# Software Quality

17-313 Spring 2025

Foundations of Software Engineering

https://cmu-313.github.io

Michael Hilton, Austin Henley, and Nadia Nahar

Sources:

- Effective Software Testing: A developer's guide. Maurizio Aniche
- Software Quality and Testing - TU Delft
- Introduction to Combinatorial Testing. Rick Kuhn
- Managing Technical Debt. Ipek Ozkaya. CMU SEI

# Administrivia

# Smoking Section

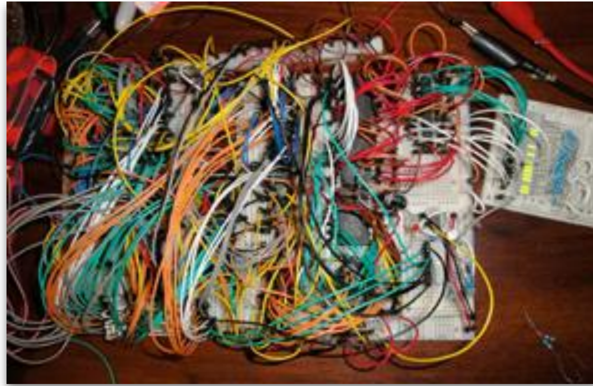
- Last **two** full rows

# Learning Goals

- Understand the concepts of software quality and technical debt
- Reflect on personal experiences of technical debt
- Learn best practices for proactively ensuring quality
- Learn techniques for creating functional tests
- Explain the importance of technical debt management
- Learn techniques for managing technical debt

# Software Quality

## Internal Quality



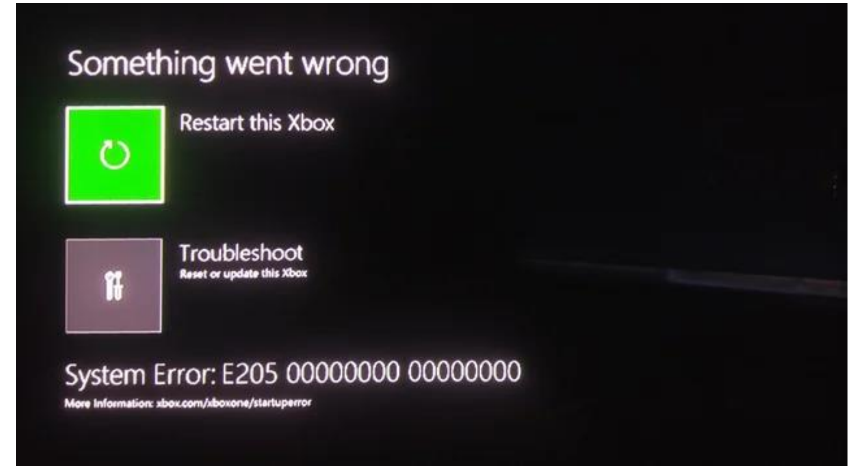
- Is the code well structured?
- Is the code understandable?
- How well documented?

## External Quality

- Does the software crash?
- Does it meet the requirements?
- Is the UI well designed?

# Testing

Assuring external quality

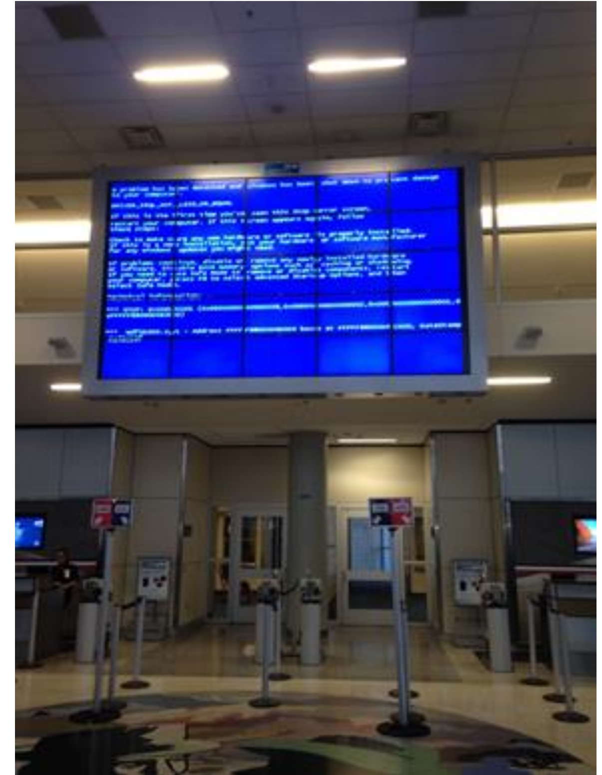

Something went wrong
Restart this Xbox
Troubleshoot
Reset or update this Xbox
System Error: E205 00000000 00000000
More Information: xbox.com/xboxone/startuperror

