MICHIGAN ENGINEERING
UNIVERSITY OF MICHIGAN

Design for Maintainability

EECS 481 (W24)

Software Development Life Cycle (SDLC)

- Requirement Analysis: Understanding what is needed.
- Planning: Determining the resources and schedule.
- Design: Architecting the software.
- Development: Writing the actual code.
- Testing: Ensuring the software works as intended.
- Deployment: Releasing the software to users.
- Maintenance: Updating and fixing the software as needed.

The Story so far…

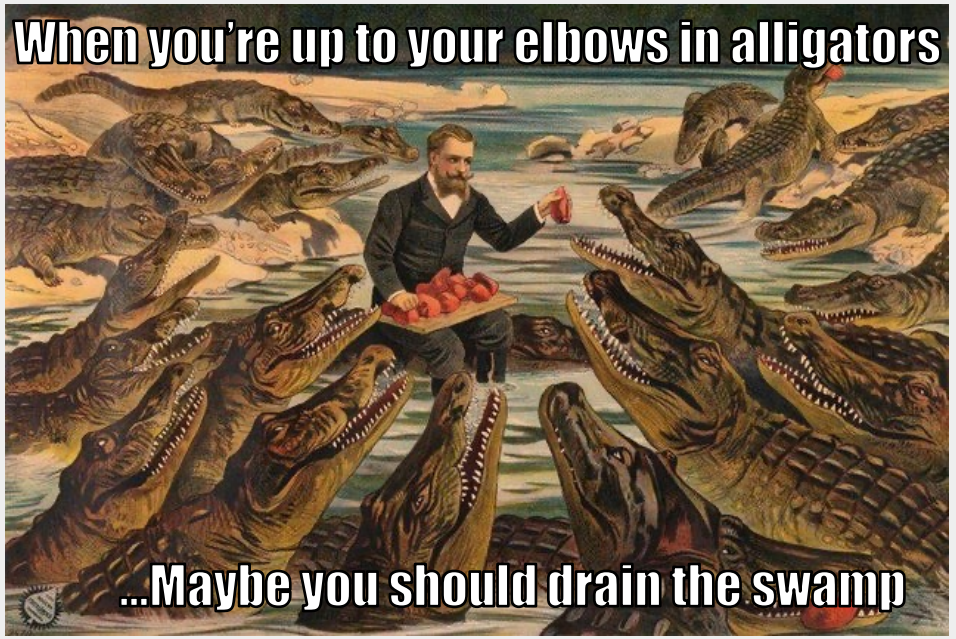
- We want to deliver and support a quality software product
  - We understand the stakeholder requirements
  - We understand process and design
  - We understand quality assurance
- How should we make process and design decisions the first time …
- … if software maintenance will be the dominant activity?

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.

Outline (the emotional journey)

- Underlying Principles for Designing for Maintainability
- D for
  - Reading
  - Change
  - Testing



MICHIGAN ENGINEERING
UNIVERSITY OF MICHIGAN

Learning Objectives: by the end of today's lecture you should be able to…

1. (*value*) believe that spending more time up front on designing for maintainability will save you time
2. (*knowledge*) provide one example of how to improve your comments and commit messages
3. (*knowledge*) give a definition of a design pattern
4. (*knowledge*) suggest a couple ways part of a program can be designed to facilitate testing

Motivation and Premise

Analogy

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

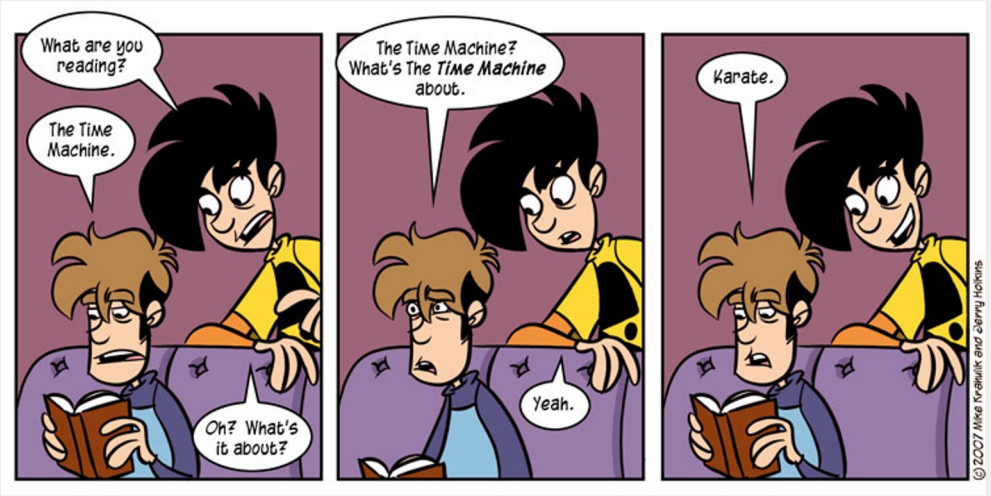
https://en.wikipedia.org/wiki/Hagia_Sophia

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

- Design for maintainability is a software engineering principle that aims to make software products easier to modify, update, and enhance over time.
- It involves applying good practices such as modularity, cohesion, coupling, documentation, coding standards, and testing to the software development process.
- Designing for maintainability can reduce the cost and effort of software maintenance, which typically accounts for a large portion of the software lifecycle.

