Quality-Assuring Software Engineering Methods

Jacek Galowicz

FH Münster

WS 2022/23

Section Content

Dates

- Blockveranstaltung: Fr 13:00 - 18:00, Sa 09:00 - 17:00
  - 04.+05.11.2022
  - 11.+12.11.2022
  - 09.+10.12.2022
  - 27.+28.01.2022
- Make sure you are enrolled in the ILIAS course for this class

Exam

- You manage exam registration with the Prüfungsamt
- Exam dates will be determined towards end of class
- Oral exam: 45 Minutes Q&A
- Content & style of the exam:
  - Distinction and evaluation of all big picture terms
  - Purpose, effort and limitation of all mentioned solutions/technologies, ability to discuss them against each other.
  - Ability to explain the value of technologies for businesses in an economic context
  - I'll try to focus on questions that cannot be answered with Auswendiglernerei
  - Slides will be shared for your personal studies but don't contain the *Nähkästchen-Stories*, which help understanding the big picture

Technical Prerequisites for this Class

- We work on Linux with Network and Virtualization
- If you run Linux in a VM, you might need Nesting for some exercises
- Installed software:
  - KVM kernel module
  - nix: `https://nixos.org/download.html` (Please don't use your distro's native packages)
  - Docker (Use your distro's native packages)
- Toolchains etc. for exercises are automatically downloaded using nix.
- nix also works on MacOS and partly WSL. For time reasons: Not supported.

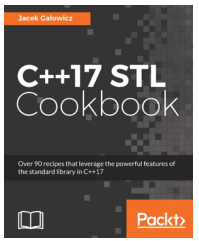
About the Lecturer - Jacek Galowicz

Born 1987, grew up in Cologne, lives in Braunschweig
Electrical Engineer, focus on technical computer science
Strengths: Software Design/Architecture, Automation of complex deployments and testing, C++, Haskell

2008-2013	Electrical Eng Degree
2012	Intern, Intel Labs
2014	SW Engineer, FireEye Inc.
2017	Co-Founder, Cyberus Technology GmbH
2023	Freelancer

RWTH AACHEN UNIVERSITY
intel
FireEye
CYBERUS TECHNOLOGY

Author, Editor



Content: The Big Picture

- What is Software Quality?
- SW Engineering, State of the Art:
  - Terms: SW Quality, Regression, Technical Debt, Reproducibility
  - SWE in comparison to other engineering disciplines
- Processes:
  - How do SWEs work in companies?
  - What enables them? What limits them?
  - What's good/bad for companies (economic perspective)?
  - What's good/bad for the SWE's carreer?
  - We don't look at industrial ISO-Norm XYZ123

Class Structure

- SW Engineering, Solo
  - Version Control Systems
  - Test-Driven Development
  - Sanitizers
  - Static Analysis
  - Fuzzing
  - Integration Tests
- SW Engineering, Collaborative
  - Synchronization of Distributed Repositories
  - Workflows with Pull-Requests, Code-Review
- Reproducible Builds
  - Isolation, explicit Dependencies
  - Complex Toolchains and Setup
  - Solutions: Docker, Nix
- Continuous Integration
  - Central Automation of Build, Tests, and Integration Tests

Knowledge-Goals of This Class

- State of the art of:
  - Testing
  - Deployment
  - Reproducibility
- Typical practical problems from different perspectives
  - Technical
  - Incompetence, Company Organization, Economy
- Ability to convey the value to team / project / product / company
  - In discussions with colleagues and management
  - Distinction between *good* and *bad* processes and organizations

About You

- What's your programming experience?
  - Which programming languages?
  - Which domains?
  - Which operating systems and hardware platforms?
- How experienced are you in the team?
  - Job
  - Hobby Projects
  - Open Source Contributions
- What do you expect from this class?
- What do you expect from your work life?

Last Notes Before We Begin

- Class-room atmosphere
  - Talking without personal feedback makes it difficult to create the best possible education experience
  - Please keep cams activated, let's make this as personal as possible
  - If i go through the slides too quick or talk too early while you are still reading too often, please comment
  - Generally, just shout: I am happy to provide more information to anything
- Discussions
  - There's a lot of technical *opinion* in these slides.
  - Let's discuss many things. If you have questions or insights, please activate your microphone and just shout!
- Let's have frequent breaks:
  - Bio-break every 45 minutes
  - 10-15 minute break every 90 minutes
  - I'll try to set timers, but if that didn't work, please shout!

Section Content

What is Software Quality?

- What is Software Quality?

What is Software Quality?

- What is Software Quality?
  - Usability
  - Correctness
  - Stability
  - Safety & Security
  - Performance
  - Memory Consumption
  - Maintainability

What is Software Quality?

- What is Software Quality?
  - Usability
  - Correctness
  - Stability
  - Safety & Security
  - Performance
  - Memory Consumption
  - Maintainability
- What are the consequences for companies if their products look bad on at least one of these points?

What is Software Quality?

- What is Software Quality?
  - Usability
  - Correctness
  - Stability
  - Safety & Security
  - Performance
  - Memory Consumption
  - Maintainability
- What are the consequences for companies if their products look bad on at least one of these points?
- $\rightarrow$ As SWEs, how do we achieve all this?

Who bets their annual salary that their code is bug free?

Who bets their annual salary that their code is bug free?

→ Your employer does! And the outcome may affect your employment too.

Special Case: Maintainability

- Why should we care? It cannot be perceived by the customer anyway.

Special Case: Maintainability

- Why should we care? It cannot be perceived by the customer anyway.
- It could affect the speed at which we can fix bugs, close security vulnerabilities or add new features.

Special Case: Maintainability

- Why should we care? It cannot be perceived by the customer anyway.
- It could affect the speed at which we can fix bugs, close security vulnerabilities or add new features.
- To measure maintainability we have to answer the following question:

> How simple is it to fix bugs or add features to a product?

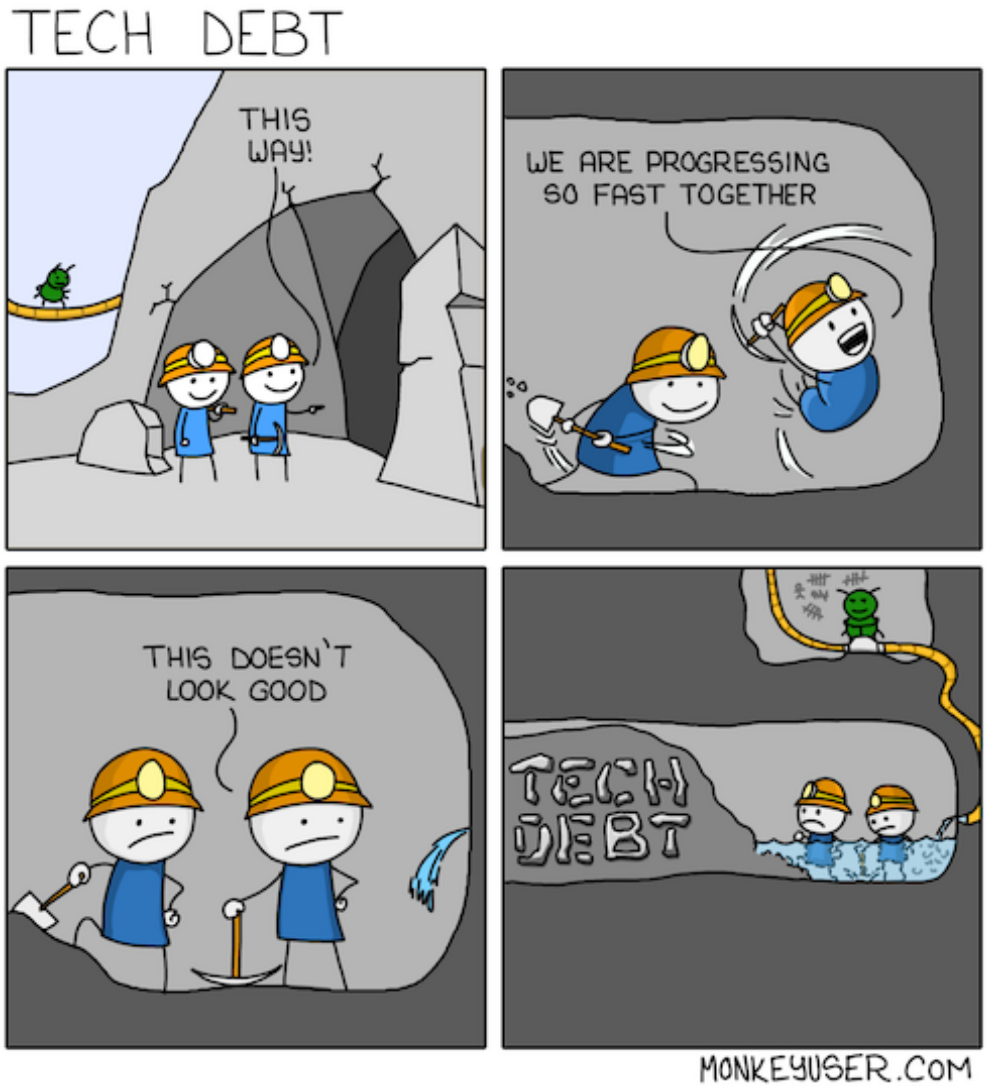
Special Case: Maintainability

- Why should we care? It cannot be perceived by the customer anyway.
- It could affect the speed at which we can fix bugs, close security vulnerabilities or add new features.
- To measure maintainability we have to answer the following question:

How simple is it to fix bugs or add features to a product?

If it is not simple: **Why?**

Technical Debt



Source: `https://www.monkeyuser.com/2018/tech-debt/`

Technical Debt

A Definition of Tech Debt

Technical debt is a concept in programming that reflects the extra development work that arises when code that is easy to implement in the short run is used instead of applying the best overall solution.

Source: techopedia.com

Tech Debt: Bad Tests Example

Oracle DB Testing Rant on Hacker News

Oracle Database 12.2. It is close to 25 million lines of C code.

What an unimaginable horror! You can't change a single line of code in the product without breaking 1000s of existing tests. Generations of programmers have worked on that code under difficult deadlines and filled the code with all kinds of crap.

