COMP0104 Software Development Practice: Technical Debt

Jens Krinke

Centre for Research on Evolution, Search & Testing
Software Systems Engineering Group
Department of Computer Science
University College London

Overview

- Lehman’s Laws of Software Evolution
- Technical Debt
- Refactoring
- Legacy Code

Lehman’s Laws of Software Evolution

Lehman’s laws of software evolution is a series of “laws” that Lehman and Belady formulated starting in 1974 with respect to software evolution.

Lehman’s Laws (subset)

- Continuing Change (1974, Law I):
  Systems must be continually adapted
  else they become progressively less satisfactory.
- Increasing Complexity (1974, Law II):
  As a system evolves its complexity increases
  unless work is done to maintain or reduce it.

Lehman's Laws (subset)

- Continuing Growth (1980, Law VI):
  The functional content of systems
  must be continually increased
  to maintain user satisfaction over their lifetime.
- Declining Quality (1996, Law VII):
  The quality of systems will appear to be declining
  unless they are rigorously maintained and adapted
  to operational environment changes.

Technical Debt (Cunningham 1992)

“Technical debt is the debt that accrues when you knowingly or unknowingly make wrong or non-optimal design decisions.”

- A quick fix instead of a good solution is adding to the technical debt and the debt is repaid when the quick fix is replaced with a well-designed solution.
- If the technical debt is not “repaid”, it will accrue “interest” as fixing the issue will become more expensive.
- Too much technical debt can make cause bankruptcy, when the software becomes unmaintainable.

…in other words…

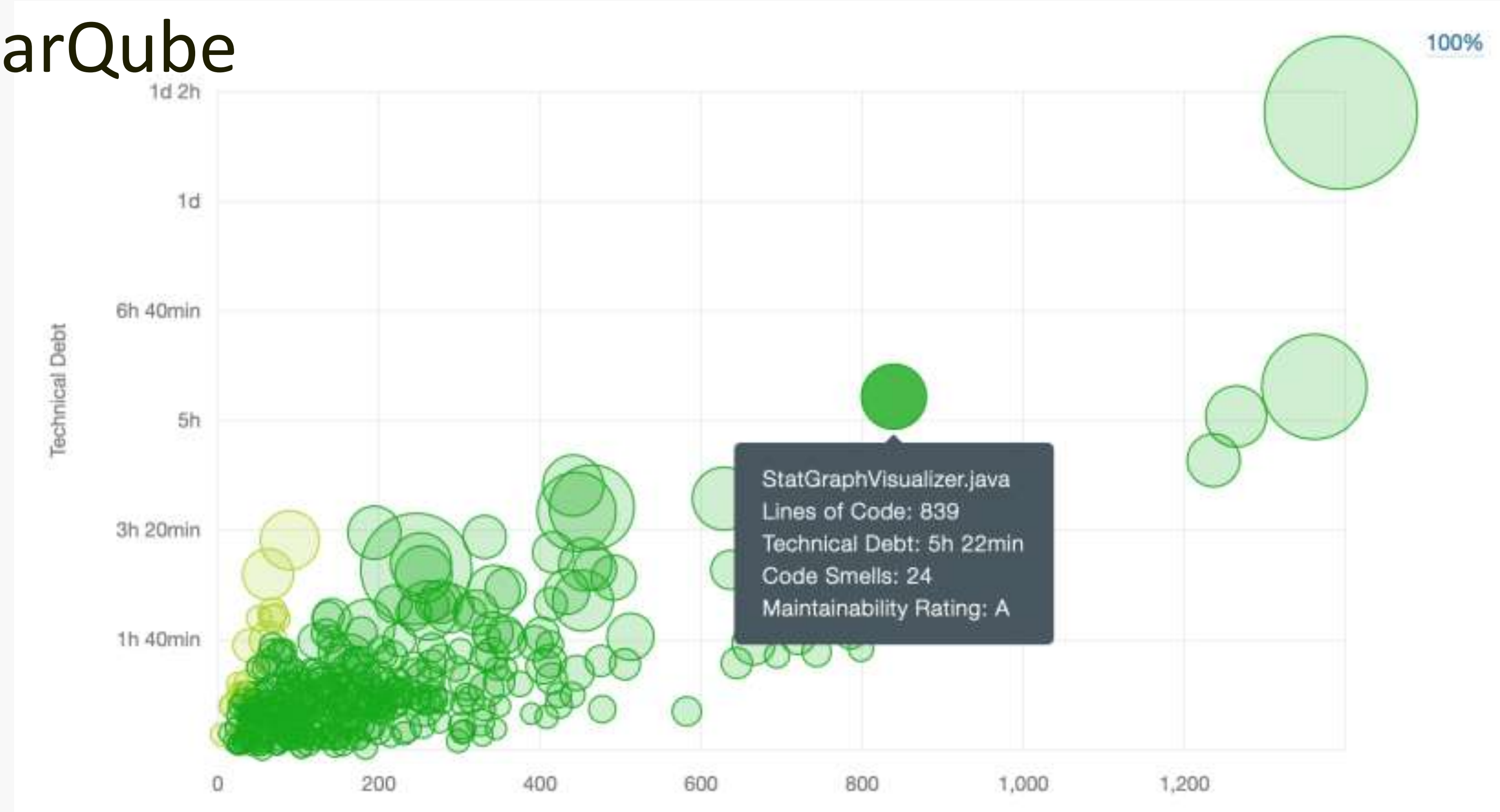
Technical debt is caused by artefacts that are