# Terminology

**Failure:**

*"Deviation of the component or system from its expected delivery, service or result"*

*"Manifested inability of a system to perform required function"*



# Terminology

**Fault / Defect:**

*“Flaw in component or system that can cause the component or system to fail to perform its required function”*

*“A defect, if encountered during execution, may cause a failure of the component or system”*

# Terminology

**Error:**

*"A human action that produces an incorrect result"*

# Terminology

**Failure:**

- Manifested inability of a system to perform required function.

**Defect (fault):**

- missing / incorrect code

**Error (mistake)**

- human action producing fault

} Bug

And thus:

- Testing: Attempt to trigger failures
- Debugging: Attempt to find faults given a failure

# Principles of Testing #1: Avoid the *absence of defects* fallacy

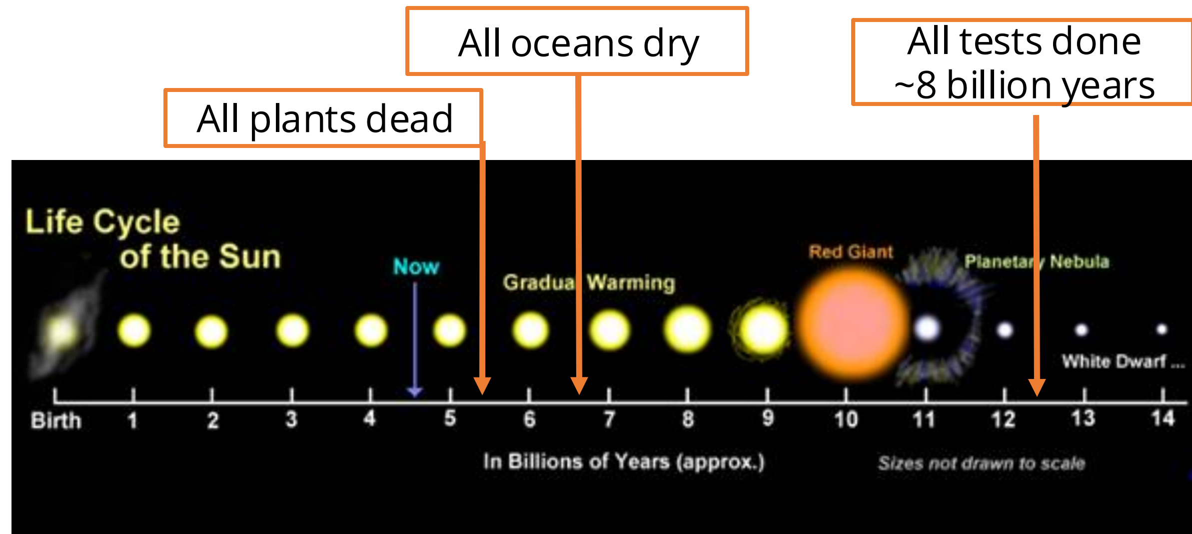
- Testing shows the presence of defects
- Testing does not show the absence of defects!
- “no test team can achieve 100% defect detection effectiveness”

*Effective Software Testing: A developer's guide.* Maurizio Aniche

# Principles of Testing #2: Exhaustive testing is impossible

```python
def is_valid_email(email: str) -> bool:
    ...
```

- A simple function, 1 input, string, max. 26 lowercase characters + symbols (@, ., _, -)
- Assume we can use 1 zettaFLOPS: $10^{21}$ tests per second




*Effective Software Testing: A developer's guide.* Maurizio Aniche

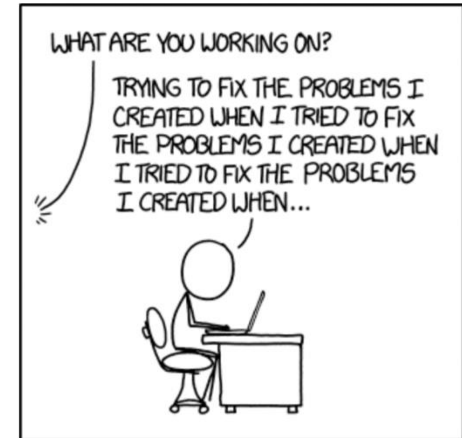
# Principles of Testing #3: Start testing early

- To let tests guide design
- To get feedback as early as possible
- To find bugs when they are cheapest to fix
- To find bugs when have caused least damage