Some Benefits of Design for Maintainability

- Improved software quality and reliability
- Increased customer satisfaction and loyalty
- Reduced technical debt and rework
- Enhanced software reuse and adaptation
- Faster delivery and deployment of software changes

Some Challenges of Design for Maintainability

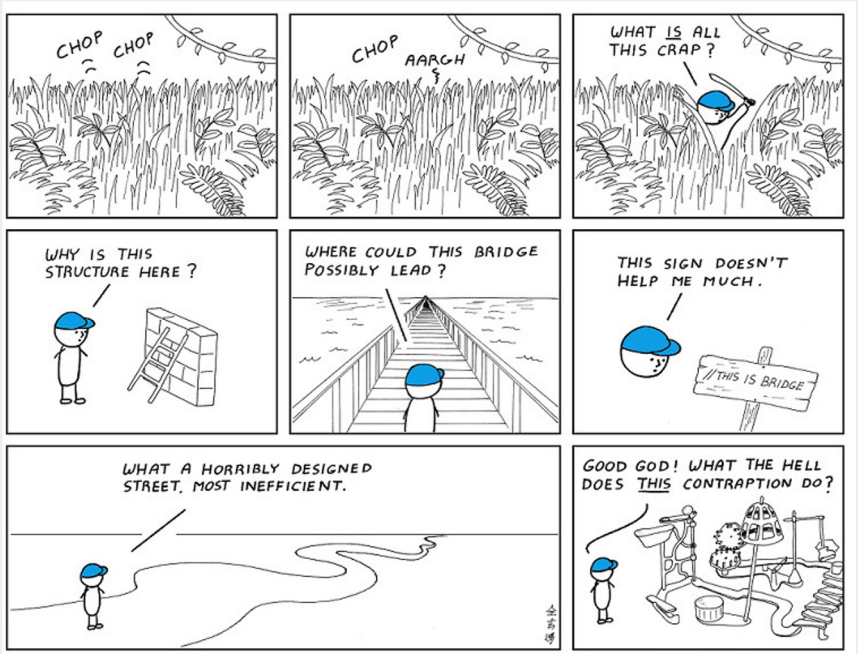
- Balancing trade-offs between functionality, performance, and maintainability
- Anticipating future requirements and changes
- Managing complexity and dependencies
- Communicating and collaborating with stakeholders
- Measuring and evaluating maintainability

https://www.sealights.io/software-quality/software-maintainability-what-it-means-to-build-maintainable-software/

https://extapps.ksc.nasa.gov/Reliability/Documents/Preferred_Practices/dfe6.pdf

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



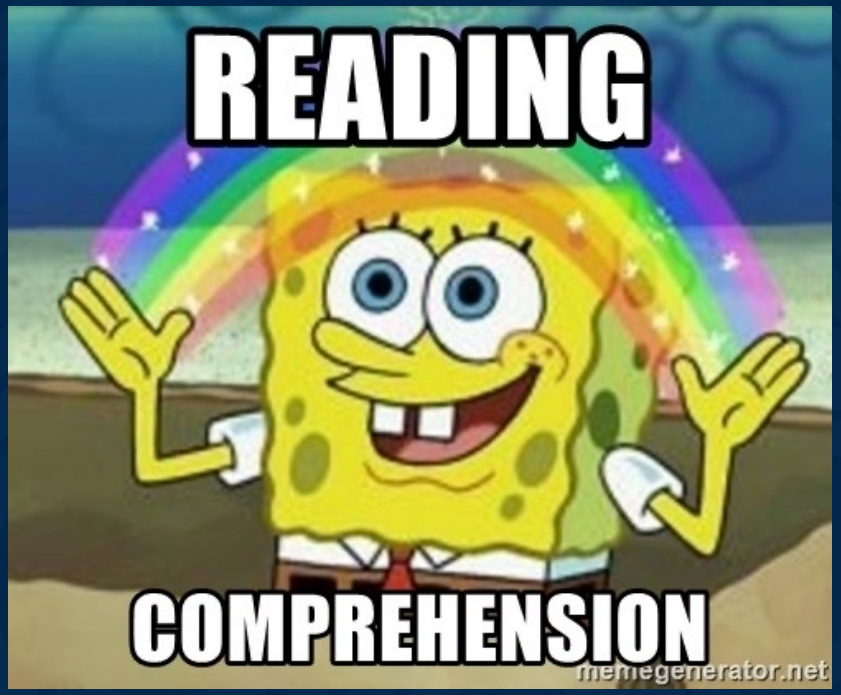
I hate reading other people's code.

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.

MICHIGAN ENGINEERING
UNIVERSITY OF MICHIGAN

Design for Comprehension



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

Design for Code Comprehension

Design for code comprehension is a software engineering principle that aims to make software products easier to read, understand, and modify by human developers.

It involves applying good practices such as naming conventions, documentation, formatting, modularity, abstraction, and testing to the software development process.

Designing for code comprehension can improve the quality, reliability, and maintainability of software products, as well as reduce the cognitive load and effort of software developers.