- Incomplete
- Immature
- Inadequate

Technical Debt in Industry

- The metaphor of technical debt is used to explain the impact of cutting corners.
- Although being coined as a metaphor, it has become an important concept in industry.
- Important:
  technical debt can be taken on without intent, but repayment usually requires intent.

SonarQube



100%
1d 2h
1d
6h 40min
5h
3h 20min
1h 40min
Technical Debt
StatGraphVisualizer.java
Lines of Code: 839
Technical Debt: 5h 22min
Code Smells: 24
Maintainability Rating: A
0
200
400
600
800
1,000
1,200

Sources of Technical Debt (1)

- Code Debt:
  - Inconsistent coding style
  - Code smells
  - Static analysis tools violations
- Design Debt:
  - Violations of design rules
  - Design smells
- Architectural Debt:
  - Sub-optimal architectural design and implementation choices

Sources of Technical Debt (2)

- Test Debt:
  - Lack of tests
  - Inadequate test coverage
  - Improper test design
  - Test smells
- Documentation Debt:
  - No / poor / outdated documentation

Faults and failures are not part of technical debt

- Faults and failures are visible to users, they affect the external quality of the software.
- Technical debt is not visible to users and affects the internal quality of the software.
- Faults and failures usually have priority over technical debt.

What causes Technical Debt?

- Lack of awareness of technical debt.
- Lack of awareness of code / design smells and refactoring.
- Lack of application of design principles.
- Lack of skilled designers.
- Schedule pressure.

External Factors

Technical debt can occur through external influences:

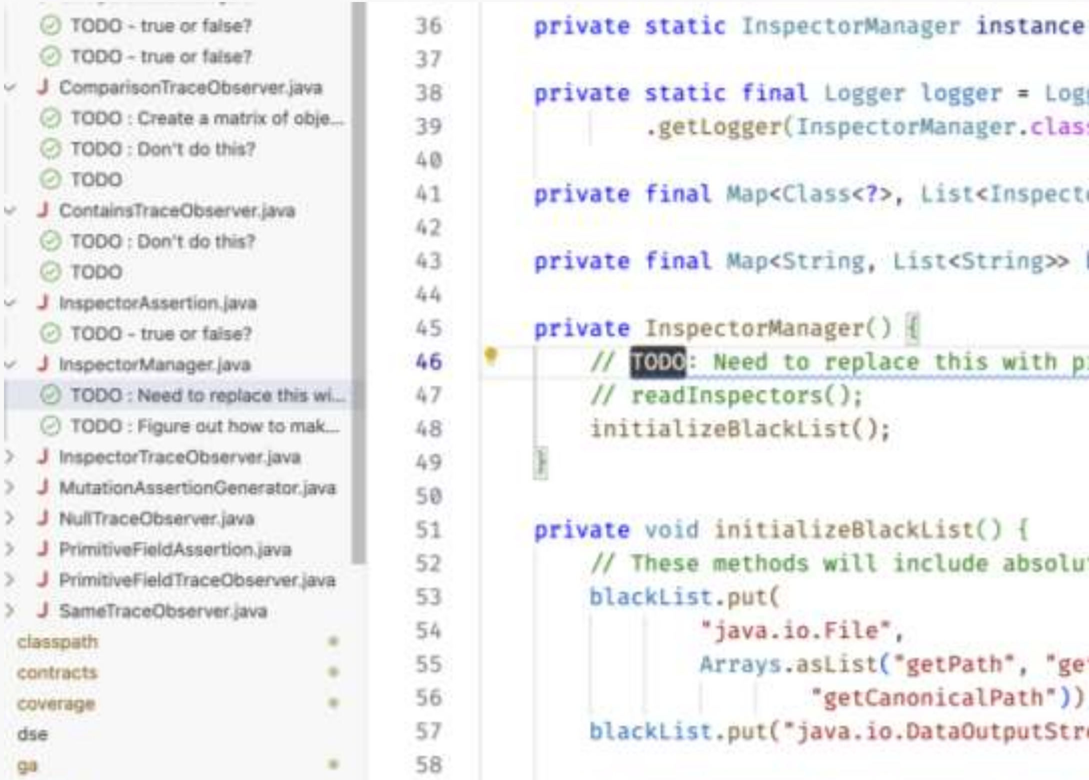
- New version of libraries or frameworks
- New version of languages
- Outdated libraries or frameworks
- Technology becoming obsolete