*Effective Software Testing: A developer's guide.* Maurizio Aniche

# Principles of Testing #4: Defects are usually clustered

- “Hot” components requiring frequent change, bad habits, poor developers, tricky logic, business uncertainty, innovative, size, …
- Use as heuristic to focus test effort




*Effective Software Testing: A developer's guide.* Maurizio Aniche

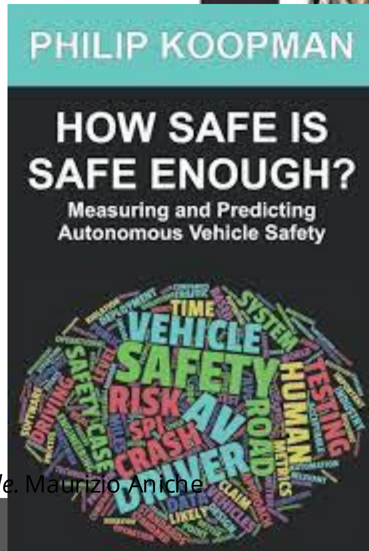
# Principles of Testing #5: The pesticide paradox

*“Every method you use to prevent or find bugs leaves a residue of subtler bugs against which those methods are ineffectual.”*

- Re-running the same test suite again and again on a changing program gives a false sense of security
- Variation in testing

*Effective Software Testing: A developer's guide.* Maurizio Aniche

# Principles of Testing #6: Testing is context-dependent



*Effective Software Testing: A developer's guide*. Maurizio Aniche

# Principles of Testing #7: Verification is not validation

Verification

- Does the software system meet the requirements specifications?
- Are we building the **software right**?

Validation

- Does the software system meet the user's real needs?
- Are we building the **right software**?

Credit: Philip Koopman

*Effective Software Testing: A developer's guide.* Maurizio Aniche

# How to create tests?

# Test design techniques

- **Opportunistic/exploratory testing**: Add some unit tests, without much planning
- **Specification-based testing ("black box"):** Derive test cases from specifications
  - Boundary value analysis
  - Equivalence classes
  - Combinatorial testing
  - Random testing
- **Structural testing ("white box"):** Derive test cases to cover implementation paths
  - Line coverage, branch coverage

# Specification Testing

Tests are based on the specification

**Advantages:**

- Avoids implementation bias
- Robust to changes in the implementation
- Tests don’t require familiarity with the code
- Tests can be developed before the implementation

```python
"""
Compute the price of a bus ride:
   - Children under 2 ride for free.
   - Children under 18 and senior citizens over 65 pay half the fare
   - All others pay the full fare of $3.
   - On weekdays (Monday to Friday), between 7am and 9am and
     between 4pm and 6pm, a peak surcharge of $1.5 is added
     to the fare.
  -  During weekends (Saturday and Sunday), there is a flat rate
     of $2 for all riders, except for children under 2.
  -  Short trips under 5 minutes during off-peak times are free,
     except on weekends.
   - If the trip occurs on a public holiday, a special holiday surcharge
     of $2 is added, ignoring other surcharges and the weekend flat rate.
 """
def bus_ticket_price(age: int,
                     ride_datetime: datetime,
                     ride_duration: int,
                     is_public_holiday: bool) -> float:
    ...
```

# What about exhaustive testing?

**Idea: Try all values!**

- **age: int** (2 - 117) years
- **datetime: DateTime** (hh:mm + M/D/Y)
- **rideTime: int** (in minutes, 1 - 2 Hours)
- **is_public_holiday: bool** (2 values)

116 x 1440 (minutes per day) x 1826 (days in the next 5 years) x 120 (ride time) x 2

**~ 72 Billion test cases**

# What about exhaustive testing?

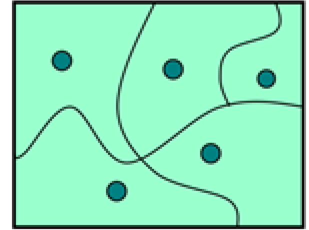
Exhaustive testing is usually impractical – even for trivially small problem

Key problem: choosing test suite

- **Small enough** to finish in a useful amount of time
- **Large enough** to provide a useful amount of validation

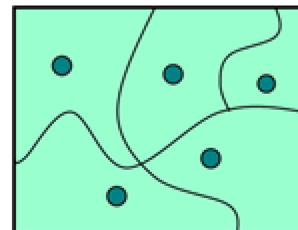
Alternative: **Heuristics**

# Equivalence Partitioning



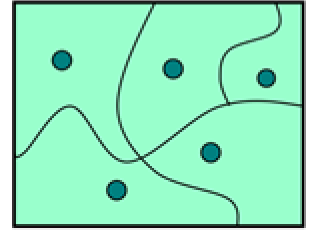
- Identify sets with same behavior (**equivalence class**)
- Try one input from each set
- Equivalence classes derived from specifications (e.g., cases, input ranges, error conditions, fault models)
- Requires domain-knowledge