Very complex pieces of logic, memory management, context switching, etc. are all held together with thousands of flags. The whole code is ridden with mysterious macros that one cannot decipher without picking a notebook and expanding relevant parts of the macros by hand. It can take a day to two days to really understand what a macro does.

...

Read the full post: `https://news.ycombinator.com/item?id=18442941`

What *Exactly* is Tech Debt?

- Deferring documentation
- Failure to refactor code and architecture
- Lack of versioning, backups, build tools, CI
- Lack of unit tests and regression tests
- Lack of coding standards / discipline
- Disregard of compiler warnings and static code analysis
- Unclean interfaces and dependencies
- Unclear / difficult to reproduce definition of dependencies

Reasons for Tech Debt

Many teams accumulate tech debt with high rates, which is partly unconcious, partly enforced by real-life limitations:

- Incoming deadlines
- Unexpectedly long project/product life time
- Tech debt avoidance is not part of any programmer's education
- *Deadline-Veteranism* that results in Company/Cultural incentives
  - A *Tactical Tornado* is a person who saves the team...
  - ...but piles up a lot of tech debt while doing that
- Process-related reasons
  - The road to hell is paved with good intentions
- Incompetence, Ignorance, Arrogance
- ...

Technical Debt Reduction in Reality

In a nutshell: Reduction of tech debt is *not sexy*.

Technical Debt Reduction in Reality

In a nutshell: Reduction of tech debt is *not sexy*.

- Brings no new features

Technical Debt Reduction in Reality

In a nutshell: Reduction of tech debt is *not sexy*.

- Brings no new features
- Fixes no bugs

Technical Debt Reduction in Reality

In a nutshell: Reduction of tech debt is *not sexy*.

- Brings no new features
- Fixes no bugs
- Functionality stays unchanged

Technical Debt Reduction in Reality

In a nutshell: Reduction of tech debt is *not sexy*.

- Brings no new features
- Fixes no bugs
- Functionality stays unchanged
- Example: Google's promotion process

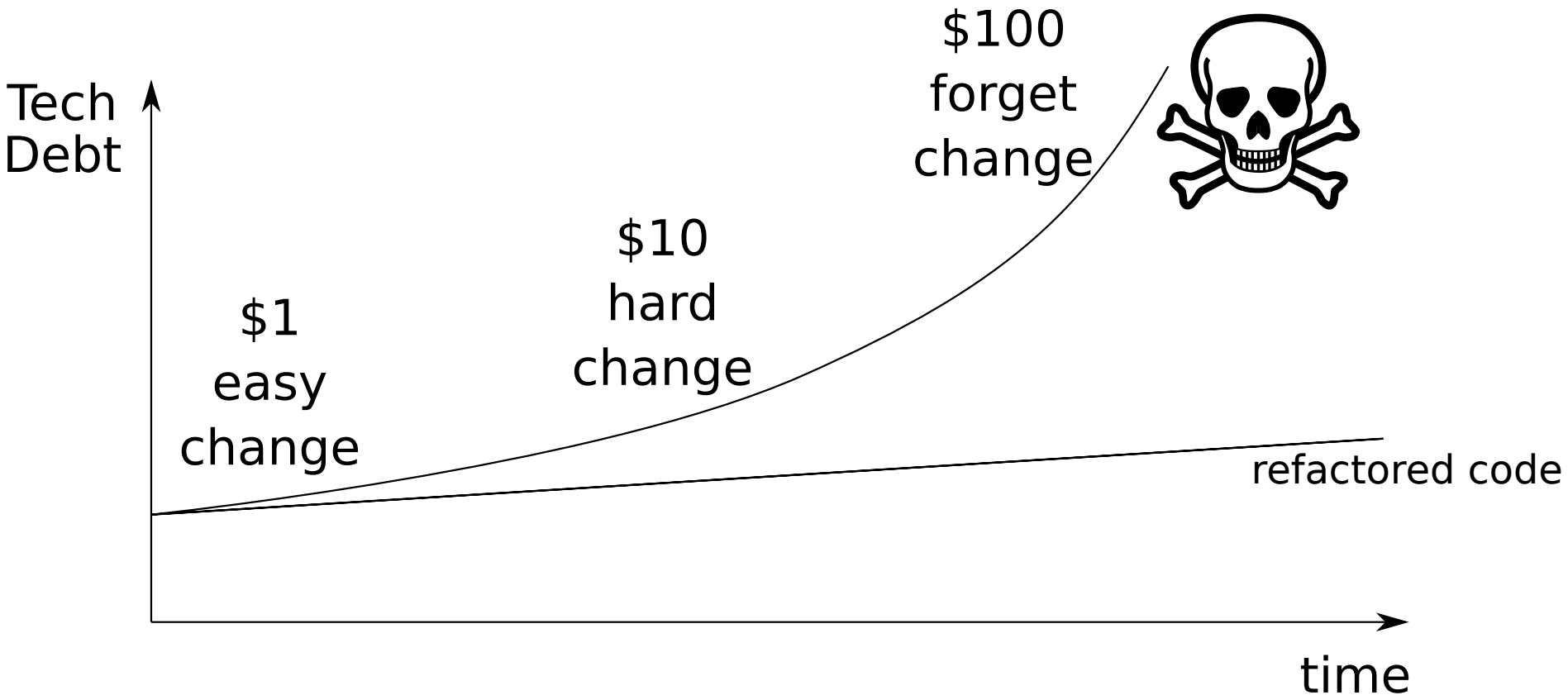
Correctness and Maintainability as Top Goals

- Bugs lead to
  - Crashes, downtime
  - Security vulnerabilities
  - Injuries/Deaths, financial/material loss

Correctness and Maintainability as Top Goals

- Bugs lead to
  - Crashes, downtime
  - Security vulnerabilities
  - Injuries/Deaths, financial/material loss
- Tech debt leads to
  - Effort&time burden for product development
  - Inability to adapt/change
  - Inability to quickly fix bugs
  - Inability to tell if a code fix corrects more bugs than it causes

Exponentially Rising Change Costs/Time with Tech Debt



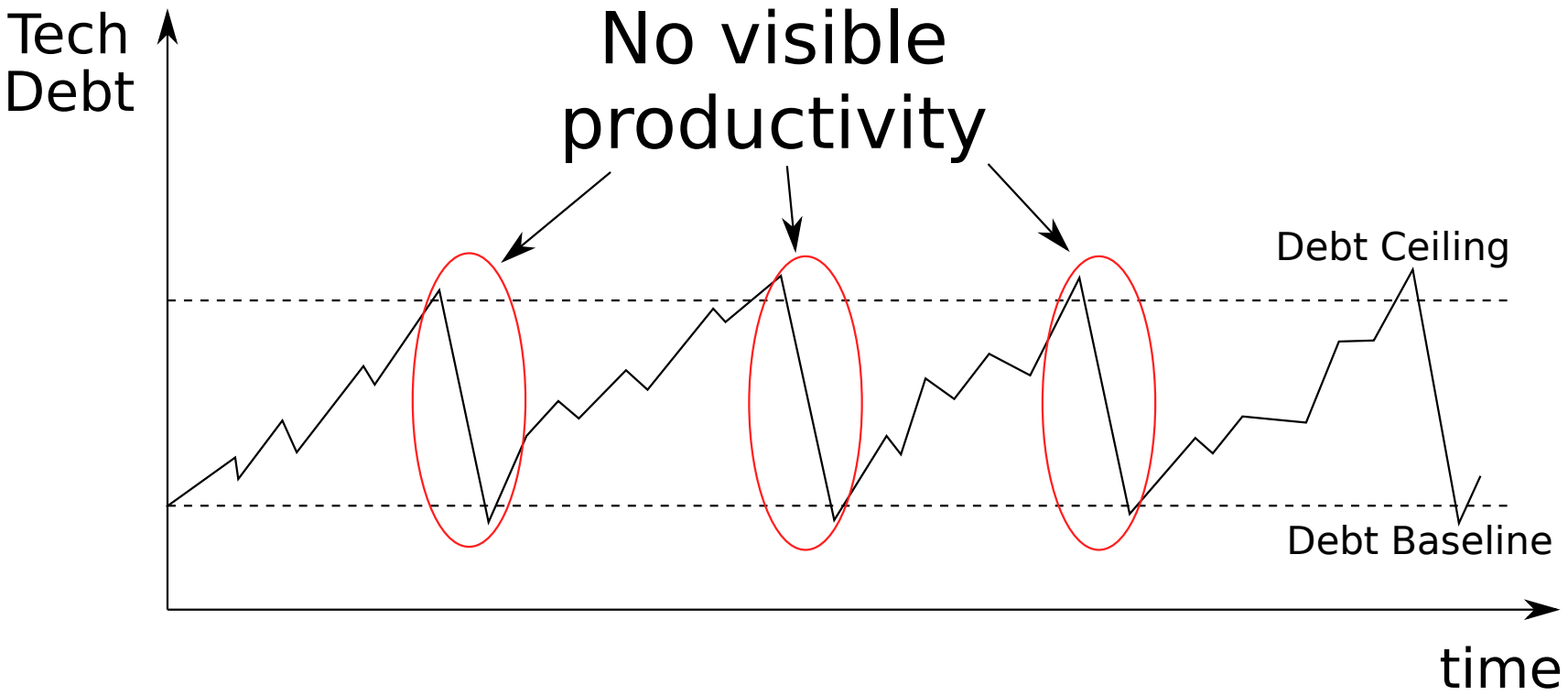
Different Historical Reasons for Tech Debt

	Reckless	*Reasonable*
Deliberate	"We have no time for big design work."	"We need to release now and figure out how to deal with the consequences later"
Unconcious	"What are software layers?"	"In retrospect it would have made sense to..."

Customer Visibility Defined Sense of Value

	visible (for customers)	*invisible (for customers)*
positive value	Feature	Architecture
negative value	Bug	Tech Debt

Tech Debt Reduction Looks Unproductive



Section Content

Why do we need Version Control Systems?

The basic needs are:

Why do we need Version Control Systems?