Some of the Benefits of Design for Code Comprehension

- Improved software quality and reliability
- Reduced technical debt and rework
- Enhanced software reuse and adaptation
- Faster delivery and deployment of software changes
- Increased collaboration and communication among developers

Some of the Challenges of Design for Code Comprehension

- Balancing trade-offs between readability, functionality, and performance
- Anticipating future requirements and changes
- Managing complexity and dependencies
- Communicating and documenting design decisions
- Measuring and evaluating code comprehension

https://www.lucidchart.com/blog/visualize-code-documentation

https://www.geeksforgeeks.org/comprehensions-in-python/

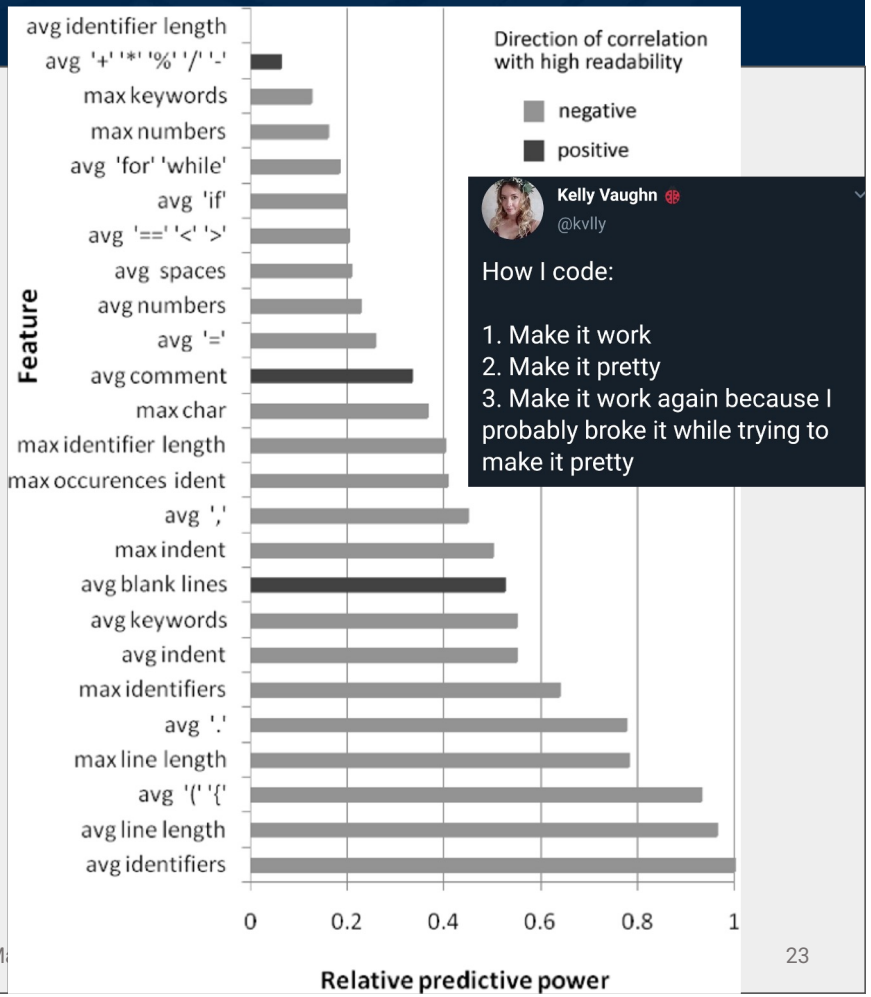
Readability

- **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

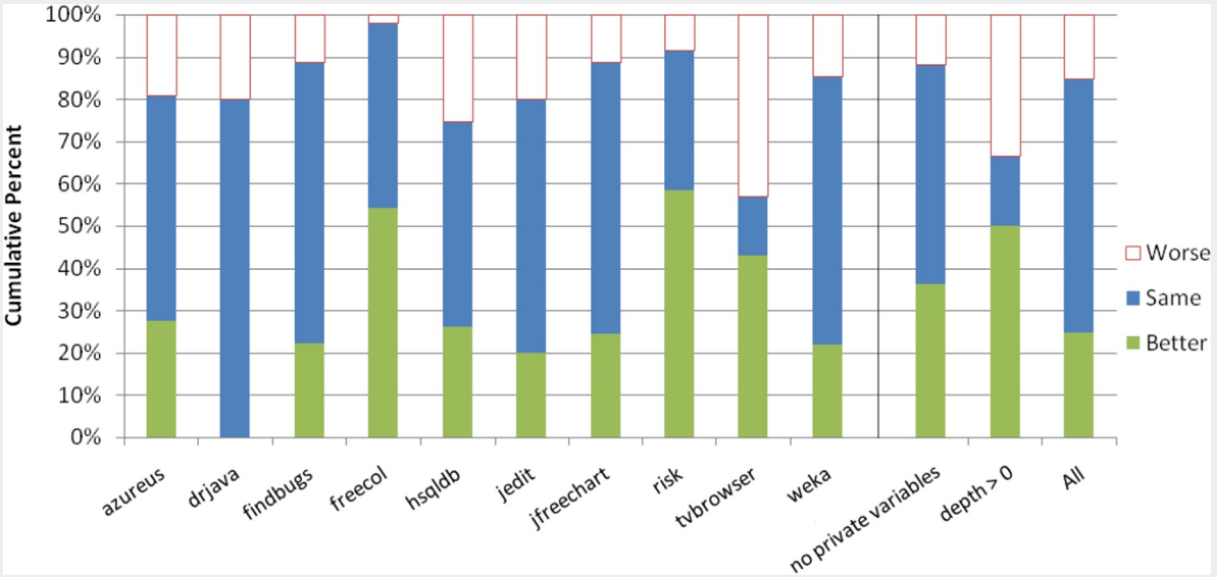
- Documentation for `@throws` information, such as `@exception IllegalArgument if id is null or id.equals("")` can be automatically inferred via tools
  - Same approach as test input generation
  - Gather constraints to reach the "throw" line
  - Then rewrite them in English
  - Instead of solving them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