# Example: Equivalence Classes?



```
1  """
2   Compute the price of a bus ride:
3     - Children under 2 ride for free.
4     - Children under 18 and senior citizens over 65 pay half the fare
5     - All others pay the full fare of $3.
6     - On weekdays (Monday to Friday), between 7am and 9am and
7       between 4pm and 6pm, a peak surcharge of $1.5 is added
8       to the fare.
9    -  During weekends (Saturday and Sunday), there is a flat rate
10      of $2 for all riders, except for children under 2.
11   -  Short trips under 5 minutes during off-peak times are free,
12      except on weekends.
13    - If the trip occurs on a public holiday, a special holiday surcharge
14      of $2 is added, ignoring other surcharges and the weekend flat rate.
15  """
16 def bus_ticket_price(age: int,
17                      ride_datetime: datetime,
18                      ride_duration: int,
19                      is_public_holiday: bool) -> float:
20     ...
```

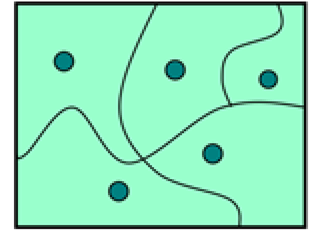
# Boundary-value analysis



**Key Insight:** Errors often occur at the boundaries of a variable value

- For each variable, select:
  - minimum,
  - min+1,
  - medium,
  - max-1,
  - maximum;
  - possibly also invalid values min-1, max+1

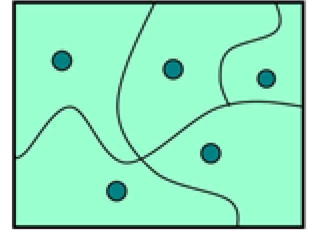
# Boundary-value analysis



```
1  """
2   Compute the price of a bus ride:
3     - Children under 2 ride for free.
4     - Children under 18 and senior citizens over 65 pay half the fare
5     - All others pay the full fare of $3.
6     - On weekdays (Monday to Friday), between 7am and 9am and
7       between 4pm and 6pm, a peak surcharge of $1.5 is added
8       to the fare.
9    -  During weekends (Saturday and Sunday), there is a flat rate
10      of $2 for all riders, except for children under 2.
11   -  Short trips under 5 minutes during off-peak times are free,
12      except on weekends.
13    - If the trip occurs on a public holiday, a special holiday surcharge
14      of $2 is added, ignoring other surcharges and the weekend flat rate.
15  """
16 def bus_ticket_price(age: int,
17                      ride_datetime: datetime,
18                      ride_duration: int,
19                      is_public_holiday: bool) -> float:
20     ...
```

| Variable | Domains |
|---|---|
| age | <2, [2,17], [18,65], >65 |
| ride_datetime | weekdays peak and off-peak, weekends peak and off-peak … |
| ride_duration | <5, >=5 |
| is_public_holiday | F, T |