The basic needs are:

- Logging of changes
- Recoverability of older project versions
- Coordination of different changes made by various developers at the same time

git

2005 released, free, distributed version management system for source code. Inventor: Linus Torvalds

git

2005 released, free, distributed version management system for source code. Inventor: Linus Torvalds

"git" is not an abbreviation, but:

"I'm an egotistical bastard, and I name all my projects after myself. First 'Linux', now 'Git'.", Linus Torvalds (source)

git

2005 released, free, distributed version management system for source code. Inventor: Linus Torvalds

"git" is not an abbreviation, but:

"I'm an egotistical bastard, and I name all my projects after myself. First 'Linux', now 'Git'.", Linus Torvalds (source)

→ We'll learn how git works and how it organizes data internally.

git Selling Points & Features

- Decentralized, distributed development model
- Scales extremely well with thousands of developers
- High performance
- Integrity and trust
  - Accountability through clear association between dev and changes
  - Immutability
  - "Blockchain!"
- Atomic transactions
- Enables branched development models
- Free open source software

git Literature

git is an extremely useful and important tool. It will be essential for your career to understand it. Understanding git will also help you in the future with upcoming, more advanced VC systems.

Literature:

- Documentation, Tutorials, etc.: `https://git-scm.com/doc`
- Free git book: `https://git-scm.com/book/en/v2`
- Book "Version Control with Git", Jon Loeliger, Matthew McCullough, August 2012 at O'Reilly Media

git: Basic Concepts

- Repository
- Index
- Content-Addressable Names using **Hashes**
- git object types:
  - Blob (*Binary large object*)
  - Tree
  - Commit
  - Tag

Hash Functions

$h : K \to S$ Mapping from keys to hashes

$|K| \geq |S|$

Hash Functions

$h : K \to S$ Mapping from keys to hashes

$|K| \geq |S|$

$K' \subseteq K$ Actually used keys

$S' := \{h(k) | k \in K'\}$ Actually used hashes

$\beta = \frac{|S'|}{|S|}$ Occupancy factor

Hash Functions

$h : K \to S$ Mapping from keys to hashes

$|K| \geq |S|$

$K' \subseteq K$ Actually used keys

$S' := \{h(k) | k \in K'\}$ Actually used hashes

$\beta = \frac{|S'|}{|S|}$ Occupancy factor

$k \neq k' \wedge h(k) = h(k')$ Collision!

Criteria for Good Hash Functions

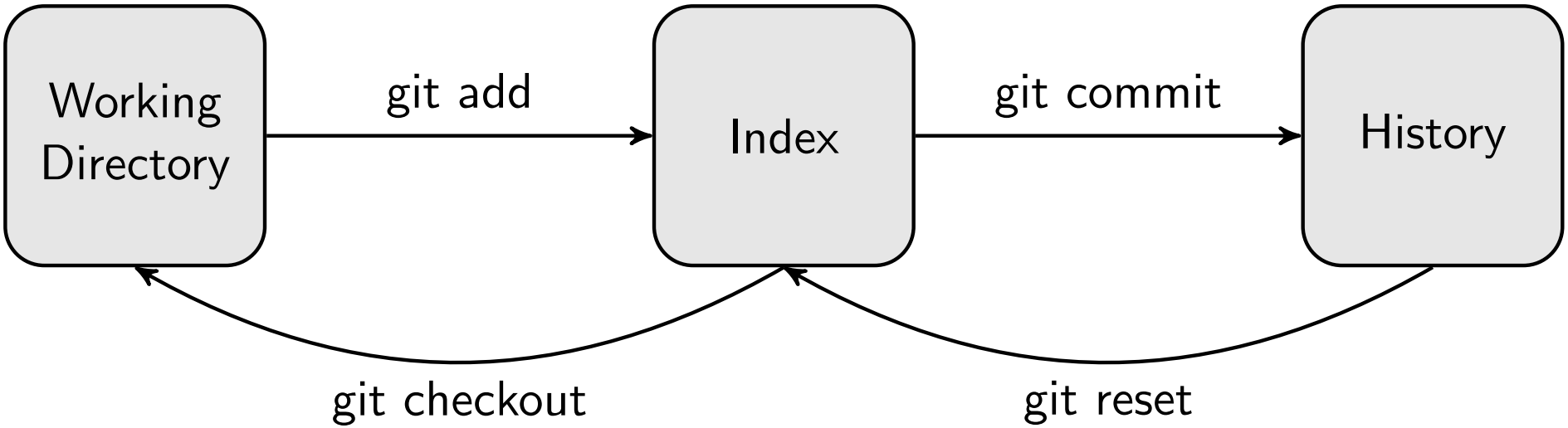
- Low likelihood of collisions
- The memory requirement for the hash value is significantly smaller than that of the key
- Chaos / avalanche effect: Small differences between keys should lead to completely different hash values
- Surjectivity - Every possible hash value should be able to occur
- Efficiency

Real-Life Hash Collision

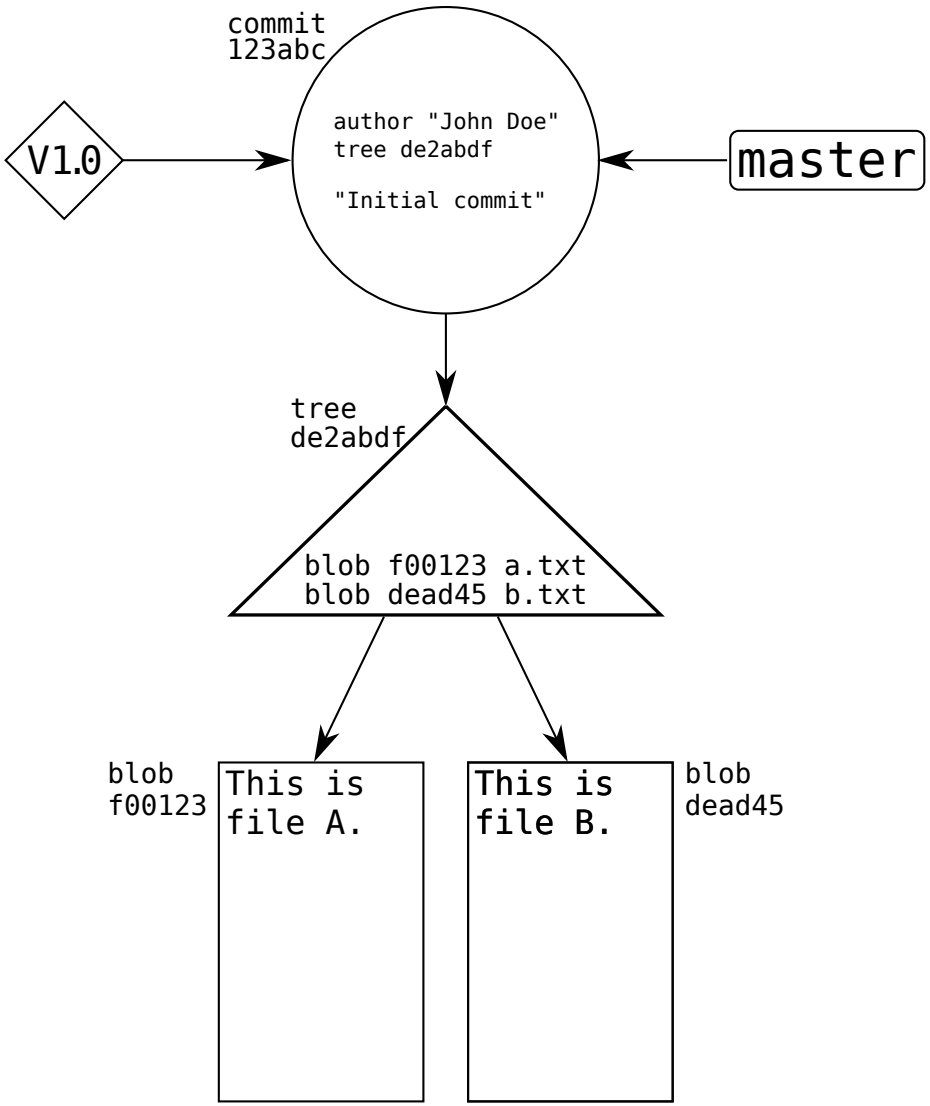
Did twitter have a hash collision? (Source: hillreporter.com)

The Index/Staging Area

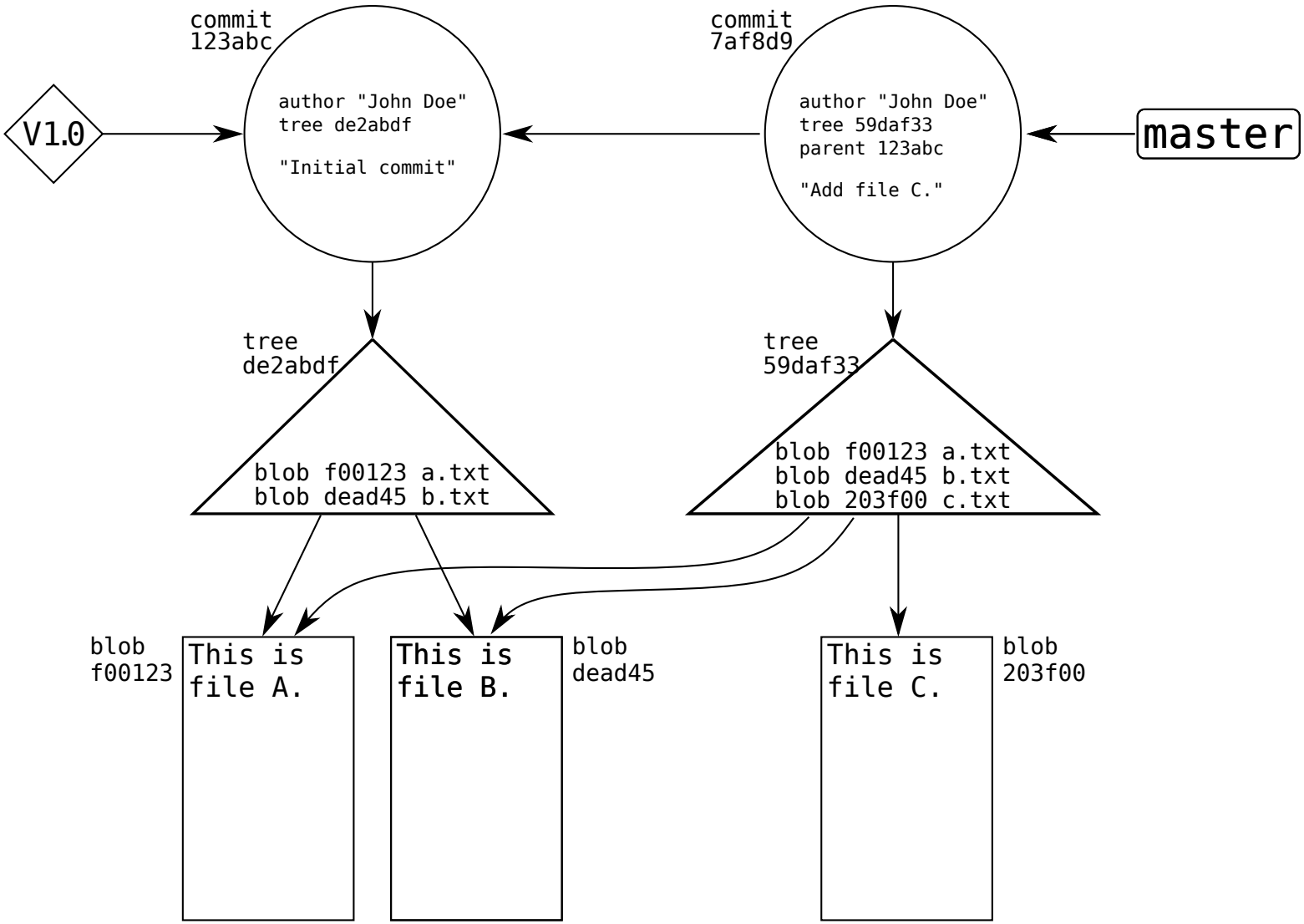
In order to commit changes, they need to be added to the *index*. Only the index's content will be committed. This way, different changes can be split up into multiple commits.