Types of Technical Debt

- Strategic Debt
  - Known
  - long-term
- Tactical Debt
  - Known
  - Short-term
- Inadvertent Debt
  - Unknown
  - Short/long term
- Incremental Debt
  - Unknown
  - Periodic

Self-Admitted Technical Debt (SATD)

- Intentionally introduced by developers.
- Typically reported in source code comments.
- Sometimes reported in issue trackers.



Quality Attributes affected by Technical Debt

- Understandability
- Changeability
- Extensibility
- Reusability
- Testability
- Reliability
- Performance
- Security
- …

How to manage Technical Debt?

- Increase awareness of technical debt.
  - Train developers
  - Train managers
- Detect and repay technical debt.
  - Identify instances of technical debt
  - Plan to recover from the identified technical debt.
- Prevent accumulation of technical debt.
  - Monitor technical debt
  - Track accumulation and repayment

When to repay technical debt?

Prioritise the repayment efforts:

- What causes the most pain to developers?
- What is the component's risk
  that the technical debt becomes an issue?
- Is the component having technical debt a hot spot,
  meaning that it is changed often.

Pay of high interest technical debt first!

The Boy Scout Rule

"Always leave the campground cleaner than you found it."

For software:

"Always check a module in cleaner than when you checked it out."
– Robert C. Martin (Uncle Bob)

https://unsplash.com/photos/Ppz6b-YUDHw

Repaying Technical Debt: Tools

- Comprehension tools
- Metric tools
- Analysis tools
- Technical debt quantification and visualization tools
- Refactoring tools

Technical Debt
Quantification and Visualization

- Quantifying technical debt and assigning a monetary value would allow informed decision.
- An approximation to quantifying technical debt is to use key metrics that can be measured.
- Trends over key metrics can be interpreted as trends over the accumulation of technical debt and repayment efforts.
- Visualising metrics and their trends can inform stakeholders.

Tools

- Tools usually focus to much on
  what can be identified and measured easily.
- Tools can distract from
  what is important and cannot be measured (easily).
- Treat tools as indicators, not as absolute truth.
- Tools are not aware of your context.

Technical Debt
and
Duplicated Code

FOCUS: TECHNICAL DEBT

IEEE Software 2012, vol 29, no 6.

