descriptive commit messages? I do try to read the commit emails, but... I can't really tell what's going on”
- Example: revision 3909 of **`iText`**'s complete commit message is “**`Changing the producer info`**”

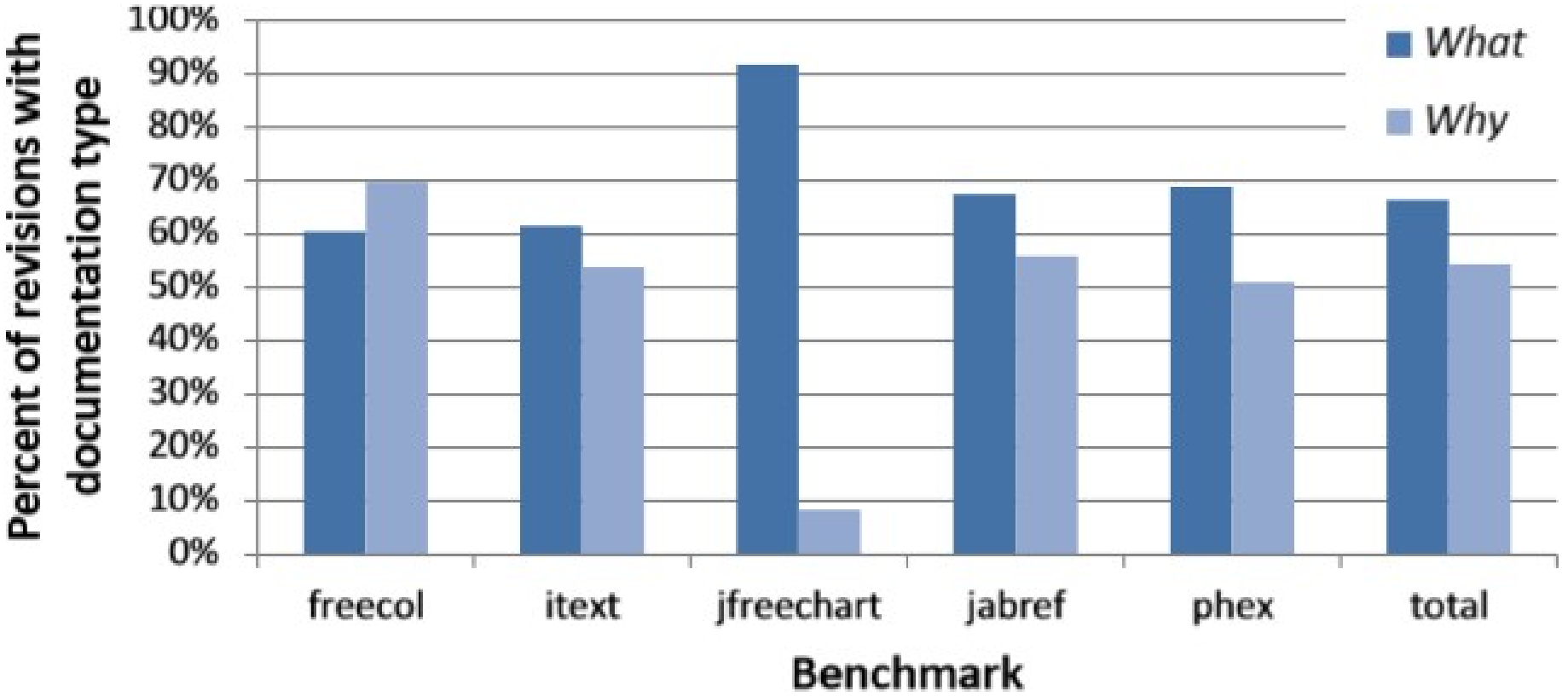
Commit Messages in the Wild (one “case study”)

- October 2021: Amazon's Twitch source code was leaked in a 125 GB data breach
- the entirety of twitch.tv with “with commit history going back to its early beginnings”