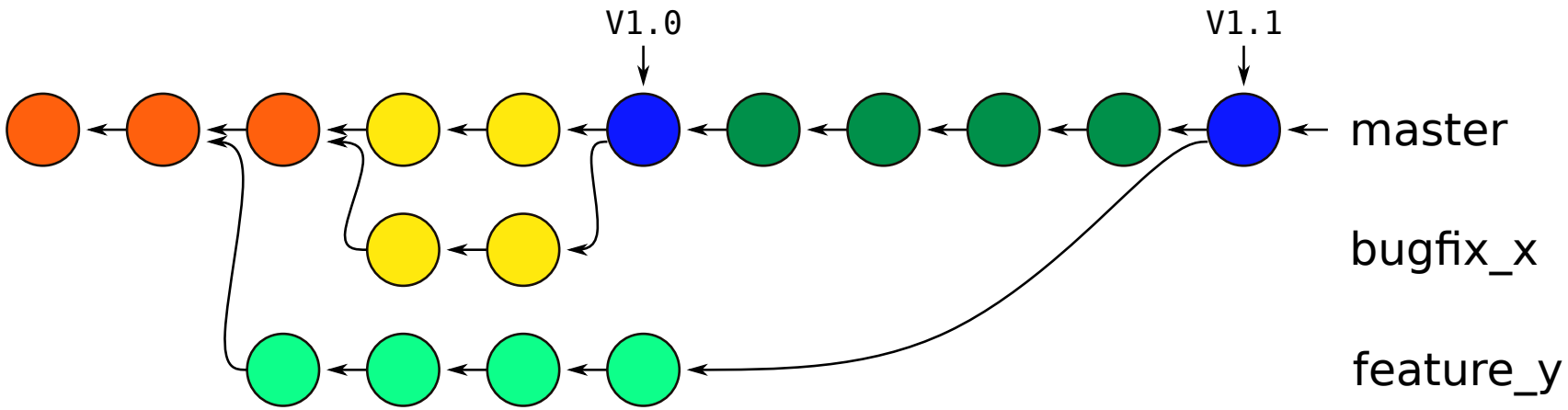
git: Internal Representation of Program Versions



git: Internal Representation of Program Versions



git Merge Workflow



git: Installation

Start a nix shell with git

```
$ nix-shell -p git
```

Install using nix

```
$ nix-env -i git
```

Install in Ubuntu

```
$ apt install git
```

git: Setup

Set up author/committer identity

```
$ git config --global user.name "John Doe"
$ git config --global user.email "john@doe.com"
```

git: Basic Commands

Command	Effect
`git init`	Initialize a project as a git repo
`git add`	Apply current state in working folder to index
`git commit`	Seal the current index in a new commit
`git checkout`	Check out or create another branch
`git merge`	Merge another branch with the current one
`git rebase`	Base the history of branch on other commit

git bisect

If a number of commits introduce a bug somewhere, but it is not clear which one - `git bisect` can perform a *binary search*.

```
$ git bisect start HEAD HEAD~20
$ git bisect run sh -c "make && make test"
$ git bisect reset
```

git bisect

If a number of commits introduce a bug somewhere, but it is not clear which one - `git bisect` can perform a *binary search*.

```
$ git bisect start HEAD HEAD~20
$ git bisect run sh -c "make && make test"
$ git bisect reset
```

Question

Which circumstances will break this workflow?

git bisect

If a number of commits introduce a bug somewhere, but it is not clear which one - `git bisect` can perform a *binary search*.

```
$ git bisect start HEAD HEAD~20
$ git bisect run sh -c "make && make test"
$ git bisect reset
```

Question

Which circumstances will break this workflow?

→ Commits which break compilation/tests disable git bisect!

Test-Driven Development

Any complex program can be viewed as a composition of many less complex individual programs.

Test-Driven Development

Any complex program can be viewed as a composition of many less complex individual programs.

Problem

Unintentional introduction of a bug can lead to instability of the whole - but does not necessarily have to be immediately visible. Or it is visible, but it is still unclear what the cause is.

Test-Driven Development

Any complex program can be viewed as a composition of many less complex individual programs.

Problem

Unintentional introduction of a bug can lead to instability of the whole - but does not necessarily have to be immediately visible. Or it is visible, but it is still unclear what the cause is.

Idea

Development of programs (unit tests) that run every minimal part program with valid and invalid entries and check whether it works correctly.

Test-Driven Development

Any complex program can be viewed as a composition of many less complex individual programs.

Problem

Unintentional introduction of a bug can lead to instability of the whole - but does not necessarily have to be immediately visible. Or it is visible, but it is still unclear what the cause is.

Idea

Development of programs (unit tests) that run every minimal part program with valid and invalid entries and check whether it works correctly.

Result

Programmers can run all unit tests at lightning speed before each commit and thus be sure that they have not broken anything that worked previously. $\rightarrow$ **Safety net**!

Kent Beck About Test-Driven Development

Off the top of my head, here are some distinctions:

- *How many code paths are executed? A unit test will typically execute very few code paths, functional and integration testing many.*
- *If a test fails, how many things could be broken? For a unit test, the ideal answer is 1. Functional and integration tests the answer could be large.*
- *How long do they take to execute? Unit tests execute quickly, functional and integration tests relatively slowly (in general).*
- *Whose perspective is used to read the test? Unit tests are read by programmers. Functional tests are read by domain experts. Integration tests are read by architects (those with a broad understanding of the design of the system).*

Source: `https://www.quora.com/What-is-the-difference-between-unit-testing-functional-testing-and-integration-testing/answer/Kent-Beck`

Test-Driven Development Basic Rules

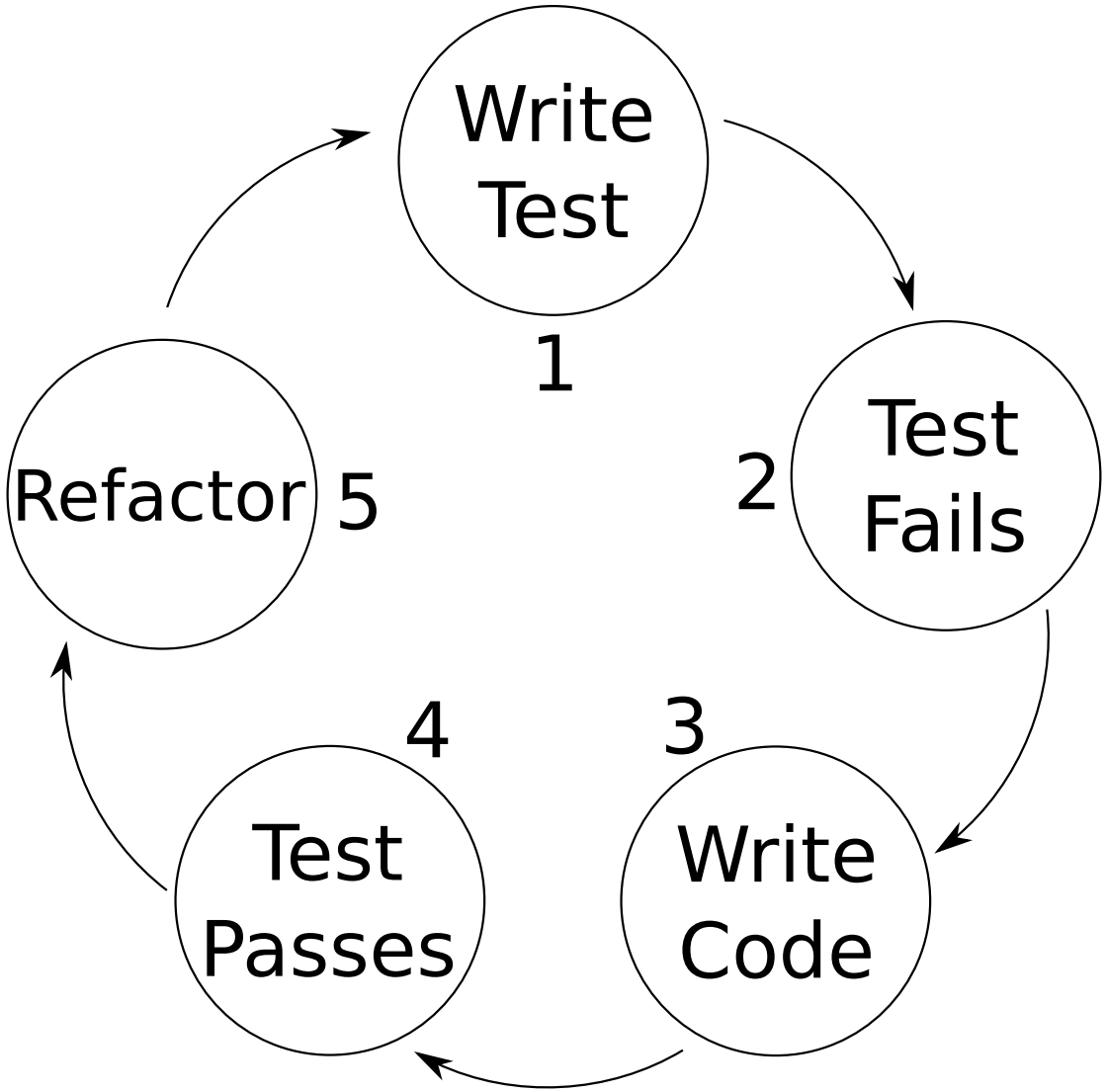
TDD can be and is in part dogmatic, according to the original and fundamental rules of Kent Beck:

Basic Rules of TDD

- Never write a single line of code unless you have a failing automated test.
- Eliminate duplication.