# Pairwise testing

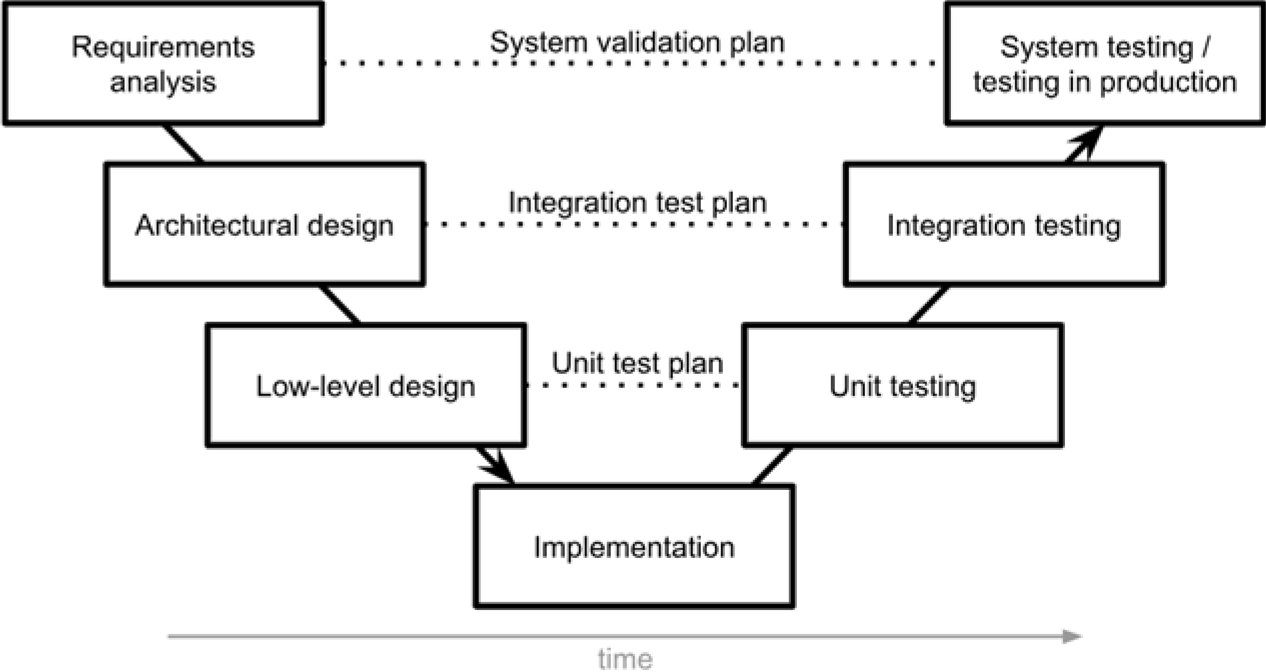


**Key Insight:** some problems only occur as the result of an interaction between parameters/components

- Examples of interactions:
  - The bug occurs for senior citizens traveling on weekends (pairwise interaction)
  - The bug occurs for senior citizens traveling on weekends during peak hours (3-way interaction)
  - The bug occurs for adults traveling long trips during public holidays that are weekends. (4-way interaction)
- **Claim: Considering pairwise interactions finds about 50% to 90% of defects**

# When to create and run tests?

# The V-Model



# Group Activity

- We are taking over the reigns of NodeBB
- Come up with a testing protocol for the system
  - What should we prioritize testing?
  - How should we test? (run it? unit test? …?)
  - When should we write new tests?
  - How do we know when to stop testing?
  - If we discover a bug, what then?

*If we spend all our time testing… how will we ever add new features?!*

# Technical Debt

# A better analogy?: Pollution

AQI Basics for Ozone and Particle Pollution

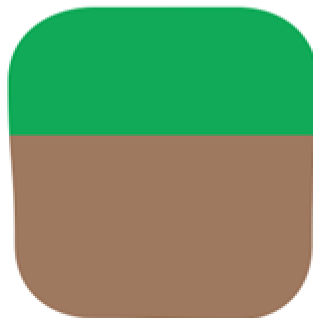
| Daily AQI Color | Levels of Concern | Values of Index | Description of Air Quality |
|---|---|---|---|
| Green | Good | 0 to 50 | Air quality is satisfactory, and air pollution poses little or no risk. |
| Yellow | Moderate | 51 to 100 | Air quality is acceptable. However, there may be a risk for some people, particularly those who are unusually sensitive to air pollution. |
| Orange | Unhealthy for Sensitive Groups | 101 to 150 | Members of sensitive groups may experience health effects. The general public is less likely to be affected. |
| Red | Unhealthy | 151 to 200 | Some members of the general public may experience health effects; members of sensitive groups may experience more serious health effects. |
| Purple | Very Unhealthy | 201 to 300 | Health alert: The risk of health effects is increased for everyone. |
| Maroon | Hazardous | 301 and higher | Health warning of emergency conditions: everyone is more likely to be affected. |

https://www.airnow.gov/aqi/aqi-basics

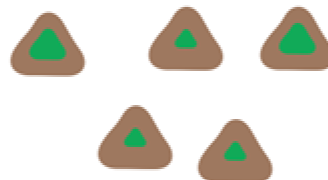
# Technical debt

*Any software system has a certain amount of essential complexity required to do its job…*

*… but most systems contain cruft that makes it harder to understand.*



*Cruft causes changes to take **more effort***

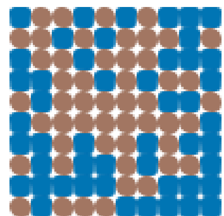


*The technical debt metaphor treats the cruft as a debt, whose interest payments are the extra effort these changes require.*

*https://martinfowler.com/bliki/TechnicalDebt.html*

# Internal quality makes it easier to add features

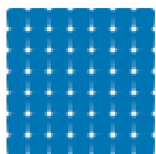
If we compare one system with a lot of cruft...