Commit Messages in the Wild

- Average size of a non-empty human written log message: 1.1 lines
- Average size of a textual diff: 37.8 lines



“Why” for Commit Messages

- Tools and algorithms have been shown to replace or provide 89% of the What information in log messages
- It is definitely good to describe what a change is doing
- But you should focus on documenting *Why*
- Get in the habit of providing two categories of information for every pull request
  - (And method summary, and …)

Trivia: SCOTUS

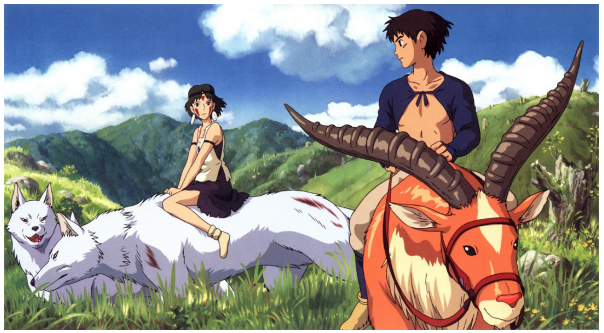
- This associate justice of the Supreme Court was born in the Bronx, went to Princeton and Yale, and was appointed by Obama. She has been associated with concern for the rights of defendants, calls for reform of the criminal justice system, and dissents on issues of race, gender and ethnic identity. For example, in *Schuette vs. CDAA* (a case about a state ban on race- and sex-based discrimination in public university admissions), she dissented that "[a] majority of the Michigan electorate changed the basic rules of the political process in that State in a manner that uniquely disadvantaged racial minorities."

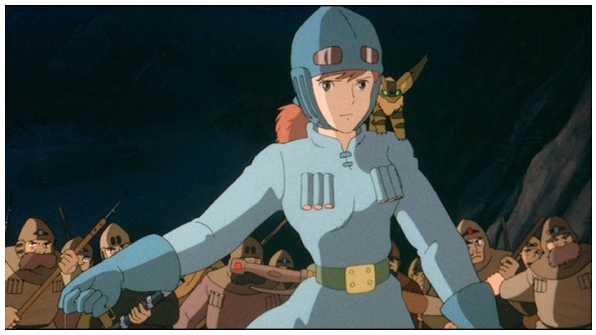
Trivia: SCOTUS 2

- This associate justice of the Supreme Court was born in Brooklyn, went to Cornell and Columbia, and was appointed by Clinton. She has been associated with gender equality and women's rights. She has been characterized for making passionate dissents and a liberal view of the law. Her dissent in *Ledbetter v. Goodyear Tire & Rubber Co.* is credited with leading to the *Lilly Ledbetter Fair Pay Act of 2009* that makes it easier to file equal pay lawsuits. Also: lace jabot collection.

Trivia: Filmmakers

- This Japanese artist was called “the best animation filmmaker in history” by Roger Ebert. He co-founded Studio Ghibli, received international acclaim, and directed films such as *Princess Mononoke* (highest-grossing film in Japan) and *Spirited Away* (also the highest-grossing film in Japan, and an Academy Award winner). He just might like airships.





Trivia: Music

- This single-reed woodwind instrument features a straight tube with a cylindrical bore and a flared bell. It is believed to date back to the year 1700 in Germany. It is commonly used in classical, military, marching, klezmer and jazz bands. Modern orchestras use soprano versions of this instrument in B♭ and A. Benny Goodman helped popularize its use in big bands for swing. The Beatles song *When I'm Sixty-Four* features a trio of these.

Psychology: Bridges?

- 85 single males, aged 18-35, walked over either a 450-long, 5-foot wide suspension bridge made of wooden boards and wire cables over the Capilano Canyon, or a solid wood bridge upriver.
  - Similar males rated the bridge a 79 out of 100 on "How fearful ..."