Literature: Test Driven Development by Example, Kent Beck, Addison-Wesley, ISBN 0-321-14653-0

TDD Workflow



General Advantages of Unit Tests

- Atomic unit tests tend to force programmers to design modular components in a structured manner
- Security against regression: Functionality that was already proven to be correct by UTs can no longer become buggy without being noticed when executing the UTs.
- UTs are an enabler for refactoring of code
- Good UTs have documentary character
- UTs can reveal unwanted architecture dependencies

Test Atomicity

Atomic Test Cases increase the value of your unit tests:

- Clearer error identification
- Lower complexity of test code

Test Complexity

If the complexity of

Unit Test Example: Email-Address Checker

Unit Test Exercise

What is a valid email address?

Valid:

- `foo@bar.com`
- `foo@bar.de`
- `a@b.job`
- `...`

Invalid:

- `foo.com`
- `foo@.com`
- `foo@com`
- `@foo.com`
- `!!!@???.~`
- `...`

Download Link

`https://qasm.de/downloads/unit-test-example.zip`

Generating Test Cases from Spec

In the last exercise we experienced that it is often hard to come up with a really *exhaustive* list of test cases.

Idea

What if we could specify how a correct implementation *behaves* and let the computer check our code against many inputs if it adheres to the spec?

Generating Test Cases from Spec

In the last exercise we experienced that it is often hard to come up with a really *exhaustive* list of test cases.

Idea

What if we could specify how a correct implementation *behaves* and let the computer check our code against many inputs if it adheres to the spec?

Formal Methods

This topic overlaps with program/algorithm verification, which is a very deep topic on its own.

$\rightarrow$ In this class, let's simply have a look at some library that helps us with automatic code testing: *Haskell Quickcheck*

Haskell Quickcheck

`https://hackage.haskell.org/package/QuickCheck`

Quickcheck Documentation Intro

QuickCheck is a library for random testing of program properties. The programmer provides a **specification** of the program, in the form of **properties** which functions should satisfy, and QuickCheck then tests that the properties hold in a large number of **randomly generated cases**. Specifications are expressed in Haskell, using combinators provided by QuickCheck. QuickCheck provides combinators to define properties, observe the distribution of test data, and define **test data generators**.

$\rightarrow$ similar libraries exist for different programming languages.

Complex Real-Life Quickcheck Example (1)

Real-Life industry context:
`https://blog.galowicz.de/2017/07/02/order2_iterator/`

Half-open interval

$$I = [a, e),\ a, e \in \mathbb{Z}^0$$
$$\forall a \leq e : a \in I$$
$$e \notin I$$

Capability Range Descriptor

$$\begin{aligned} c &= (a, o) \\ e &= a + 2^o \\ I_c &= \{c_1, ..., c_n\} \end{aligned}$$

Rules

1. $\forall i \in I_c : \quad a_i \mod 2^{o_i} \stackrel{!}{=} 0$
2. $\forall i \in I_c : \quad a_i + 2^{o_i} \stackrel{!}{=} a_{i+1}$
3. $\sum_i^{I_c} 2^{o_i} \stackrel{!}{=} e - a$

Complex Real-Life Quickcheck Example (2)

Algorithm

What is the *minimal* number of CRDs for a given interval?

Interval notation

$$\begin{aligned} I &= [a, e) \\ S &= e - a \end{aligned}$$

Powers of 2 helper function

$$\begin{aligned} P(x) &= \{p | \sum_p 2^p = x\} \\ P^{-1}(x) &= \sum_i^{P(x)} p_i \end{aligned}$$

Algorithm to obtain smallest *number* of CRDs possible

$$\begin{aligned} P_s &= \max P(S) \\ P(b) &= P(e) \cap \{p_s, ..., p_\infty\} \\ b &= P^{-1}(P(b)) \\ |I_c| &= |P(b - a)| + |P(e - b)| \end{aligned}$$

Implementation in Haskell

Download Link

`https://qasm.de/downloads/quickcheck-example.zip`

Useful functions:

- 2^x: `2^x`
- $a|b$: (bitwise OR): `a .|. b`
- Cast Number-Literal to Word: `(123 :: Word)`
- Get highest/Lowest power of 2 in number: `highestBit 10` $\rightarrow 3$, `lowestBit 10` $\rightarrow 1$
- Minimum or maximum of 2 numbers: `min`, `max`
- Compile & Test Loop: `ghcid`
- REPL: `ghci Main.hs`

Automatic Unit Test Generation Challenges

Automatic Unit Test Generation is extremely uncommon in the industry.

Some reasons are:

- Specifying properties correctly is often not easy
- . . . or simply much harder than writing "enough" "realistic" test cases
- Transforming random byte streams into valid test inputs is hard for complex structured input types (e.g. XML, language, . . .)
- General problems with untestability mostly also apply here
- It might end up being hard to justify the effort of building a test data generator for more than a day in front of management.

$\rightarrow$ identify wisely where automatic test generation is the right fit, and then don't waste its potential.

Code Coverage

Bugs can occur in any line of code.

Basic Blocks

Programs consist of many branches, loops, etc. - at the machine language level they are called "basic blocks".

Question

Have we tested every line of program code with a unit test if its execution does not result in all basic blocks being executed?

Code Coverage can be measured in "percent of blocks executed".

Code Coverage as a Metric

$x\%$ Coverage $\rightarrow$ Sounds like a great metric for code quality!

What could go wrong?

What happens when you have programmers who just work strictly for 100% coverage because management demands it?

Code Coverage as a Metric

$x\%$ Coverage $\rightarrow$ Sounds like a great metric for code quality!

What could go wrong?

What happens when you have programmers who just work strictly for 100% coverage because management demands it?

Coverage Reality

Estimates of the perfect coverage number are hardly realistic in practice. Still, you want "good", "great" coverage.

Common Problems in the Industrial Practice

- Late introduction of unit tests: Difficult "testability" of the system
- Unit tests have an invisible value: sometimes not approved by management!
  $\rightarrow$ "Too much effort, too little benefit!"
- Bad unit tests that were created under time pressure (e.g. through senseless following of metrics)
- Lack of discipline with new code that doesn't come with new tests
- Inexperienced / poorly trained programmers in testing often fail to build "atomic" tests

Limitations of Unit Testing

- Unit tests do not test the *interaction* of components
  $\rightarrow$ Integration tests help here
- The quality / scope of unit tests tells nothing about how *usable* the software is in the end.

Most important Limitation of Tests

Testing shows the *presence* of bugs, not their *absence*. - Edsger W. Dijkstra

See also: `https://qasm.de/unit-test-limitations`

Sanitizers

What is a sanitizer?

A sanitizer is compiler function that can be optionally enabled. It instruments code to provide additional control mechanisms. These mechanisms create obvious error messages/crashes that help detecting "undefined behavior" and run-time errors.

$\rightarrow$ Unit tests and entire applications can be built with activated sanitizers.

Sanitizers

What is a sanitizer?

A sanitizer is compiler function that can be optionally enabled. It instruments code to provide additional control mechanisms. These mechanisms create obvious error messages/crashes that help detecting "undefined behavior" and run-time errors.

$\rightarrow$ Unit tests and entire applications can be built with activated sanitizers.

Usage

```
$ g++ -fsanitize=<sanitizer> -o my-app main.cpp
$ ./my-app # Emits error messages at runtime
```

Undefined Behavior (UB)

UB describes code whose behavior is not specified in the language standard.

Undefined Behavior (UB)

UB describes code whose behavior is not specified in the language standard.

Examples:

- Division by 0
- Dereferencing of `NULL` pointers or dangling pointers
- Out-of-bounds access to arrays
- Signed integer overflow
- Strict aliasing violation
- ...

Undefined Behavior (UB)

UB describes code whose behavior is not specified in the language standard.

Examples:

- Division by 0
- Dereferencing of `NULL` pointers or dangling pointers
- Out-of-bounds access to arrays
- Signed integer overflow
- Strict aliasing violation
- ...

Potential for optimization

UB represents optimization opportunities for the compiler.

Which Kinds of Sanitizers Exist?

It depends on the compiler. Clang examples:

- AddressSanitizer
- ThreadSanitizer
- MemorySanitizer
- UndefinedBehaviorSanitizer
- DataFlowSanitizer
- LeakSanitizer
- ...

Compiler Documentation:

GCC `https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html`

Clang `https://clang.llvm.org/docs/index.html`

Detection of Important Error Classes

- Race conditions
- Out-of-bounds access to heap, stack, globals
- Use-after-destruction
- Use-after-free
- Use-after-return
- Double-free, invalid free
- Memory leaks
- Initialization order problems
- Uninitialized read
- Misaligned- and null-pointer access
- Signed integer overflow
- Lossy implicit conversion between numeric data types
- . . . and *many* more . . .