Managing Technical Debt with the SQALE Method

Jean-Louis Letouzey and Michel Ilkiewicz, Inspearit

// The SQALE (software quality assessment based on life-cycle expectations) method provides guidance for managing the technical debt present in an application's source code. //

2012

Characteristic:
Testability

Requirement:
There are no cloned parts of 100 tokens or more

Remediation microcycle:
Refactor with IDE
and write tests

Remediation function:
20 minutes
per occurrence

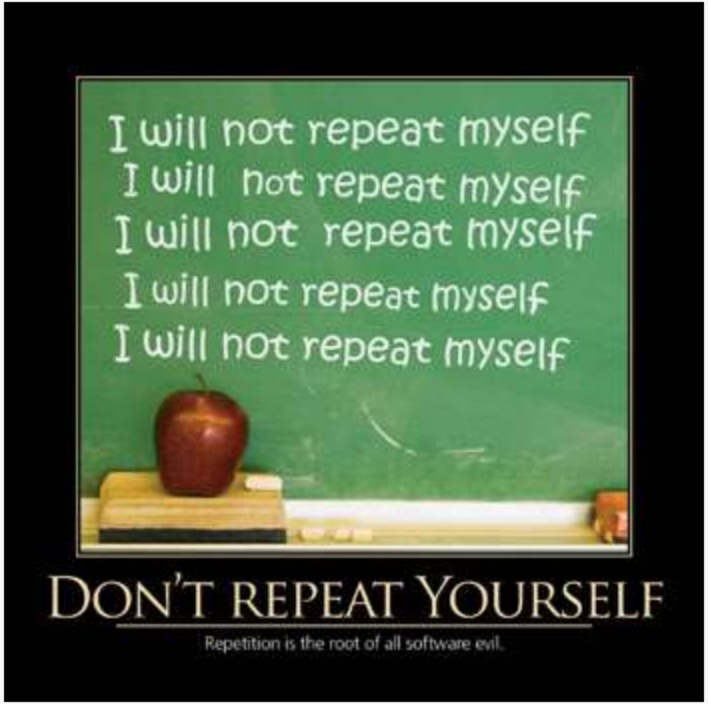
The «SQALE» Analysis Model

2010

Another testability indicator is the presence of duplicated code. Indeed, every piece ~~of duplicated~~ code increases the unit test effort.

Letouzey JL, Coq T. The SQALE analysis model: An analysis model compliant with the representation condition for assessing the quality of software source code.
International Conference on Advances in System Testing and Validation Lifecycle 2010

Don't Repeat Yourself (DRY) Principle



The Don't Repeat Yourself (DRY) principle states that duplication in logic should be eliminated via abstraction;
duplication in process should be eliminated via automation.

http://marchoeijmans.blogspot.co.uk/2012/06/dont-repeat-yourself-dry-principle.html

Find out how the IfSQ Standards are used around the world: search for **"IfSQ Maintainability Rating"**

SPM-3—Copy/Paste Programming

Defect Indicators: A largely similar or identical section of code appears in two or more places in the program or set of programs.

Risks: Having to modify identical code in multiple places:

- increases the likelihood of making slightly different modifications under the mistaken assumption that you have made identical ones,
- increases the time needed to make changes,
- increases maintenance costs in direct proportion to the number of times the code has been copied/pasted.

Assessment:

- If you see a block of code that looks familiar, backtrack through the already assessed code looking for similar blocks. If the blocks could be implemented as a (parameterised) subroutine, mark the second and subsequent blocks.

Remedy: Isolate a single copy of the code into a separate program (e.g., method, function, subroutine) and reuse it by calling it from the places in which it was used.

References:

- Why You Should Use Routines, Routinely (Steve McConnell), 1998.

Research Findings:

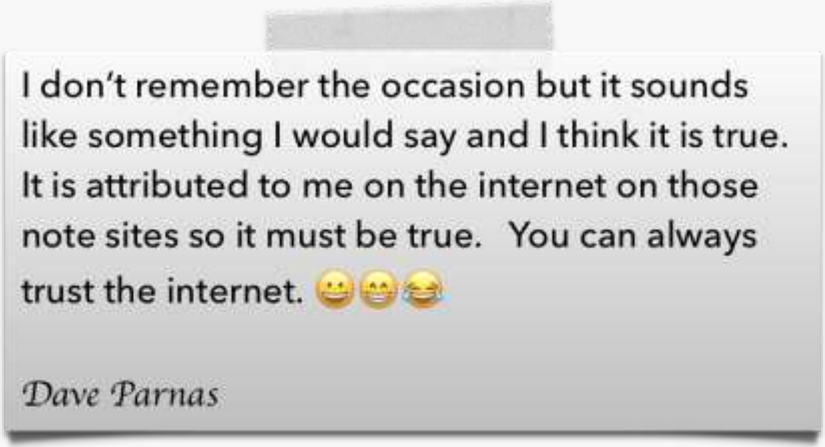
- Don't Repeat Yourself (DRY):
  The DRY principle: Don't Repeat Yourself,
- Repetitive code indicates poor design:
  Copy and Paste is a design error.