Psychology: Bridges

- After crossing either the control or experimental bridge, subjects were approached by a male or female interviewer
  - “She explained that she was doing a project for her psychology class on the effects of exposure to scenic attractions on creative expression. She then asked potential subjects if they would fill out a short questionnaire” and then write a story based on a neutral picture.
- Upon completion she thanked them and then tore off a corner of a sheet of paper and wrote down her name and phone number, inviting each subject to call if he wanted to talk further.
  - The control group was told her name was Donna and the experimental group was told her name was Gloria …

Psychology: Misattribution of Arousal

- 23/33 filled out the questionnaire on the experimental bridge, 22/33 on the control bridge
- The stories were scored for sexual imagery using TAT scoring
  - Experimental group: 2.47 for sexual imagery vs. 1.41 in the control group ($p < 0.01$)
- In the experiment group, 50% of them called her, while in the control group, only 12.5% did so ($p < 0.02$)

[Dutton and Aron. *Some evidence for heightened sexual attraction under conditions of high anxiety*. J. Personal and Social Psychology. 1974.]

Psychology: Misattribution of Arousal

- The **misattribution of arousal** is a process whereby people unconsciously mistake physiological symptoms (e.g., blood pressure, shortness of breath: symptoms of fear) with arousal. This includes perceiving a partner as more attractive because of a heightened state of stress.
- Later studies found that **confidence** can also be affected by misattribution of arousal. Participants were asked to complete a task with a noise in the background; some were told the noise might make them nervous, others were told it would have no effect or that there was a deadline: "which resulted in those participants [who attributed their arousal to external noise] feeling more confident that they did well on the tasks than those that attributed their arousal to the performance anxiety from the task". ("We used SE process XYZ during the last stressful push; coincidentally, I think I like XYZ ...")

Design for Change and Reuse

- In class, many programs are written once, to a fixed specification, and thrown away
- In industry, many programs are written once and then modified as requirements, customers, and developers change
- Many fundamental tenets of object-oriented design facilitate subsequent change
  - You've seen these before, but now you are in a position to really appreciate the motivation!

Design Desiderata

- Classes are open for extension and modification without invasive changes
- Subtype polymorphism enables changes behind interfaces
- Classes encapsulate details likely to change behind (small) stable interfaces
- Internal parts can be developed independently
- Internal details of other classes do not need to be understood, contract is sufficient
- Class implementations and their contracts can be tested separately (unit testing)

Design for Reuse: Delegation

- **Delegation** is when one object relies on another object for some subset of its functionality
  - e.g., in Java, Sort delegates functionality to some Comparator
- Judicious delegation enables code reuse
  - Sort can be reused with arbitrary sort orders
  - Comparators can be reused with arbitrary client code that needs to compare integers
  - Reduce “cut and paste” code and defects

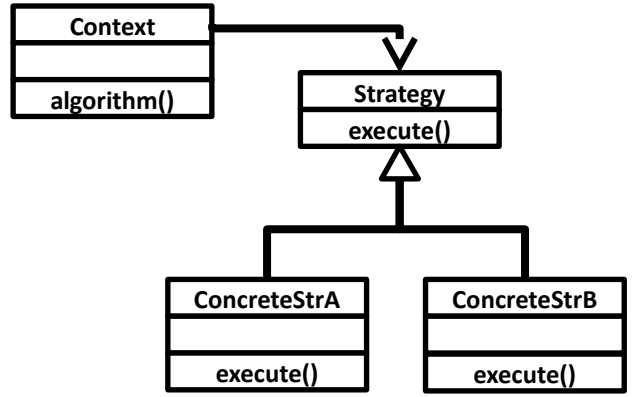
Design for Change: Motivation

- Amazon.com processes millions of orders each year, selling in 75 countries, all 50 states, and thousands of cities worldwide. These countries, states, and cities have hundreds of distinct sales tax policies and, for any order and destination, Amazon.com must be able to compute the correct sales tax for the order and destination. Over time:
  - Amazon moves into new markets
  - Laws and taxes in existing markets change