Sanitizer Example Code

Download Link

`https://qasm.de/downloads/sanitizer-example.zip`

To do:

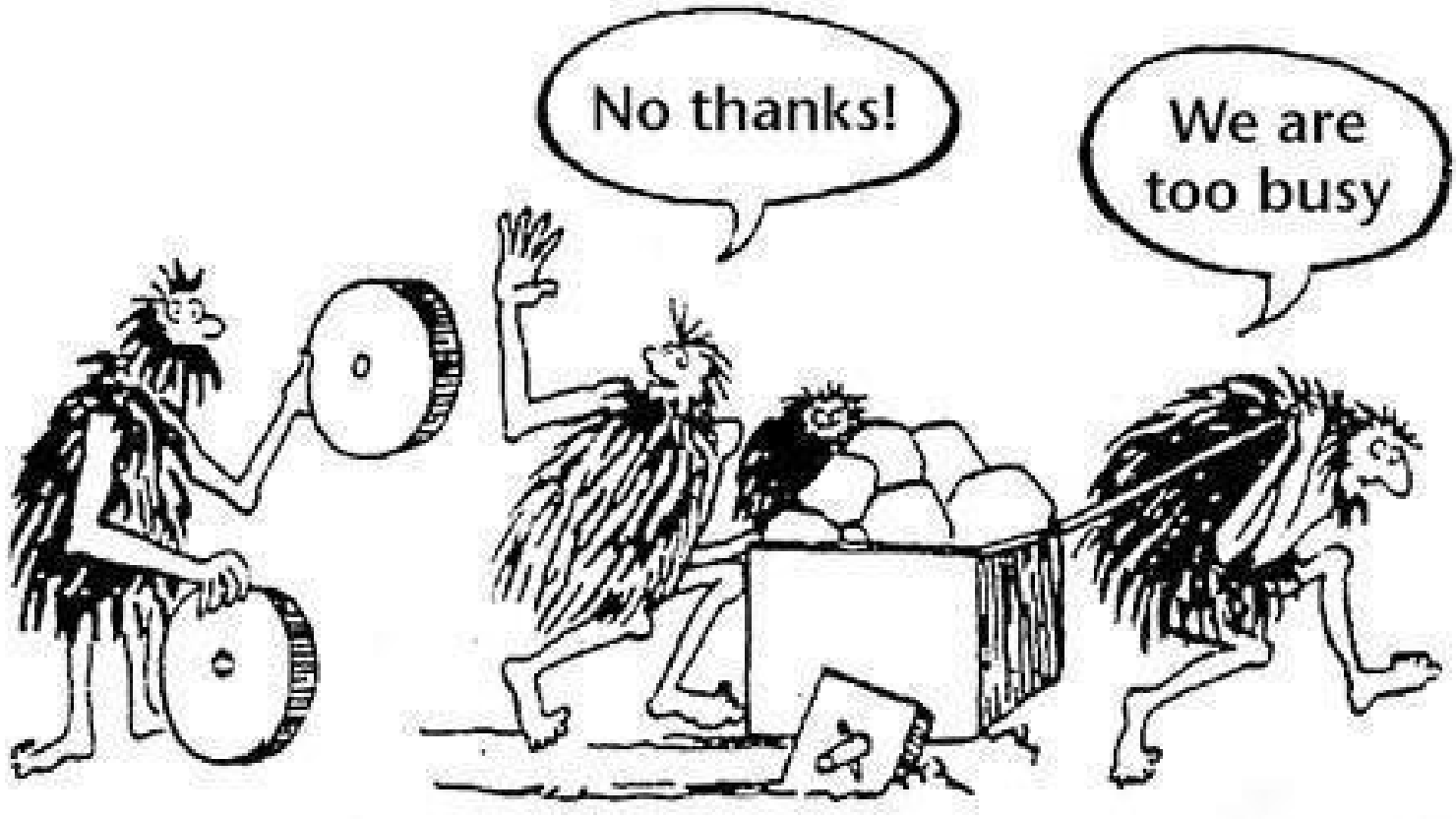
- Read through the minimalistic example C++ apps and understand them
- Each of them contains bugs → try to see them
- Run the apps without sanitizers
- Run the apps with sanitizers → understand their output

Limits of Sanitizers

- Trigger at run time - potential is capped by your code coverage
- False positives
- Higher memory consumption, less performance through additional control structures
- Limited operating system support
- No support for static linking (or other similar compiler / linker settings)
- No support for binaries without debug symbols

Unit tests (and sometimes debug builds in general) should generally always be compiled and run with a "healthy" selection of sanitizers. It also makes sense to build different UT binaries with different (otherwise mutually incompatible sanitizers) and run them all.
The finished application or parts of it can, depending on the type of requirements, also be built with sanitizers in integration tests.

Acceptance of Sanitizers in the Industry



Common Problems in the Industrial Practice

- In case of late involvement in the development process:
  - Too much UB code discovered at once, conversion too time-consuming - but it has been running well for years for paying customers
  - Lots of false positives that are perceived as *noise*
- "We don't need it - every change runs through on Windows / Mac / Linux VMs anyway."
- Compilers in use are too old or simply do not offer sanitizers and code may no longer compile on newer compilers
- The build system makes integration of sanitizers hard
- For practical reasons, the application cannot be built / operated with sanitizers. Neither do unit tests work, because there are none and the architecture cannot be tested.

Static Analysis

What is Static Analysis?

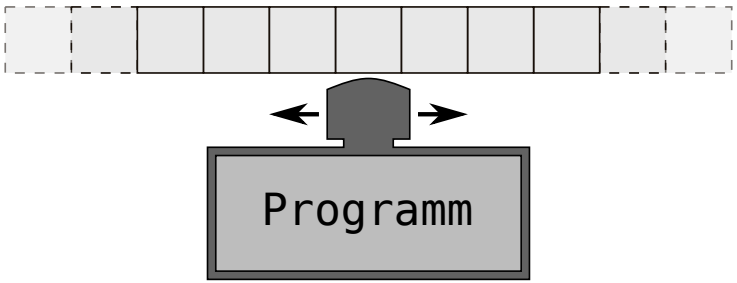
Static analysis tries to automatically find errors by analyzing the program text.

In the past, compilation and static analysis were done by separate tools with similar workflows. (See also "Linter" and "Style Checker") The analysis capabilities of compilers tend to become increasingly better and might make static analyzers unnecessary.

Automatically Enforcing Rules

Static analyzers can highlight certain "style issues" that do not have to / should not be a known concept to the compiler. (e.g. duplication) These are then also based on rules that can be configured.

The Halting Problem



Fundamental Question

Is there a generic function f which, for a Turing machine t and some program input e, can decide whether the program will **stop** at some point without executing it?

$$f(t,e) = \begin{cases} \mathit{true} & \text{if program does terminate} \\ \mathit{false} & \text{if program does not terminate} \end{cases}$$

The Halting Problem

Fundamental Answer

Alan Turing proved in 1936 that the holding problem **cannot** be decided.

The Halting Problem

Fundamental Answer

Alan Turing proved in 1936 that the holding problem **cannot** be decided.

Practical Implications

What does this mean for static analysis?

Limitations of Static Analysis

There are pretty good static analyzers out there, but they will always be limited by the Halting Problem.

- SA is by no means a substitute for run-time tests
- Usually lots of false positives

Common Problems in the Industrial Practice

- Many proprietary static analyzers are expensive
  - Rare updates
  - No support for language standards that are too new, as upgrade is too expensive / impossible
  - Everything is supposed to be done with this tool now instead of getting the build system ready for other tools.
- . . . if they are good, but unfortunately run centrally:
  - Often difficult / lengthy adaptation to projects.
  - Too complicated to set up for quick analysis of some program
- Due to the usually enormous number of false positives, sometimes true positives are "clicked away"
- Technical impossibility to analyze code from pull requests *before* merge

Limits of Programmer Creativity

Test Case Variety

In order to uncover as many bugs as possible, the programmer has to be as creative as possible while writing test cases.

Question

Is that enough, or are inputs easily overlooked that can lead to bugs or even crashes?

Fuzzing

Fuzzing was born in 1989 at the University of Wisconsin-Madison by Barton Miller et al. developed.

Idea

Exploit (or create) an input interface of the program in order to feed it with random inputs - until the program crashes or produces incorrect results

$\rightarrow$ For hours or days!

"Strategic" Randomness

In order not to waste CPU time and to test as many input combinations as possible with high throughput, good fuzzers generate inputs strategically:

- No identical input is tried more than once
- Valid sample input sets from the programmer are used as a starting point for random modification
- Code coverage instrumented binaries are fuzzed:
  This way the fuzzer can determine which inputs lead purposefully to which basic blocks in order to explore all areas of the program
- Inputs that lead to a certain basic block are intelligently shortened

Real-Life Industry Example: Google Project Zero Windows Kernel Font Fuzzing

Microsoft Windows used to parse font files (partly due to performance) *in the kernel*, which only changed with Windows 10.

From 2015, Google engineers have spent more than a year automatically generating font files randomly and loading them into the Windows kernel in order to provoke crashes / errors.

Errors found can be potentially exploitable security holes!