Famous Quote

1998

David Parnas says that if you use copy and paste while you’re coding, you’re probably committing a design error.

S. McConnell. 1998. Why you should use routines...routinely. IEEE Softw. 15, 4



Duplicated Code as Technical Debt Examples

2011

For example, if, on average, 3-star and 4-star system snapshots have 10% and 3% duplicated code respectively, then it is inferred that the amount of code that needs to be changed/refactored to improve the level of quality from 3-star to 4-star would be 7% of the total LOC.

Nugroho A, Visser J, Kuipers T. An empirical model of technical debt and interest.
International Workshop on Managing Technical Debt 2011

Duplicated Code as Technical Debt Examples

2012

the cost of writing the system from the ground up [5]. The cause of this unfortunate situation is usually less visible: The internal quality of the system design is declining [6], and duplicated code, overly complex methods, noncohesive classes, and long parameter lists are just a few signs of this decline [7]. These issues, and many others, are usually

Marinescu R. Assessing technical debt by identifying design flaws in software systems. IBM Journal of Research and Development. 2012 56(5).

Duplicated Code as Technical Debt Examples

2014

$$Debt = duplication + violations + comments + coverage + complexity + design \quad (1)$$

$$duplication = cost\_to\_fix\_one\_block * duplicated\_blocks \quad (2)$$

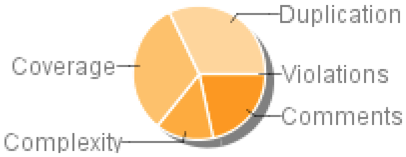
Griffith I, Reimanis D, Izurieta C, Codabux Z, Deo A, Williams B.
The correspondence between software quality models and technical debt estimation approaches.
International Workshop on Managing Technical Debt 2014

2009

Technical Debt

$ 14'605

29 man days



The current version of the plugin is 0.2 and uses the following formula to calculate the debt :

Debt(in man days) = cost_to_fix_duplications + cost_to_fix_violations + cost_to_comment_public_API + cost_to_fix_uncovered_complexity + cost_to_bring_complexity_below_threshold

