Software Quality

17-313 Fall 2024

Foundations of Software Engineering

https://cmu-313.github.io

Michael Hilton and Rohan Padhye

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

S3D Software and Societa Systems Department

Carnegie Mellon University

Administrivia

- Reminder: TA interest form here:

P4:

- **Checkpoint #1 Deliverables** – 55 pts - due Friday, Nov 8, 11:59pm
  - Architectural Design Document (40 pts)
  - Basic LLM Experiment (15 pts)
- **Checkpoint #2 Deliverables** – 10 pts - due Wed, Nov 13, 11:59pm
  - LLM Experiment Integration Checkpoint (10 pts)
- **Final Deliverables** - 60 pts - due Friday, November 15, 11:59pm
  - Final LLM Experiment Implementation (30 pts)
  - Evaluation Report (30 pts)

Smoking Section

- Last **two** full rows

P3 Discussion

P4 DEM0

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
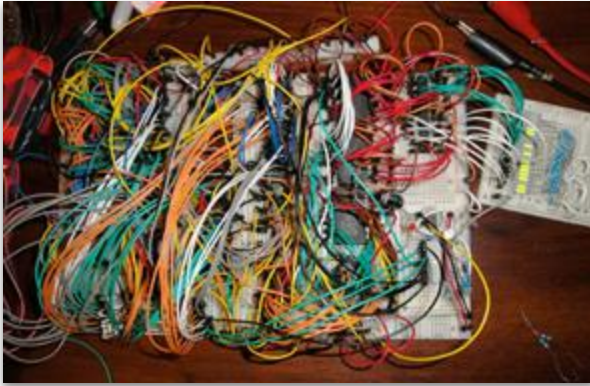
- Is the code well structured?

- Is the code understandable?

- How well documented?

External Quality

- Does the software crash?

- Does it meet the requirements?

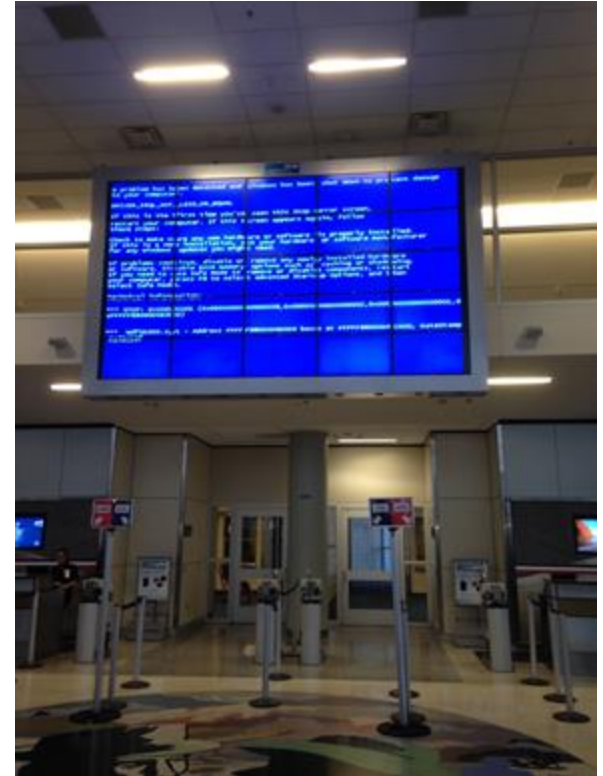- Is the UI well designed?