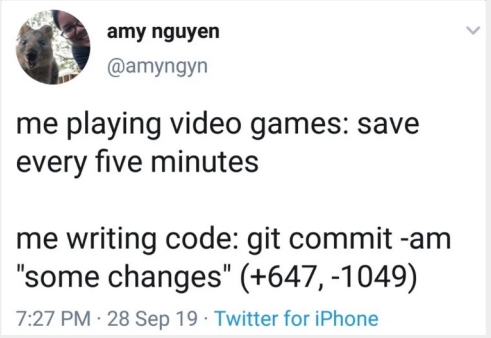
- Tools are at least as accurate as humans 85% of the time, and are better 25% of the time
  - Tools can do What – so have humans focus on Why

[Automatic Documentation Inference for Exceptions]



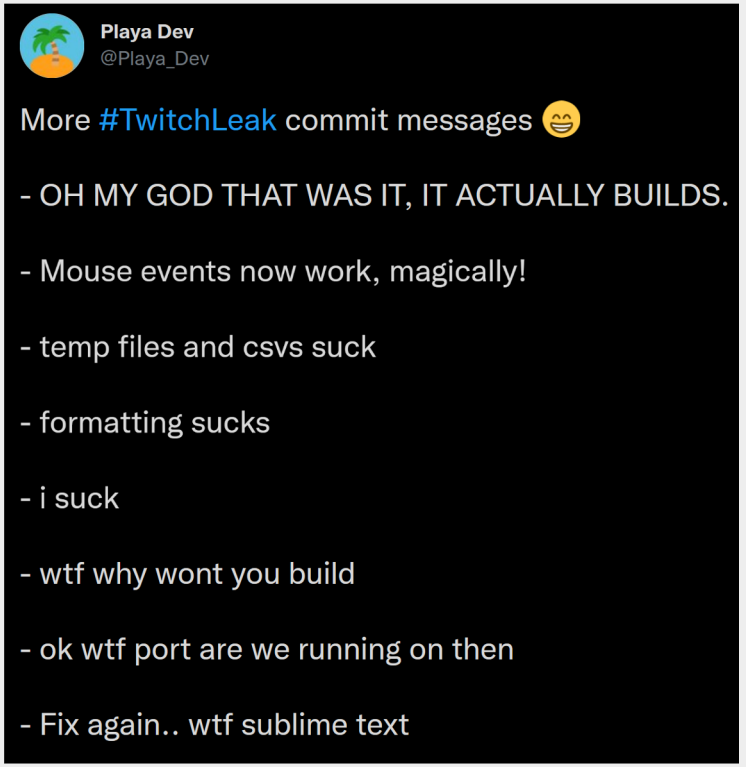
Documenting Commit Messages

- 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`**"



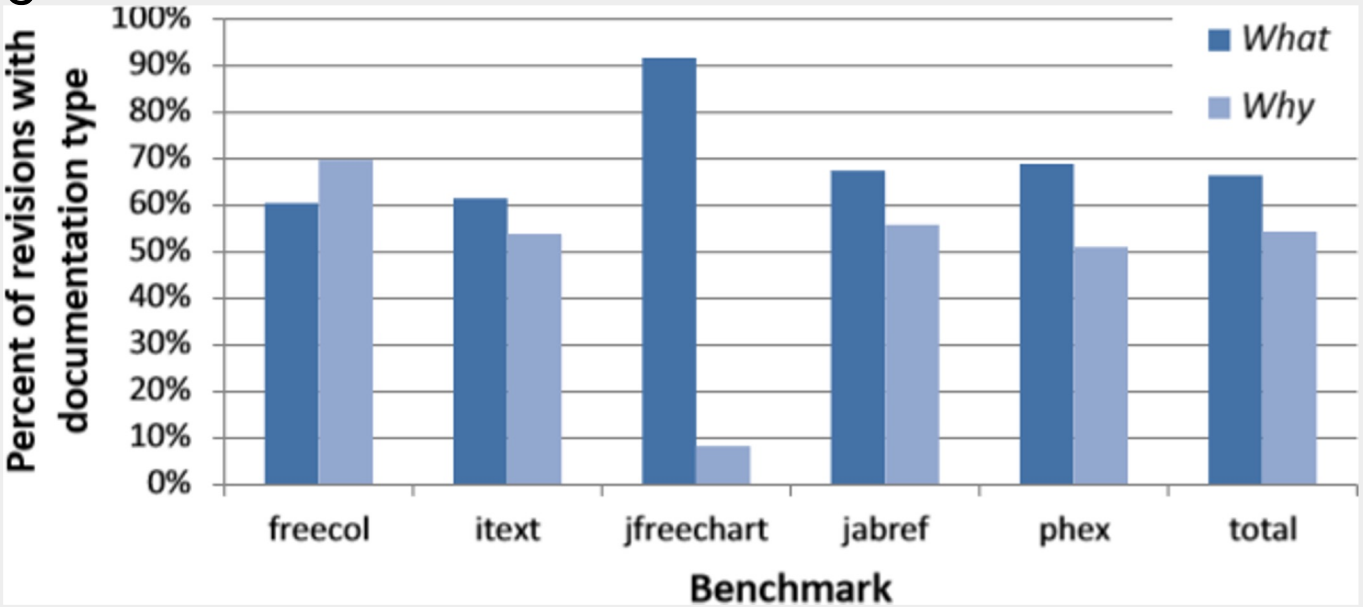
Commit Messages in the Wild (one “case study”)