the cruft means new features take longer to build

this extra time and effort is the cost of the cruft, paid with each new feature

...to an equivalent one without



free of cruft, features can be added more quickly

# Examples of technical debt

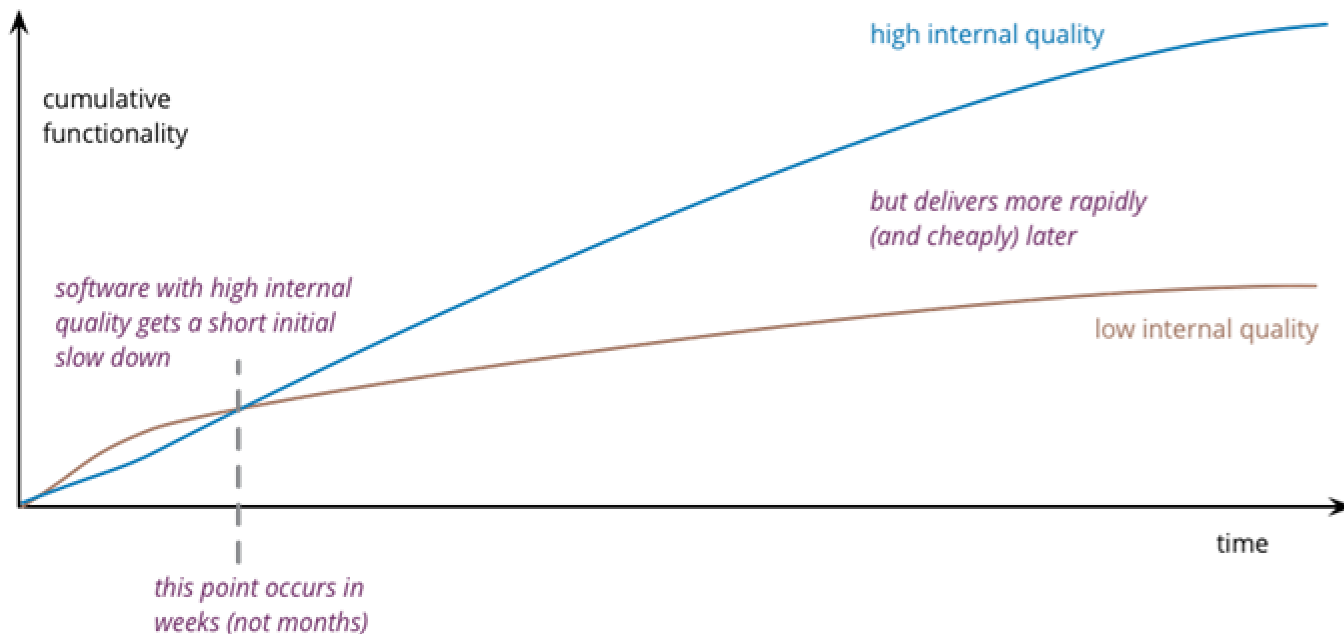
# Technical Debt != Bad Internal Quality

*"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.** "*

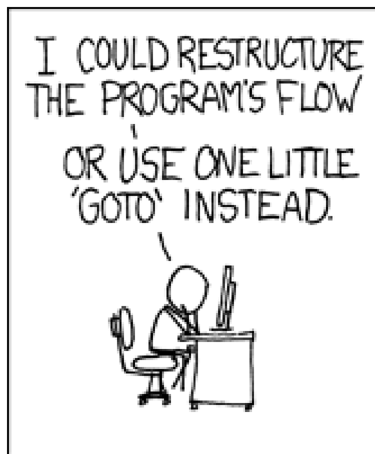
*"Technical debt is a contingent liability whose impact is l**imited to internal system qualities** – primarily, but not only, **maintainability** and **evolvability**."*

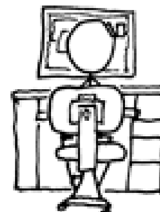
*Managing Technical Debt: Reducing Friction in Software Development.* Philippe Kruchten, Robert Nord, Ipek Ozkaya

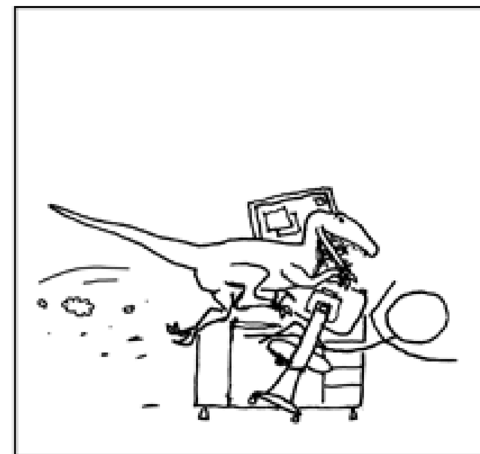
# High internal quality is an investment