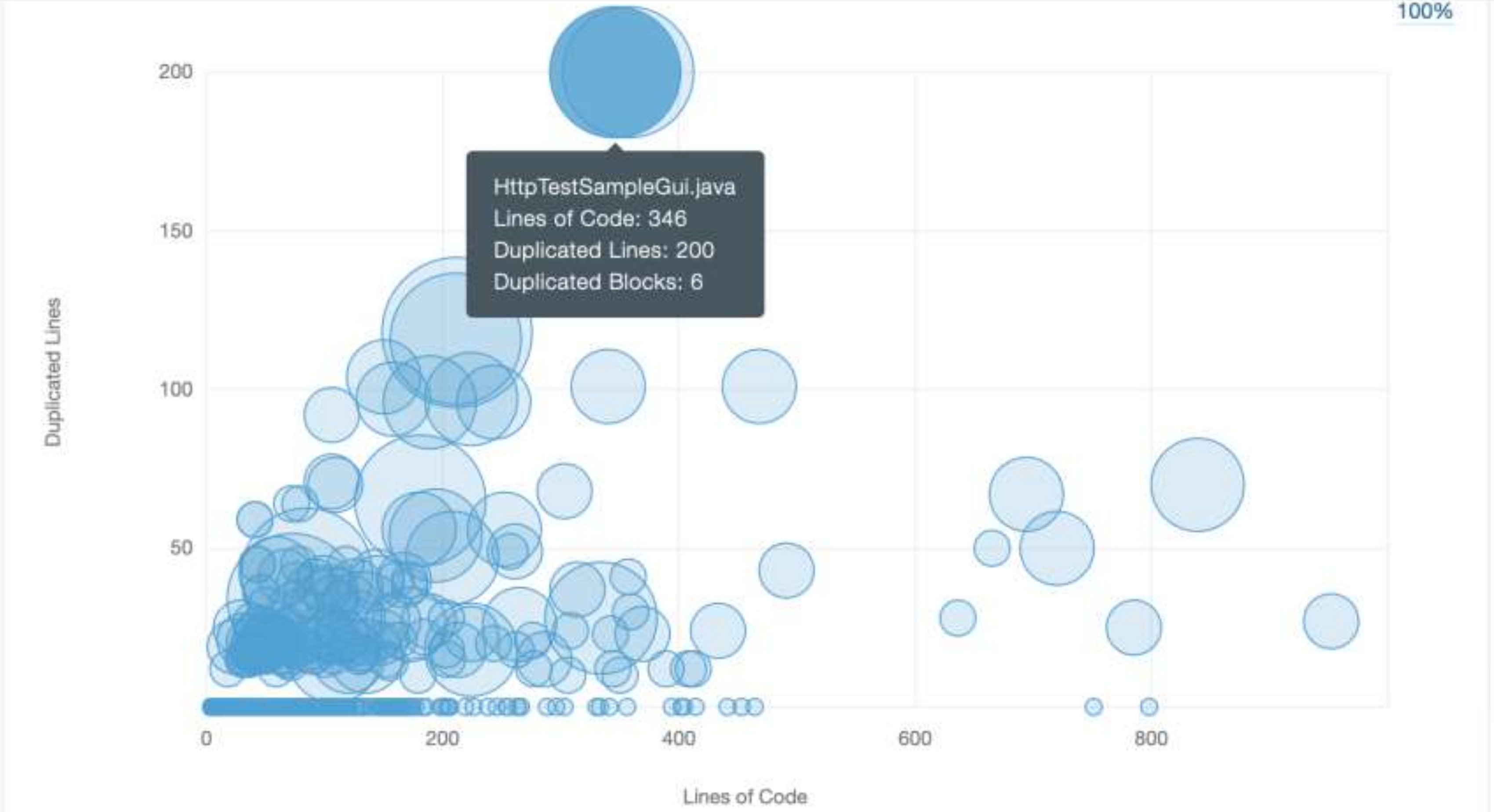
Where :

Duplications = cost_to_fix_one_block * duplicated_blocks

O. Gaudin, "Evaluate your technical debt with Sonar," Sonar, 2009. http://www.sonarqube.org/evaluate-your-technical-debt-with-sonar/

JMeter

500 / 1,386 files



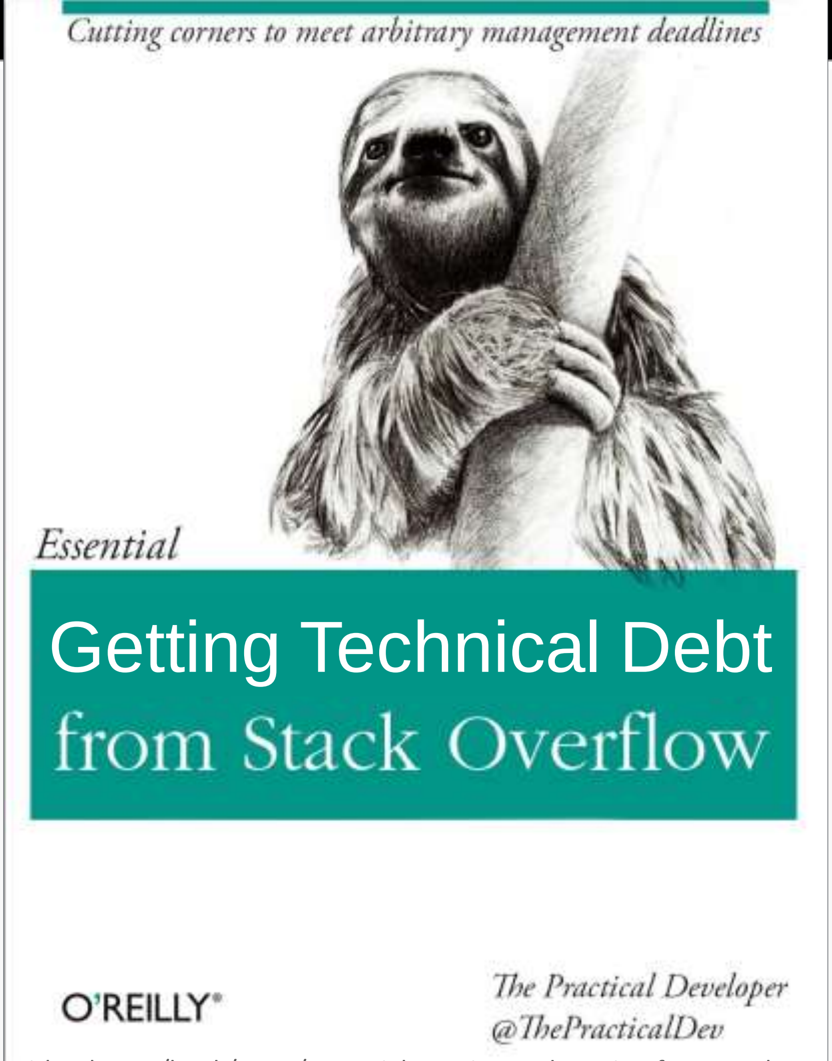
Empir Software Eng
DOI 10.1007/s10664-008-9076-6