- 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

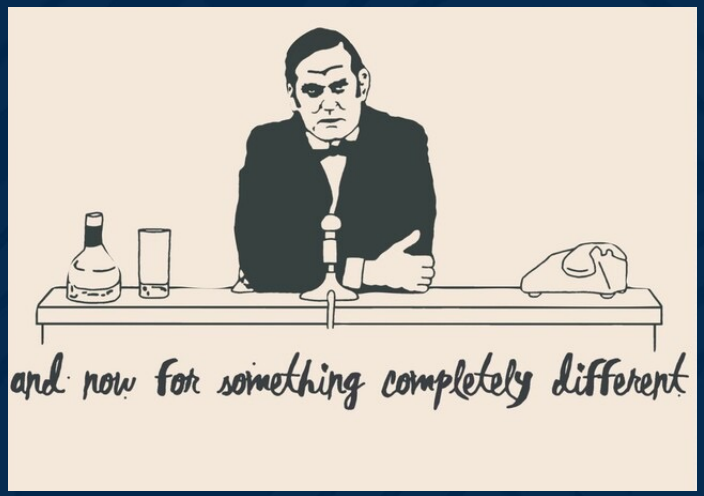


"Why" for Commit Messages

- 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 …)

MICHIGAN ENGINEERING
UNIVERSITY OF MICHIGAN

Trivia Break



MICHIGAN ENGINEERING
UNIVERSITY OF MICHIGAN

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."

MICHIGAN ENGINEERING
UNIVERSITY OF MICHIGAN

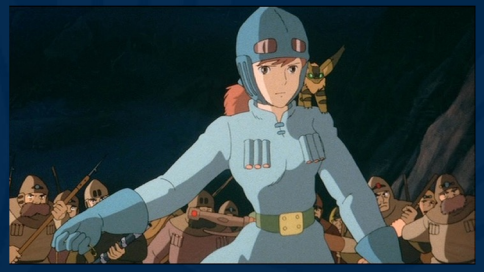
Trivia: SCOTUS 2

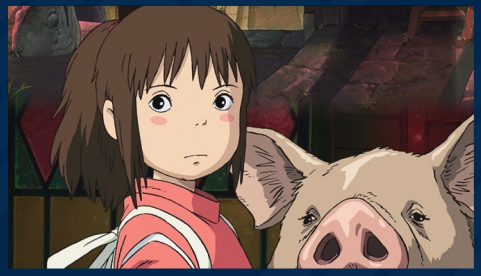
- This associate justice of the Supreme Court was born in Brooklyn, went to Cornell and Columbia, and was appointed by Clinton. She has been associated with gender equality and women's rights. She has been characterized for making passionate dissents and a liberal view of the law. Her dissent in *Ledbetter 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.

MICHIGAN ENGINEERING
UNIVERSITY OF MICHIGAN

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.





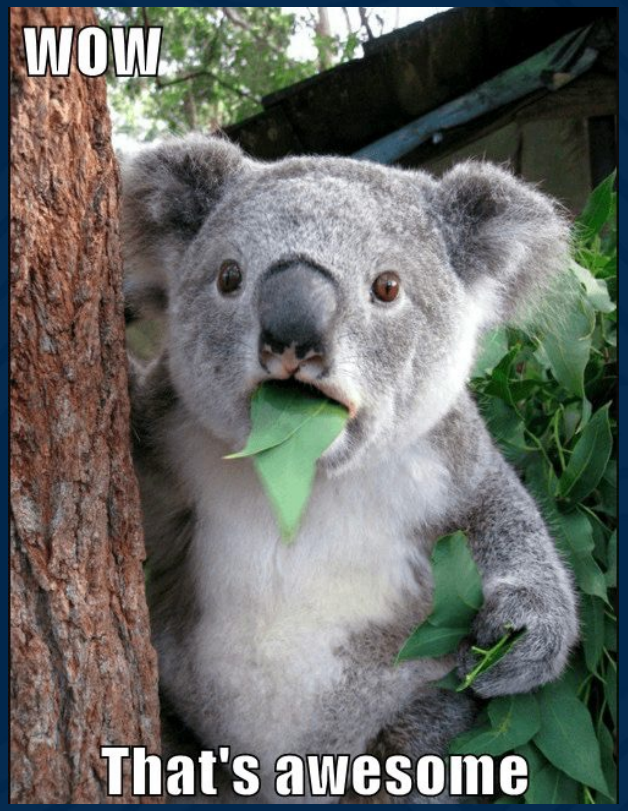
MICHIGAN ENGINEERING
UNIVERSITY OF MICHIGAN

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.