Testing

Assuring external quality

Carnegie Mellon University

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!
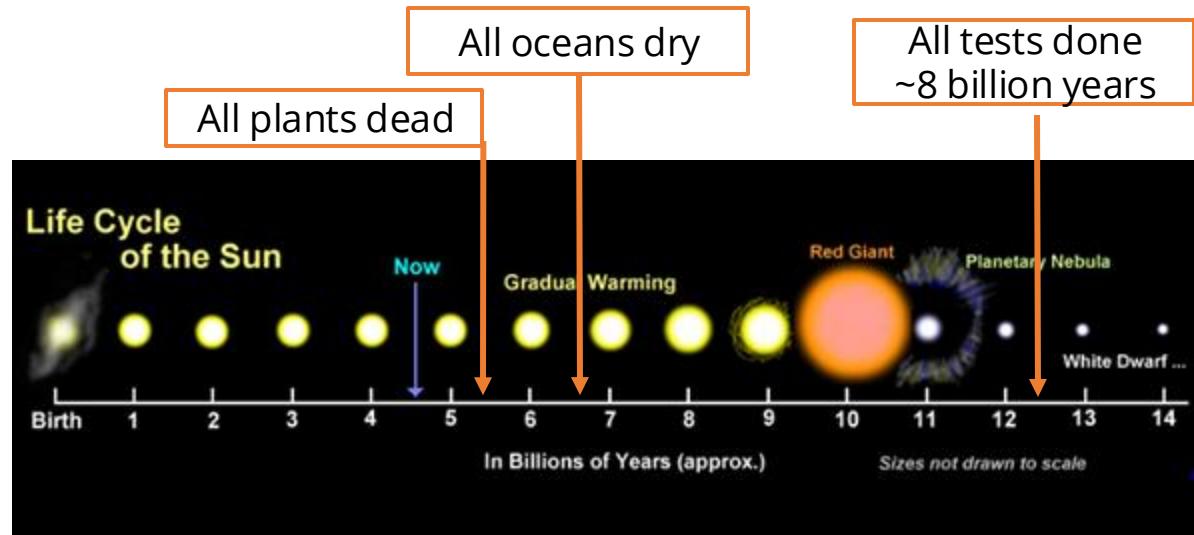- "no test team can achieve 100% defect detection effectiveness"

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

Carnegie
Mellon
University

Principles of Testing #2: Exhaustive testing is impossible

```
1  def is_valid_email(email: str) -> bool:
2      ...
```

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

All plants dead

All oceans dry

All tests done ~8 billion years

Life Cycle of the Sun

Now

Gradual Warming

Red Giant

Planetary Nebula

White Dwarf ...

Birth 1 2 3 4 5 6 7 8 9 10 11 12 13 14

In Billions of Years (approx.)

Sizes not drawn to scale

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

S3D Software and Societa
Systems Department

Carnegie
Mellon
University

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

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

Principles of Testing #6:
Testing is context-dependent
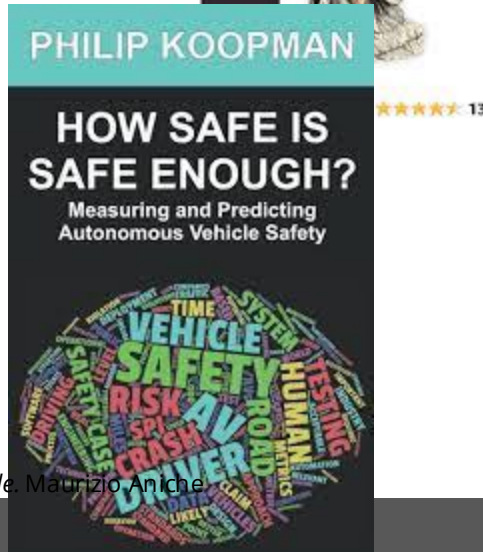
Effective Software Testing: A developer's guide. Maurizio Aniche.