Software Design Patterns

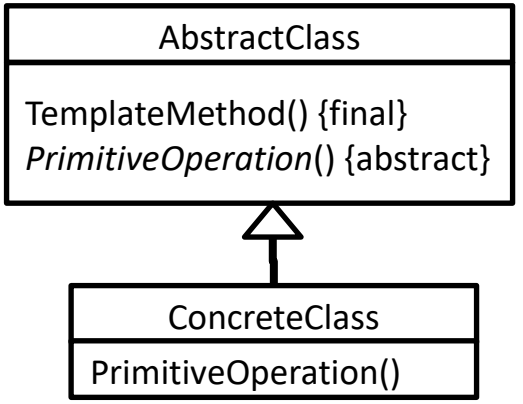
- A software **design pattern** is a general, reusable solution to a commonly-occurring problem within a given context in software design.
  - (Other lectures have more details.)

Strategy Design Pattern



- Problem: Clients need different variants of an algorithm
- Solution: Create an interface for the algorithm, with an implementing class for each variant of the algorithm
- Consequences:
  - Easily extensible for new algorithm implementations
  - Separates algorithm from client context
  - Introduces extra interfaces and classes: code can be harder to understand; adds overhead if the strategies are simple

Template Method Design Pattern



- Problem: An algorithm has customizable and invariant parts
- Solution: Implement the invariant parts of the algorithm in an abstract class, with abstract (unimplemented) primitive operations representing the customizable parts of the algorithm. Subclasses customize the primitive operations.
- Consequences
  - Code reuse for the invariant parts of algorithm
  - Customization is restricted to the primitive operations
- Inverted ("Hollywood-style") control for customization: "don't call us, we'll call you" (cf. comparison function in sorting)
- Invariant parts of the algorithm are not changed by subclasses

Template Method vs. Strategy

- Both support variation in a larger context
- Template method uses inheritance + an overridable method
- Strategy uses an interface and polymorphism (via composition)
  - Strategy objects are reusable across multiple classes
  - Multiple strategy objects are possible per class

Design for Extensibility: Contracts and Subtyping

- **Design by contract** prescribes that software designers should define formal, precise and verifiable interface specifications for components, which extend the ordinary definition of abstract data types with **preconditions**, **postconditions** and invariants
- A subclass can only have ***weaker*** preconditions
  - My super only works on positive numbers, but I work on all numbers
- A subclass can only have ***stronger*** postconditions
  - My super returns any shape, but I return squares
- This is just the Liskov Substitution Principle!

Design for Testability

- If the majority cost of software engineering is maintenance, and the majority cost of maintenance is QA, and the majority cost of QA is testing
- It behooves us to design our software so that **testing** is effective
  - Design to admit testing
  - Design to admit fault injection
  - Design to admit coverage
  - Recognize “free test” opportunities

Design to Admit Testing

- Consider a **library oriented architecture**, a variation of modular programming or service-oriented architecture with a focus on separation of concerns and interface design
  - "Package logical components of your application independently - literally as separate gems, eggs, RPMs, or whatever - and maintain them as internal open-source projects … This approach combats the tightly-coupled spaghetti so often lurking in big codebases by giving everything the Right Place in which to exist."

Unit Testing

- Recall: it is hard to generate test inputs with high coverage for areas “deep inside” the code
  - Must solve the constraints for main(), then for foo(), then for bar(), etc., all at the same time!
- The farther code is from an entry point, the harder it is to test
  - This is one of the motivations behind Unit Testing
- Solution: design with more entry points for self-contained functionality (cf. AVL tree, priority queue, etc.)

Example: Model View Controller

- Suppose you are designing Angry Birds
- It's a game, and also a simulation, so MVC is a reasonable choice
- Design so that it can be tested without someone actually playing the game!
  - e.g., have an **interface** where abstract commands can be queued up: one way to get them is from the UI, but another is programmatic
  - “If I create a world with blocks X, Y and Z and then we launch bird A at angle B, does C occur within five timesteps?”