MICHIGAN ENGINEERING
UNIVERSITY OF MICHIGAN

Design for Change and Reuse



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!

The Object-Oriented Design Paradigm

The object-oriented design paradigm in software engineering is a way of designing software systems based on the concept of objects, which are entities that have data and behavior.

Objects can interact with each other by sending messages and can be grouped into classes, which define their common attributes and methods.

Object-oriented design aims to make software more modular, reusable, extensible, and maintainable by following some principles, such as abstraction, encapsulation, inheritance, and polymorphism.

https://www.educative.io/blog/object-oriented-programming

Object-Oriented Programming Languages

- Object-oriented design paradigm in software engineering is widely used in many programming languages such as Java, C++, Python, etc.

https://mazer.dev/en/software-engineering/object-oriented-programming/oop-theory/what-is-oop-object-oriented-programming/

- It helps to create software systems that are easy to understand, modify, test, and reuse.

Object-Oriented Design (OOD)

- Object-oriented design (OOD) is a software engineering technique that involves creating a software system or application using an object-oriented paradigm. This means that the software is composed of objects, which are entities that have attributes (data) and behaviors (methods).
- Objects can interact with each other through messages, which are requests for actions or information.
- Objects can also be grouped into classes, which are categories that define the common properties and methods of a set of objects.
- Classes can inherit features from other classes, which allows for code reuse and abstraction.

Benefits of OOD

- It allows for modeling complex systems in a natural and intuitive way, by using real-world concepts and entities.
- It supports modularity, encapsulation, and polymorphism, which are principles that enhance the maintainability, extensibility, and reusability of software.
- It facilitates the development of large-scale and distributed software systems, by enabling the decomposition of problems into smaller and manageable units.
- It promotes software quality and reliability, by enabling the use of design patterns, UML diagrams, and testing frameworks.

Challenges of OOD

- It requires a good understanding of the problem domain and the user requirements, as well as the object-oriented concepts and principles.
- It involves a trade-off between simplicity and flexibility, as well as between performance and abstraction.
- It may introduce some overhead and complexity, due to the use of inheritance, dynamic binding, and message passing.

https://www.javatpoint.com/software-engineering-object-oriented-design

https://www.geeksforgeeks.org/oops-object-oriented-design/

https://www.scaler.com/topics/software-engineering/object-oriented-design/

Examples of Object-Oriented Software

Microsoft Word: This is a word processing software that allows users to create, edit, format, and print documents. Microsoft Word is written in C++ and uses object-oriented design principles such as abstraction, encapsulation, inheritance, and polymorphism.

Minecraft: This is a sandbox video game that allows players to build and explore a virtual world made of blocks. Minecraft is written in Java and uses object-oriented design principles such as abstraction, encapsulation, inheritance, and polymorphism.

Instagram: This is a social media platform that allows users to share photos and videos with their followers. Instagram is written in Python and uses object-oriented design principles such as abstraction, encapsulation, inheritance, and polymorphism.

UML Diagrams

UML diagrams are a way to visualize the design of a system using different types of diagrams.

UML stands for Unified Modeling Language, and it is a standard notation for many types of diagrams that can be grouped into three main categories: behavior diagrams, interaction diagrams, and structure diagrams.

https://en.wikipedia.org/wiki/Unified_Modeling_Language

UML Diagrams

- UML was developed in the 1990s by a group of experts from different object-oriented methods and notations.
- It was adopted as a standard by the Object Management Group (OMG) in 1997 and has been revised several times since then.
- UML can be used with various tools, such as Microsoft Visio, Visual Paradigm, or Rational Software.

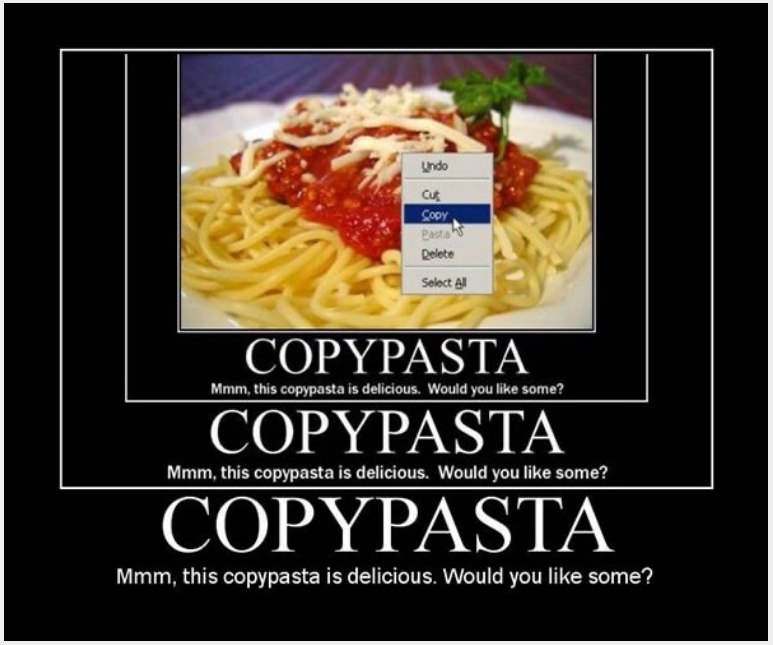
https://creately.com/blog/diagrams/uml-diagram-types-examples/

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.
- It is not a finished design that can be transformed directly into source or machine code. Rather, it is a description or template for how to solve a problem that can be used in many different situations.

https://en.wikipedia.org/wiki/Software_design_pattern

Software Design Patterns

- Software design patterns are formalized best practices that the programmer can use to solve common problems when designing an application or system.
- They typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved. They also provide implementation hints and examples.