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
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      ...
```

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

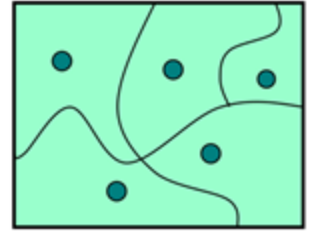**~ 72 Billion test cases**

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
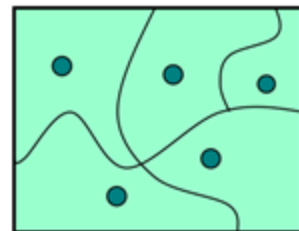- **Large enough** to provide a useful amount of validation

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

Carnegie
Mellon
University

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      ...
```
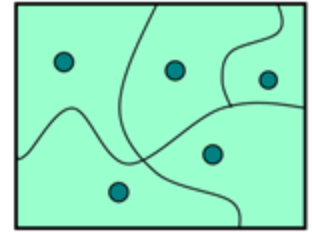
Boundary-value analysis
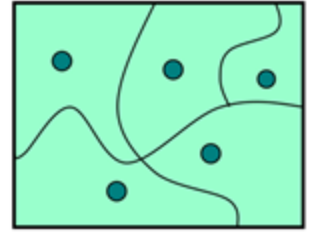
Key Insight: Errors often occur at the boundaries of a variable value