Fault Injection

- Microsoft's Driver Verifier sat between a driver and the operating system and “pretended to fail (some of the time)” to expose poor driver code
- The CHESS project sat between a program and the scheduler and “forced strange schedules” to expose poor concurrency code
- Hardware, OS and Networking errors can occur infrequently, but you still want to test them
  - Must design for it!

Level Of Indirection

- Old adage: the solution to everything in computer science is either to add a level of indirection or to add a cache
- Don't have your code call fopen() or cout or whatever directly
- Instead, add a very thin **level of indirection** where you call my_fopen which then calls fopen
- Later add “if coin_flip() then fail else ...” to that indirection layer to **inject faults**

Designing for Coverage-Based Testing

- Code coverage has many flaws
  - At a high level, simple coverage metrics do not align with covering requirements (cf. traceability)
- Solutions
  - Better test suite adequacy metrics (mutation, etc.)
  - Design and write the code so that high code coverage correlates with high requirements coverage!

Recall: Implicit Control Flow

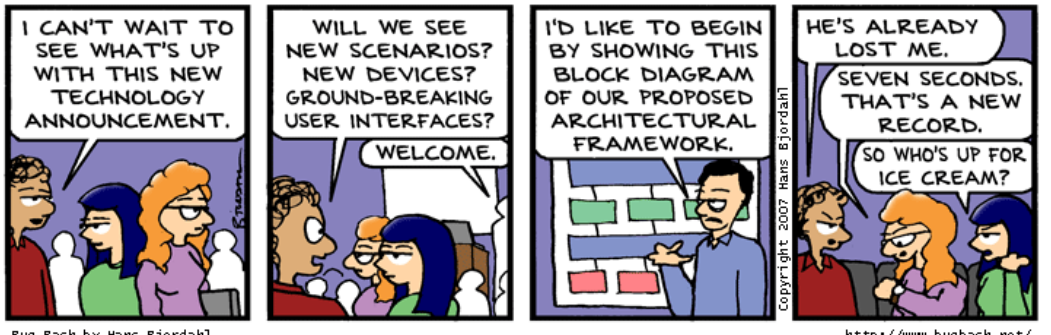
- Line coverage was often inadequate because "visit line 5 when ptr==null" could be very different from "visit line 5 when ptr !=null"
  - Because "*ptr = 9" is really "if (ptr == null) abort(); else *ptr = 9;"
- Consider explicit conditionals that check requirements adherence
  - To get coverage points for reaching the true branch, the test will have to satisfy the requirement

Requirement Coverage

- Quality requirement: “finish X within Y time”
  - Add in “get the time”, “do X”, “get the time”, “subtract”, “if t2 – t1 < Y then ...”
- You could also encode these in test oracles
- Explicit Conditional Pros
  - Testing tools can help you reason about partial progress
  - Testing tools can try to falsify claims
- Explicit Conditional Cons
  - Muddies meaning of coverage (100% not desired)

Tests for Free

- Many programs transform data from one format to another (cf. adapter pattern)
- If the program is implementing a function with similar domain and range, you can often get high-coverage tests “for free” by composing the program with itself
  - If possible, design your program so that this is possible (cf. as a library)



Examples

- **Inversion**
  - Forall X. unzip(zip(x)) = x
  - Forall X. deserialize(serialize(x)) = x
  - Forall X. decrypt(encrypt(x)) = x
- **Convergence**
  - Forall X. indent(indent(x)) = indent(x)
  - Forall X. stable_sort(stable_sort(x)) = stable_sort(x)
  - Forall P1. Forall I. If P2 = compile(decompile(compile(P1))) then P1(I)=P2(I)
    - mp3enc/mp3dec, jpg2png/png2jpg,

Note: you may need a non-exact **comparator**!

Hints for Practice

- Find 5 commit messages and 5 comments on github and try to write “Why” documentation for them
- Write an Eiffel program that uses pre- and post-conditions and inheritance
- How would you design the Autograder to support fault injection?
- How would you design mutate.py as a library that takes a list of edit operations? When should mutate(p,[e1,e2]) = mutate(p,[e2,e1])?

Questions?

- HW5!