# What actions cause technical debt?







# What actions cause technical debt?

Tightly-coupled components

Poorly-specified requirements

Business pressure

Lack of process

Lack of documentation

Lack of automated testing

Lack of knowledge

Lack of ownership

Delayed refactoring

Multiple, long-lived development branches

…

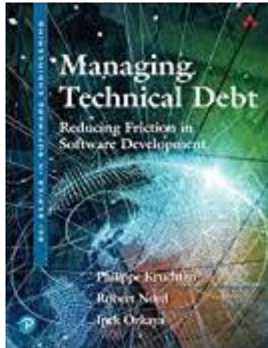
# Bitrot: Even if your software doesn't change, it will break over time

# EVERYONE CREATES TECHNICAL DEBT

# Bad: Too much technical debt

- Bad code can be demoralizing
- Conversations with the client become awkward
- Team infighting
- Turnover and attrition
- Development speed
- …

# How to manage technical debt?

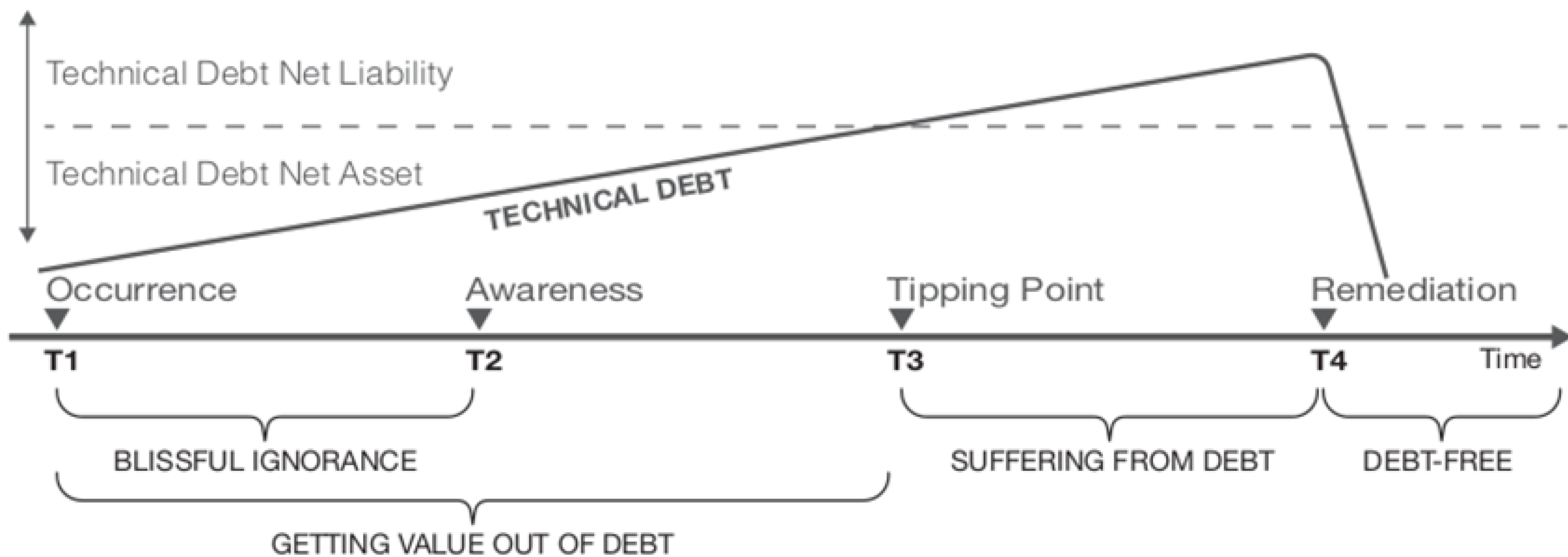



*Managing Technical Debt: Reducing Friction in Software Development.*
Philippe Kruchten, Robert Nord, Ipek Ozkaya

# Principles of Technical Debt Management

1. Technical debt is a useful rhetorical concept for dialogue.
2. If you do not incur any form of interest, then you probably do not have actual technical debt.
3. All systems have technical debt.
4. Technical debt must trace to the system.
5. Technical debt is not synonymous with bad quality.
6. Architecture technical debt has the highest cost of ownership.
7. All code matters!
8. Technical debt has no absolute measure.
9. Technical debt depends on the future evolution of the system.