- 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     ...
```

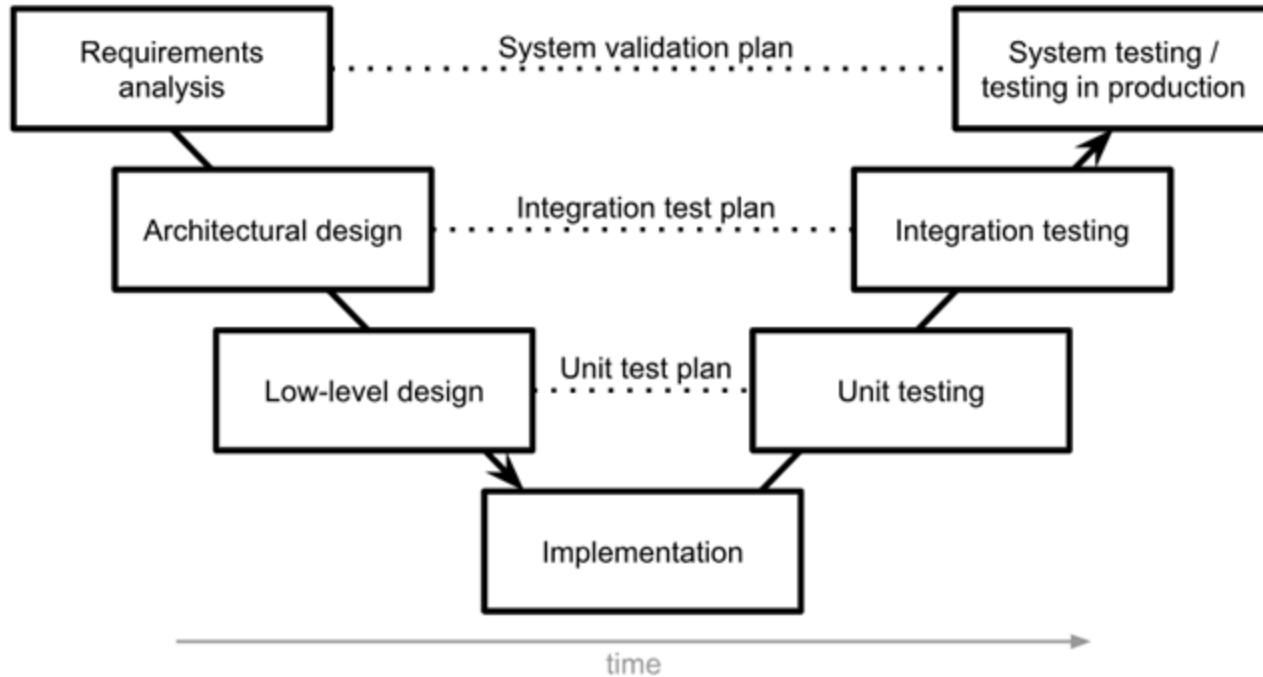| Variable	Domains
age	<2, [2,17], [18,65], >65
ride_datetime	weekdays peak and off-peak, weekends peak and off-peak …
ride_duration	<5, >=5
is_public_holiday	F, T

Pairwise testing
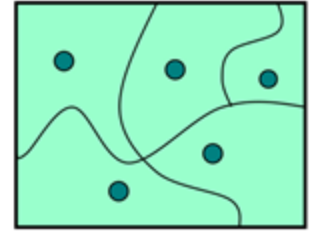
Key Insight: some problems only occur as the result of

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

Requirements analysis ···· System validation plan ···· System testing / testing in production

Architectural design ···· Integration test plan ···· Integration testing

Low-level design ···· Unit test plan ···· Unit testing

Implementation

time

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

S3D Software and Societal Systems Department

Carnegie Mellon University

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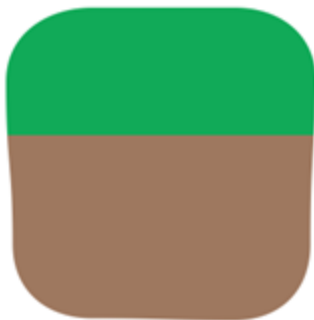
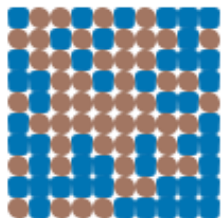Cruft causes changes to take **more effort**

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

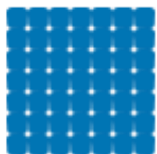If we compare one system with *a lot of cruft*...