2006

"Cloning considered harmful" considered harmful: patterns of cloning in software

Cory J. Kapser · Michael W. Godfrey

We found that as many as 71% of the clones could be considered to have a positive impact on the maintainability of the software system.



https://www.gitbook.com/book/tra38/essential-copying-and-pasting-from-stack-overflow/details

Repaying Technical Debt through Refactoring

Repaying of Technical Debt (1)

- Code Debt:
  - Fix coding style violations
  - Fix static analysis tools violations
  - Code-level refactoring
- Design Debt:
  - Design-level refactoring
- Test Debt:
  - Improve tests (COMP0103)
- Documentation Debt:
  - Improve documentation

Refactoring

- Refactoring is the practice of changing a system or component design without changing its behavior.
- Refactor to improve the quality of software.
- Refactoring requires a high-quality test suite to check that the refactoring operations did not break the software.

Example Refactoring:
Remove Duplication (1)

- Two (or more) methods of the same class
  contain the same expression.
- Apply the **Extract Method** refactoring:
  Turn the fragment into a method
  whose name explains the purpose of the method.
- Replace all usages of the expression
  by a call to the new method.

Example Refactoring: Remove Duplication (2)

- Two (or more) methods of sibling subclasses contain the same expression.
- Apply the **Extract Method** refactoring in both classes.
- Apply the Pull Up Method refactoring: Move them to the superclass.

A Catalogue of Refactorings

- Martin Fowler (1999) “Refactoring: Improving the Design of Existing Code”.
- https://www.refactoring.com

When to refactor

- When adding new features
- When fixing bugs
- During code review

Refactoring Best Practices

- Do not mix refactoring operations with behavior changes.
- Either refactor than change behavior or change behavior the refactor.

Inheriting Technical Debt: Legacy Code

Legacy Code

- Usually older code (but not always)
- Code without sufficient communication artefacts to explain its intent.

Alternative Communication Artefacts

- Tests are very effective communication artefacts
- Readable code
- Etc.

Characteristics of Legacy Code

- Poor architecture
- Non-uniform coding styles
- Poor or non-existing documentation
- Mythical “oral” documentation
- No tests (or minimal test coverage)

Changing Legacy Code

- Most development activities in legacy code need to preserve behavior.
- Without tests there is a high risk to make mistakes introducing unintentional changes to behavior.

Before Making a Change

- Plan the change
- Identify the change points
- Understand the code around the change points.
- Add tests to cover the change points.
- Refactor the change points
  - To allow testing
  - To improve understanding
- Add tests for the changed behavior (see test-driven development)

The Refactoring Dilemma

- To be able to refactor, one should have tests.
- To add tests, one often has to refactor.

Concepts (1/2)

- Lehman's laws is a series of "laws" with respect to software evolution.
- Technical debt is the debt that accrues when you knowingly or unknowingly make wrong or non-optimal design decisions.
- The metaphor of technical debt is used to explain the impact of cutting corners. Although being coined as a metaphor, it has become an important concept in industry.

Concepts (2/2)

- It is hard to quantify technical debt, but software measurements can be used as an approximation.
- Refactoring is the practice of changing a system or component design without changing its behavior.
- Refactoring is needed when working with legacy code, to improve the understanding and to allow testing.
- Legacy code usually has insufficient tests and documentation.