Benefits of using Software Design Patterns

- They can speed up the development process by providing tested, proven development paradigms.
- They can improve the readability and maintainability of the code by using consistent and familiar terminology.
- They can promote code reuse and reduce duplication by abstracting common features and behaviors.
- They can facilitate communication and collaboration among developers by providing a common vocabulary and reference.

Challenges of using Software Design Patterns

- They can increase the complexity and learning curve of the code by introducing new concepts and abstractions.
- They can be overused or misused, leading to unnecessary or inappropriate design decisions.
- They can become outdated or irrelevant as technology and requirements evolve.

Some popular software design patterns

Factory Method: This is a creational pattern that defines an interface for creating an object, but lets subclasses decide which class to instantiate. This allows the creation process to be deferred to runtime.

Template method: This is a behavioral pattern that defines the skeleton of an algorithm or an operation in terms of a series of steps and allows the subclasses to implement some of the steps according to their specific needs while ensuring that the overall structure and sequence of the algorithm are preserved by the superclass,

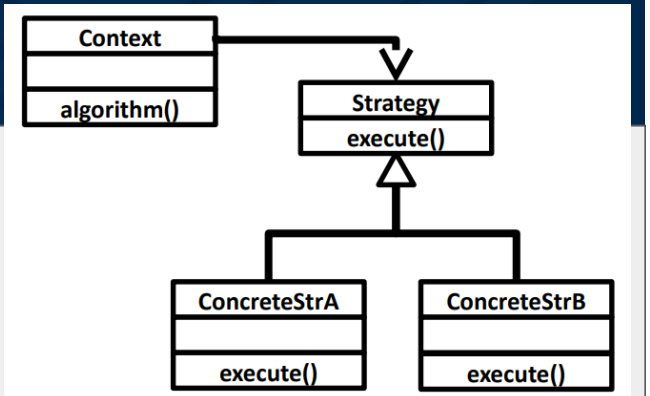
Decorator: This is a structural pattern that attaches additional responsibilities to an object dynamically. This provides a flexible alternative to subclassing for extending functionality.

Strategy: This is a behavioral pattern that defines a family of algorithms, encapsulates each one, and makes them interchangeable. This lets the algorithm vary independently from clients that use it.

Software Design Patterns

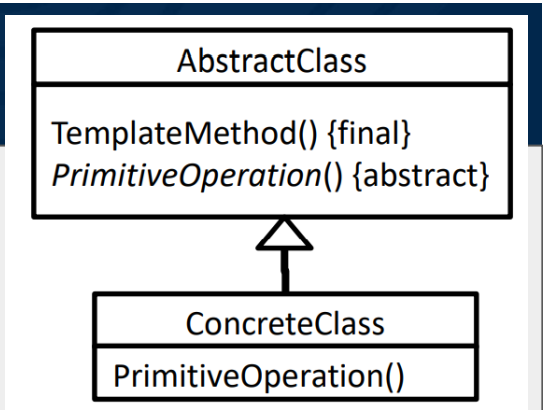
- A software **design pattern** is a general, reusable solution to a commonly occurring problem within a given context in software design.
  - (Other lectures have more details.)

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 by Contract (DbC),

- Design by Contract (DbC), is an approach for designing software that focuses on specifying contracts that define the interactions among components.
- A contract consists of preconditions, postconditions, and invariants, which are expressed as assertions that must be satisfied by the participating components.
- DbC was invented by Bertrand Meyer in the 1980s and is supported by some programming languages, such as Eiffel, Ada, and D.

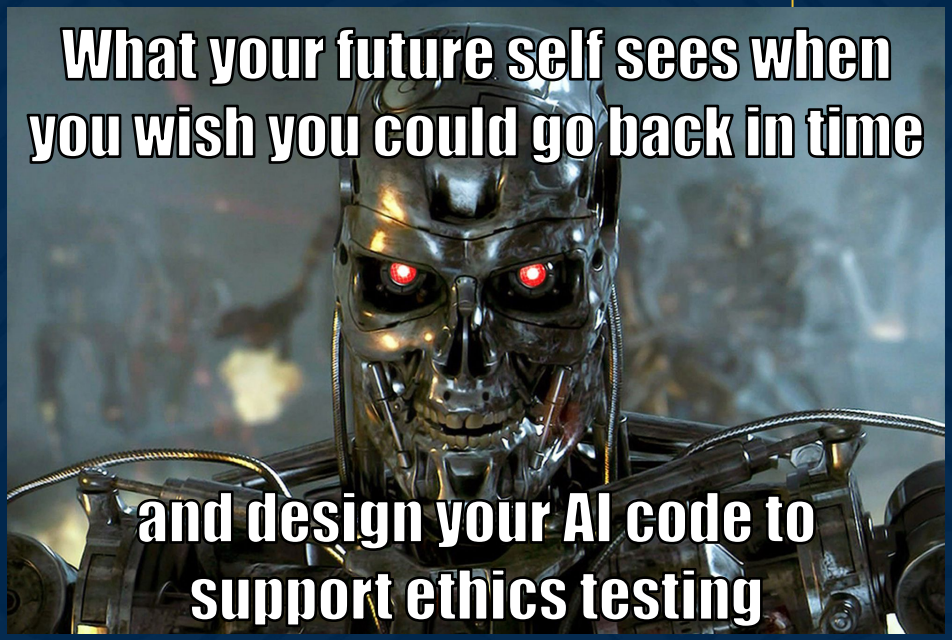
https://en.wikipedia.org/wiki/Design_by_contract