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

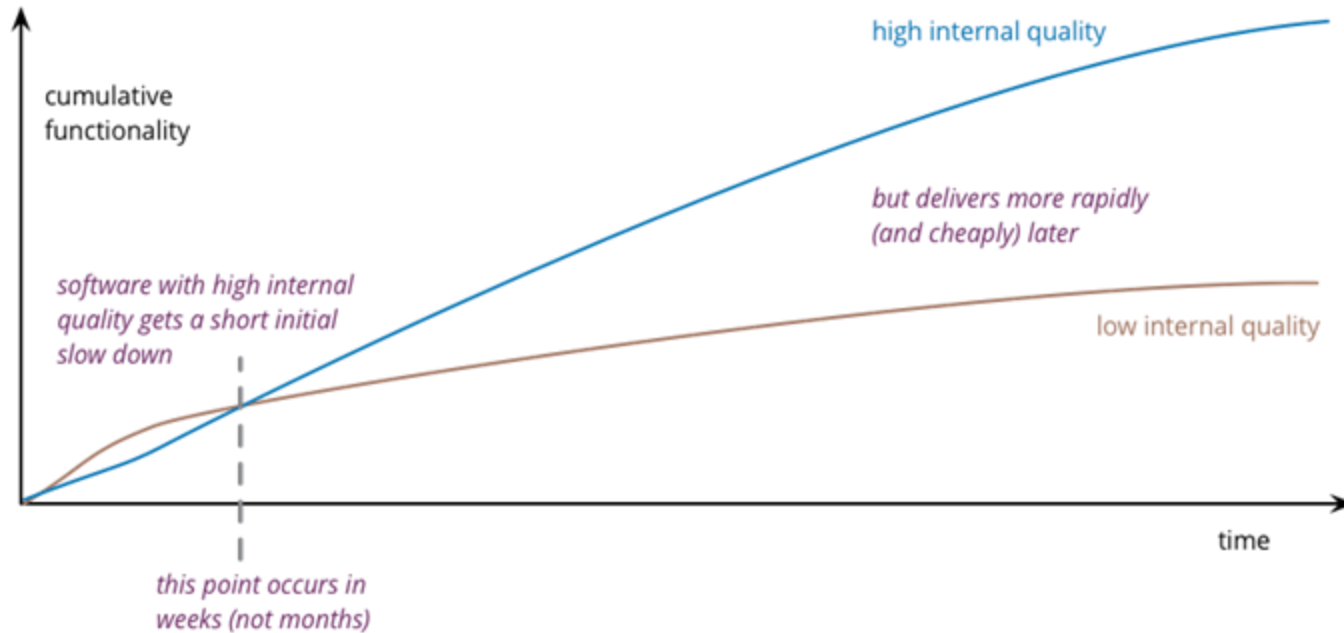free of cruft, features can be added more quickly

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

S3D Software and Societal Systems Department

Carnegie Mellon University

High internal quality is an investment

cumulative functionality

high internal quality

but delivers more rapidly (and cheaply) later

software with high internal quality gets a short initial slow down

low internal quality

time

this point occurs in weeks (not months)

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
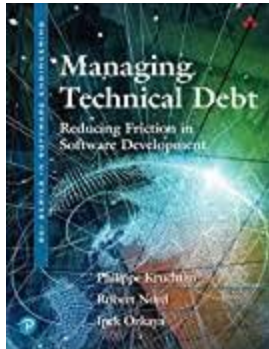
EVERYONE

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?
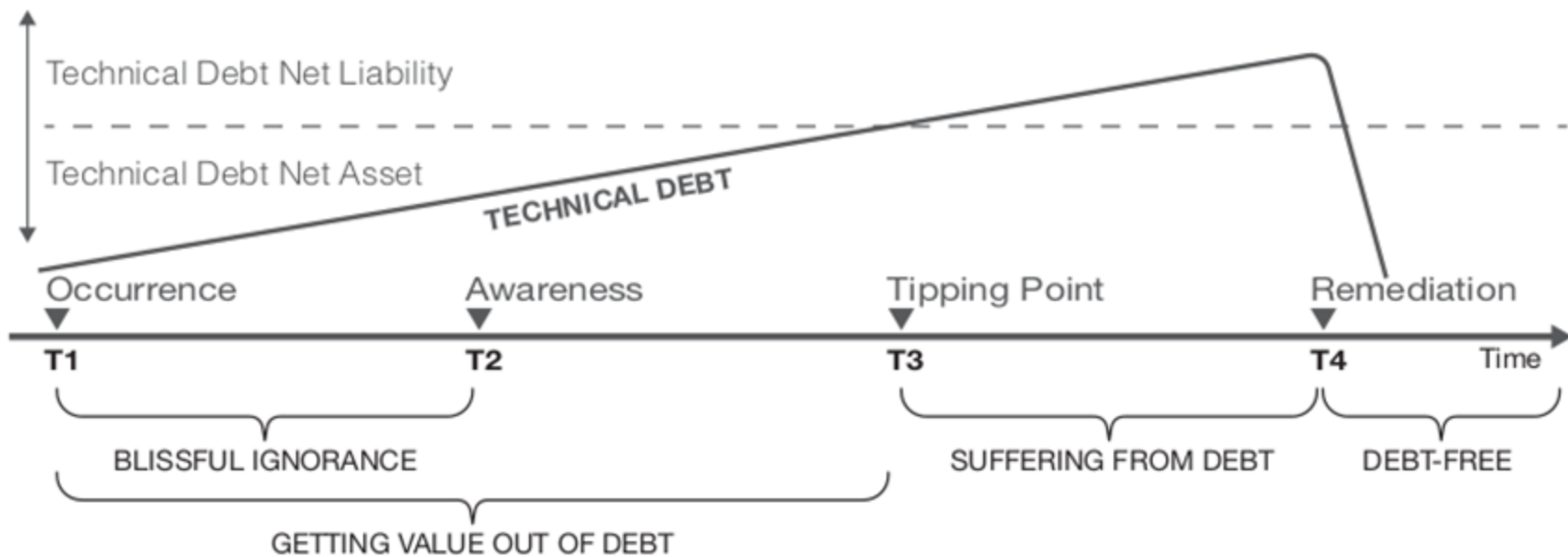
Managing technical debt

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

Group Activity

Describe two examples of technical debt in NodeBB

1. Deliberate, prudent
2. Reckless, inadvertent

Discuss the reason for incurring debt (e.g., value added?) and the debt payback strategy

Summary:

- Software Quality is hard

- Life involves tradeoffs

Carnegie Mellon University