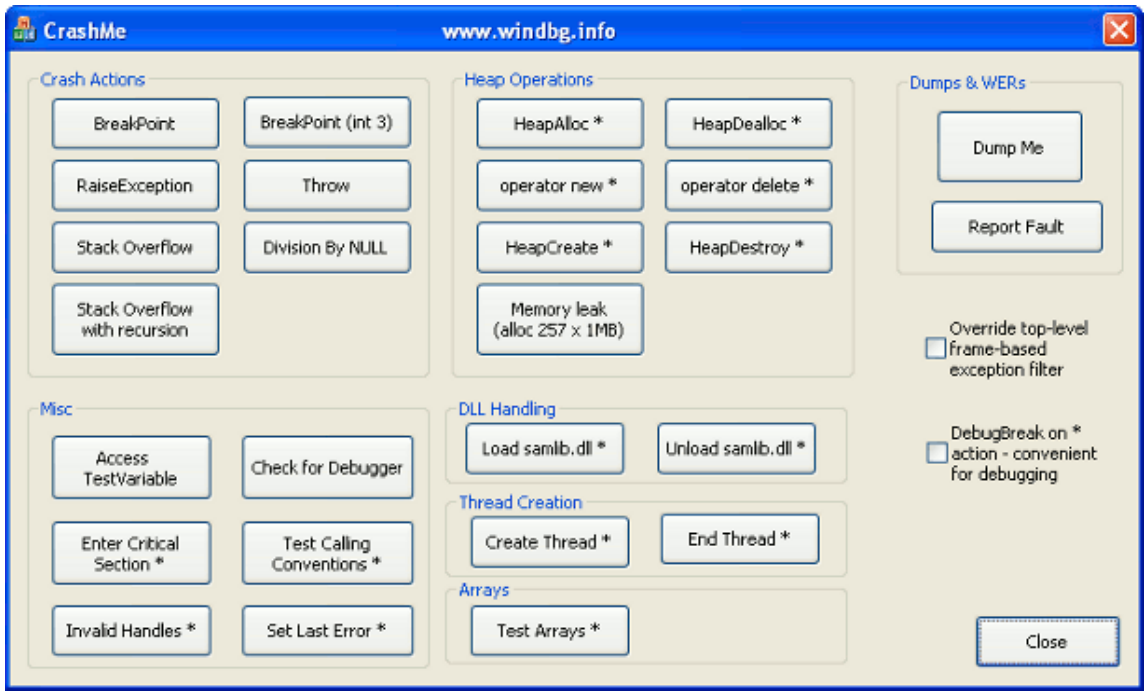
Google Project Zero Windows Kernel Font Fuzzing Results

Results

- 16 vulnerabilities found in Windows binaries from Microsoft
  - 2 of those happened to reveal the same vulnerability as already known from existing malware (!)
- More than 50 bugs found in FreeType library
  (easier to fuzz than Windows kernel, because OSS and thus fuzzer was easier to adapt)

Real-Life Industry Example: VMM Fuzzing

CrashMe is a Windows app for fuzzing debugger and VMM environments.



We found bugs with it in our VMM!

Homepage: `http://www.windbg.info/apps/46-crashme.html`

Real-Life Industry Example: Patchelf Fuzzing with AFL

AFL discovered crashes within *seconds*. It discovered 19 unique bugs after 30 minutes.

Fuzzing Exercise

Fuzzing an RPN Calculator with AFL

Download Link

`https://qasm.de/downloads/fuzzing-rpn-calculator.zip`

To do:

- Understand the RPN calculator app
- Build and fuzz it
- Understand how the inputs that the fuzzer discovered lead to crashes
- Fix the app until it withstands long fuzzing times

Importance of Fuzzing

Fuzzing is a realistic way to find security holes in *black box* products.
$\rightarrow$ it is therefore also popular among security researchers and "hackers".

How *responsible* is it to deliver software products that run in privileged modes and accept user input on customer machines without having vigorously fuzzed them beforehand?

As the SW author, you have the advantage that fuzzing is easier thanks to detailed knowledge of and access to the internal product structure.

Limitations of Fuzzing

- It is fundamentally just *trying out* inputs
- Fuzzer generally has little to *no understanding* of the program
  $\rightarrow$ can't decide if program behaves *correctly*!
- Non-deterministic time until bug discovery
- Programmers usually have to build an adapter interface from random number byte streams to the interface to be tested
  This is often not trivial! (Heap allocator example)

Common Problems in the Industrial Practice

- Fuzzing is often **unknown** to developers and QA specialists
  - Unclear utility
  - Unclear effort
- Hard-to-test code is usually also difficult to fuzz

Integration Tests

- Knowing about the limitations of UTs, we still need to test the *interaction* of code modules
- Integration tests are not fundamentally different from unit tests
  - The effort is more in automatically pulling up, playing through and tidying up a test environment, which is much more complex than that of a unit test
  - When a unit test does I/O, it is actually no longer a unit test
  - The "unit" is *not atomic* here, but can e.g. be a complete network of virtual machines that simulate the real-life application environment
- Parts of the application or the entire application are stimulated in the RL environment and the correct reaction is tested
- Integration tests are often also called **end-to-end** test

Levels of integration

It makes sense to **split** an app into debuggable and testable mini-apps.

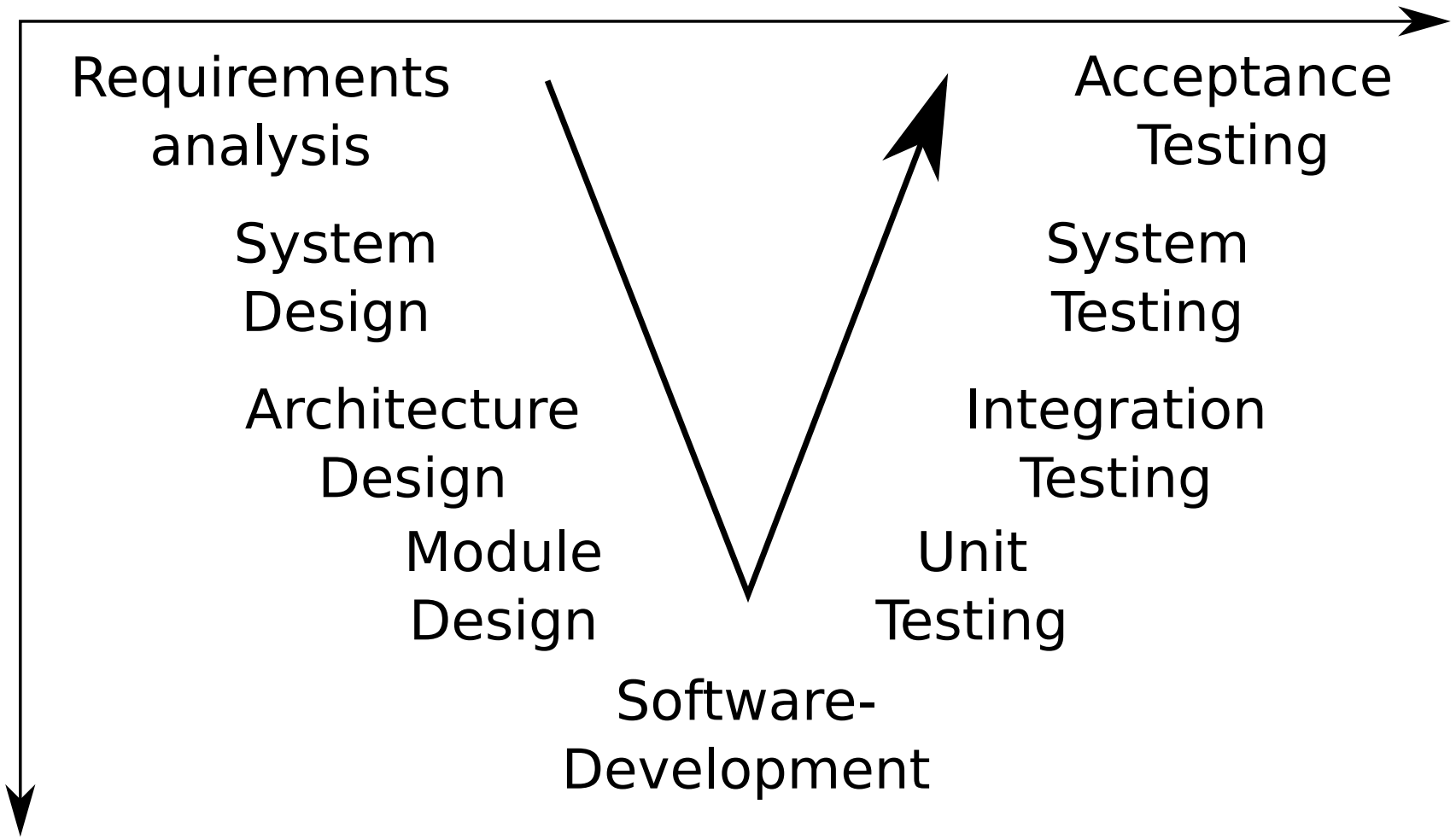
Where to split?

Split lines appear very natural on the borders between modular software. Monolithic software projects are very hard to split, and therefore also hard to test well.

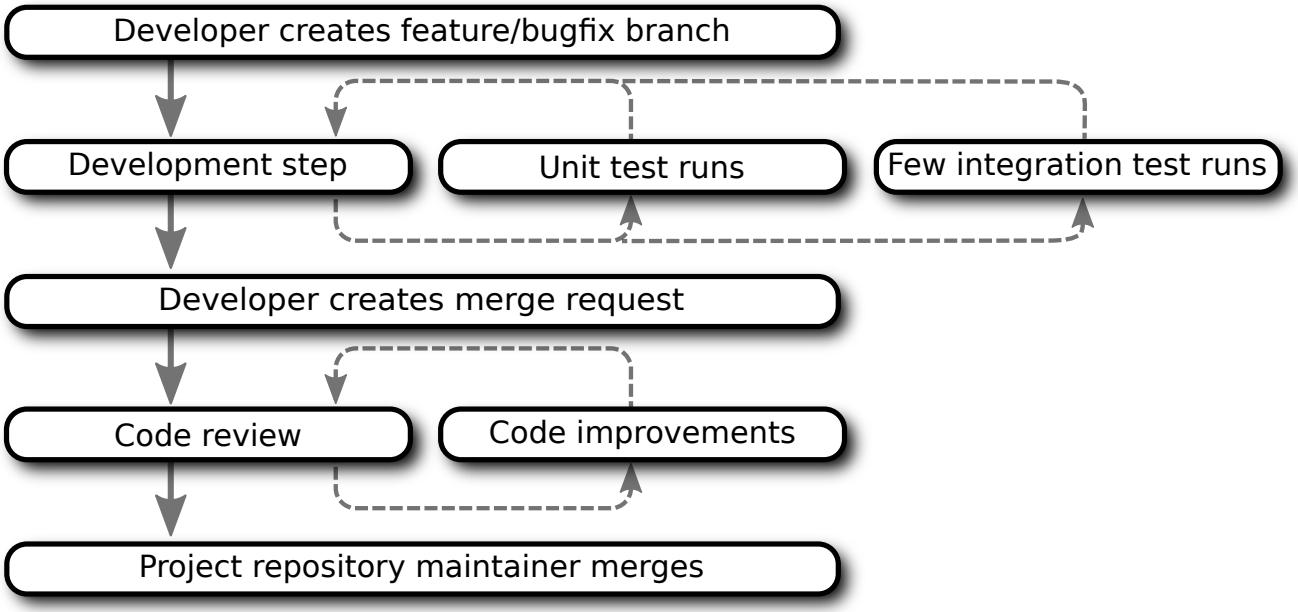
Example: Userspace application with custom kernel module

The app can be tested for correct processing of mocked input from a fake kernel module. The kernel module can be loaded into a virtual machine and then stimulated with test data from a fake app.