# When should we reduce technical debt?


Technical Debt Net Liability
Technical Debt Net Asset
TECHNICAL DEBT
Occurrence
Awareness
Tipping Point
Remediation
T1
T2
T3
T4
Time
BLISSFUL IGNORANCE
SUFFERING FROM DEBT
DEBT-FREE
GETTING VALUE OUT OF DEBT

# Managing technical debt

Organizations needs to address the following challenges continuously:

1. Recognizing technical debt
2. Making technical debt visible
3. Deciding when and how to resolve debt
4. Living with technical debt

# Not all technical debt is the same

| | **Reckless** | **Prudent** |
|---|---|---|
| **Deliberate** | *“We don’t have time for design”* | *“We must ship now and deal with consequences (later)”* |
| **Inadvertent** | *“What’s layering?”* | *“Now we know how we should have done it”* |

https://martinfowler.com/bliki/TechnicalDebtQuadrant.html

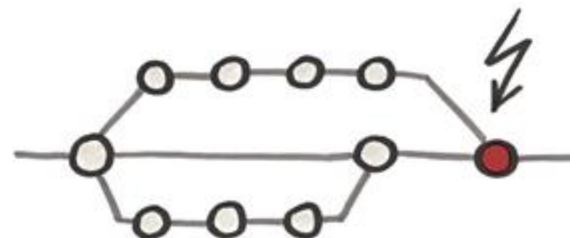
# How can we avoid (inadvertent) technical debt?

# Common Anti-Patterns

- Not having a QA process! Or no-one follows it

# Common Anti-Patterns

- Not having a QA process! Or no-one follows it
- Bad version control practices
  - Everyone commits to the main branch
  - Long-lived feature branches
  - Huge PRs



| COMMENT | DATE |
|---|---|
| CREATED MAIN LOOP & TIMING CONTROL | 14 HOURS AGO |
| ENABLED CONFIG FILE PARSING | 9 HOURS AGO |
| MISC BUGFIXES | 5 HOURS AGO |
| CODE ADDITIONS/EDITS | 4 HOURS AGO |
| MORE CODE | 4 HOURS AGO |
| HERE HAVE CODE | 4 HOURS AGO |
| AAAAAAAA | 3 HOURS AGO |
| ADKFJSLKDFJSDKLFJ | 3 HOURS AGO |
| MY HANDS ARE TYPING WORDS | 2 HOURS AGO |
| HAAAAAAAAANDS | 2 HOURS AGO |

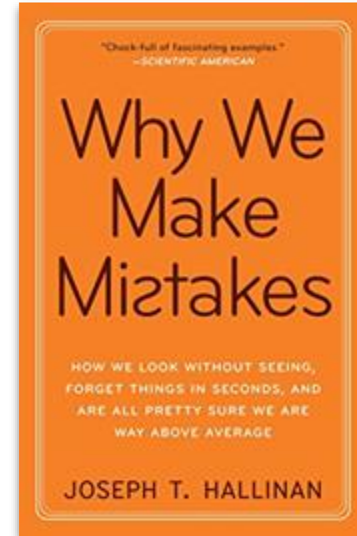
AS A PROJECT DRAGS ON, MY GIT COMMIT MESSAGES GET LESS AND LESS INFORMATIVE.

# Common Anti-Patterns

- Not having a QA process! Or no-one follows it
- Bad version control practices
- Slow and encumbering QA processes
  - changes take forever to get merged
  - time could be better spent on new features

# Common Anti-Patterns

- Not having a QA process! Or no-one follows it
- Bad version control practices
- Slow and encumbering QA processes
- Reliance on repetitive manual labor
  - focused on superficial problems rather than structural ones
  - results may vary (e.g., manual testing)
  - mistakes will happen!



# Case Study: Knight Capital

## Knightmare: A DevOps Cautionary Tale

D7 DevOps April 17, 2014 6 Minutes

I was speaking at a conference last year on the topics of DevOps, Configuration as Code, and Continuous Delivery and used the following story to demonstrate the importance making deployments fully automated and repeatable as part of a DevOps/Continuous Delivery initiative. Since that conference I have been asked by several people to share the story through my blog. This story is true – this really happened. This is my telling of the story based on what I have read (I was not involved in this).

In layman's terms, Knight Capital Group realized a $460 million loss in 45-minutes. Remember, Knight only has $365 million in cash and equivalents. **In 45-minutes Knight went from being the largest trader in US equities and a major market maker in the NYSE and NASDAQ to bankrupt.**

# Summary:

- Software Quality is hard
- Life involves tradeoffs