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!

MICHIGAN ENGINEERING
UNIVERSITY OF MICHIGAN

Design for Testing



Design for Testability (DFT)

- Design for testability in software engineering is an approach that aims to make software systems easier and more effective to test, both during development and after deployment.
- Design for testability can help to improve the quality and reliability of software systems by enabling various types of testing, such as unit testing, integration testing, system testing, acceptance testing, and regression testing.
- Design for testability can also help to reduce the cost and time of software development and maintenance by facilitating test automation, test coverage, test reuse, and test feedback.

https://www.infoq.com/articles/testability/

https://www.geeksforgeeks.org/design-for-testability-dft-in-software-testing/

Methods used in DFT

- Separating the concerns and responsibilities of different components, using modular and layered structures, interfaces, and contracts.
- Encapsulating the internal details and states of the components, using abstraction and information hiding.
- Isolating the dependencies and interactions of the components, using dependency injection, inversion of control, and mocks or stubs.
- Exposing the inputs and outputs of the components, using parameters, return values, and exceptions.
- Verifying the behavior and functionality of the components, using assertions, logging, and debugging tools.

Design for Testability

- If the majority cost of software engineering is maintenance, 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 (LOA)**, a variation of modular programming, or service-oriented architecture with a focus on the 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."

https://en.wikipedia.org/wiki/Library_Oriented_Architecture

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?”

Model View Controller (MVC)

- MVC, or Model-View-Controller, is a software architectural design pattern that separates application logic into three interrelated components- the model, view, and controller.
- The model is the component that handles the data and business logic of the application.
- The view is the component that handles the presentation and user interface of the application.
- The controller is the component that handles the communication and coordination between the model and the view. https://en.wikipedia.org/wiki/Architectural_pattern

Fault Injection

Fault injection is a testing technique for understanding how computing systems behave when stressed in unusual ways. This can be achieved using physical- or software-based means or using a hybrid approach.

Fault injection can help to improve the quality and reliability of software systems by enabling various types of testing, such as unit testing, integration testing, system testing, acceptance testing, and regression testing.

Fault injection can also help to identify and exploit vulnerabilities in software systems, such as bypassing security checks or corrupting data.

https://en.wikipedia.org/wiki/Fault_injection

Fault Injection Tools

Fault injection requires special tools and equipment to perform the attacks, such as glitch generators, electromagnetic pulse generators, or lasers.

Some of these tools are commercially available, such as ChipWhisperer, ChipShouter, or Spider. Others can be custom-built by the attackers.

The cost and complexity of fault injection attacks vary depending on the type of fault injection and the target system.

https://www.riscure.com/fault-injection/

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

- The 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

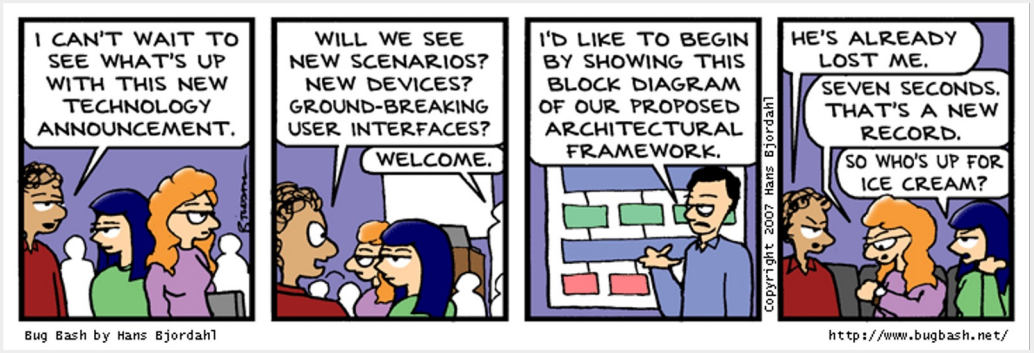
- Line coverage was often inadequate because “visit line 5 when ptr==null” could be very different from “visit line 5 when ptr !=null”
  - Because “*ptr = 9” is really “if (ptr == null) abort(); else *ptr = 9;”
- Consider explicit conditionals that check requirements adherence
  - To get coverage points 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])?

MICHIGAN ENGINEERING
UNIVERSITY OF MICHIGAN

Questions?

- HW5 is due next Monday!