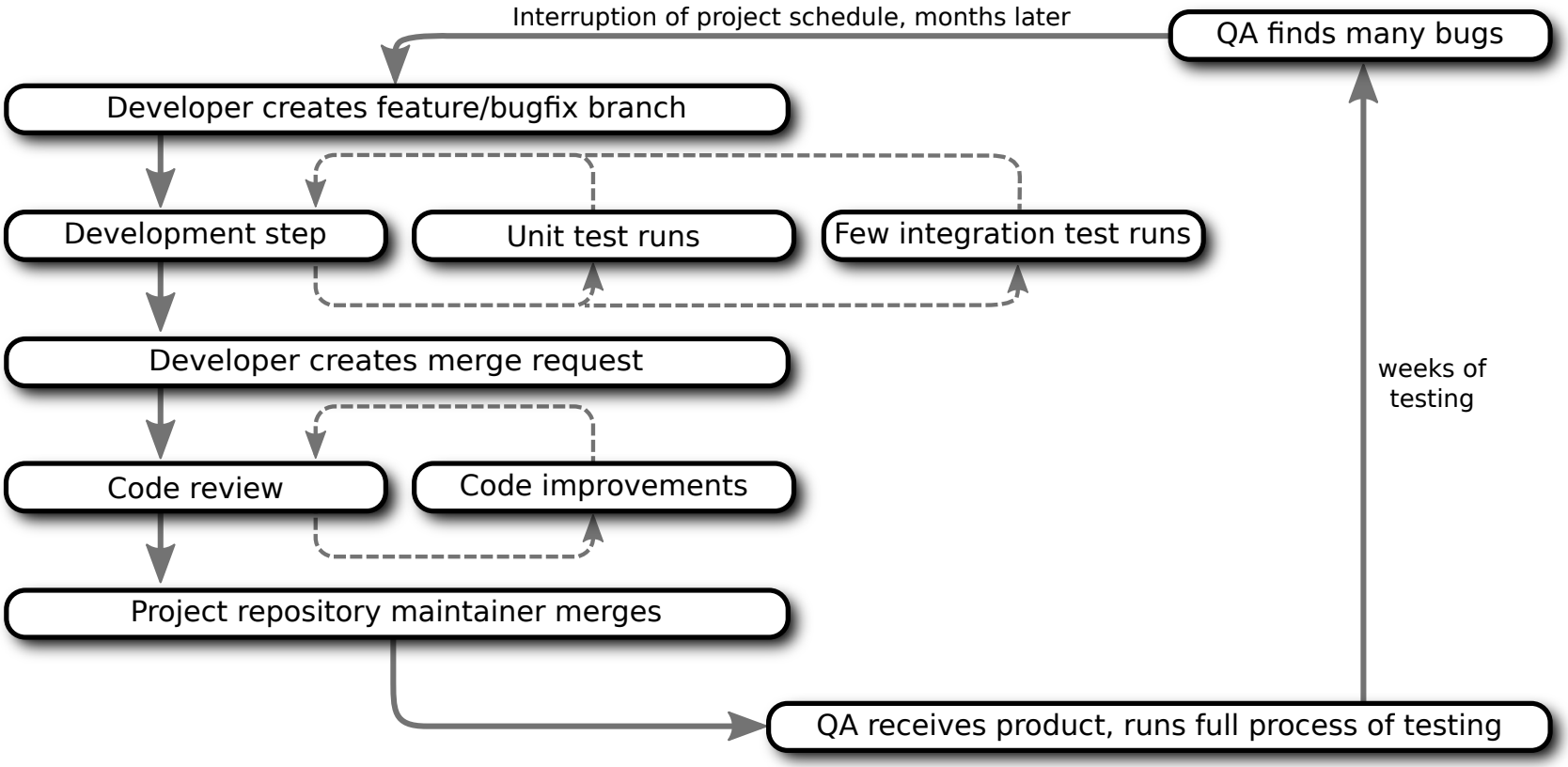
V-Model


Requirements analysis
System Design
Architecture Design
Module Design
Software-Development
Unit Testing
Integration Testing
System Testing
Acceptance Testing

Typical Development Workflow (1)



Typical Development Workflow (2)



Implications of the V-Model on SWE

QA-Specialists perform higher integration testing

- The more complex, the less automation → High effort and expense
- Does not happen per merge-request or daily/weekly, but **per release**

If bugs/problems are discovered short before release...

- Developers are already on the next feature/bugfix
- Which feature/bugfix/commit/developer caused the problem?
- "Short before release" means "under deadline pressure"
  → Chaos & Overhours!

Separation Between SWE & QA

"Over the fence" SWE&QA processes

For QA, SWE's product is a black box.
For SWE, QA's tests are a black box.
→ Is the problem really in the product or rather in the failing test?

- SWE and QA should communicate changes in the feature set and adapt tests before it's too late
  → most organizations are not set up like this 😕
- SWE does not have time to write good integration/e2e tests.
  QA does not have the right competence to automate them properly.

Career Tip: Improve the Regression Feedback Time

Section Content

Synchronization of Distributed Repositories

- We learned about git and how to use it on local repositories
- Next: Collaborative workflows with git for bigger projects

Synchronization of Distributed Repositories

- We learned about git and how to use it on local repositories
- Next: Collaborative workflows with git for bigger projects
- Central questions:
  - How to coordinate changes when multiple people/teams work on the same project concurrently?
  - What ways exists to split projects over multiple repositories?
  - Which new problems emerge from multi-repositories?

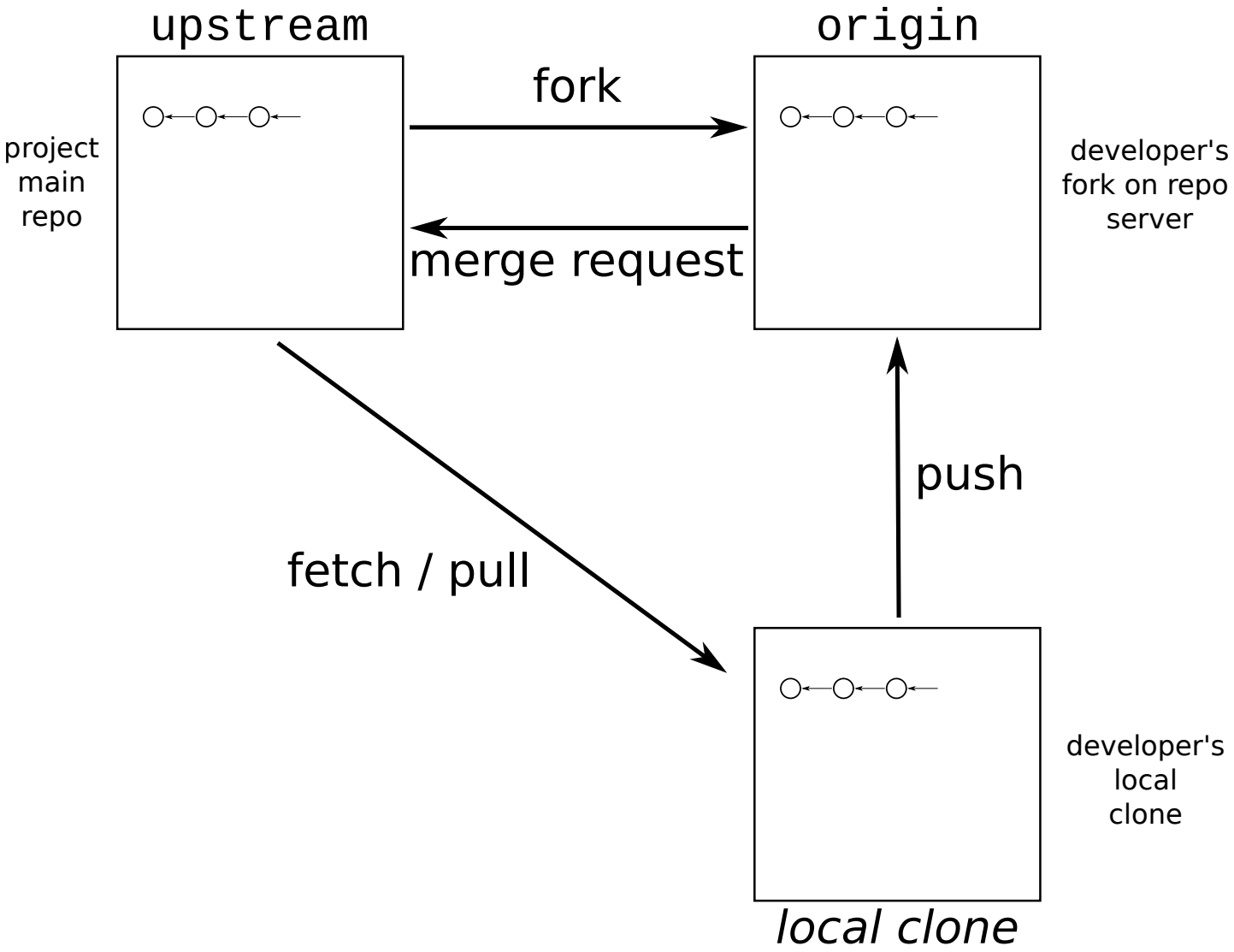
Branching

- Centralized VCS like CVS, SVN, Perforce, Clear Case
  - All devs commit & push on same branch (or few branches)
  - Committing and pushing is one action
- Decentral VCS like git
  - Every dev has their own branch(es) which don't need to be synced with central server
  - Branches can be merged, rebased on each other, and history rewritten
  - Work begins based on some branch and is later merged back into it (e.g. `master`)

git: Basic Commands for Synchronization of Repos

Command	Effect
`git clone <url>`	Create fresh clone of existing repo
`git remote add <name> <url>`	Create local reference to remote repo
`git reset <commit>`	Reset local repo state to commit
`git fetch`	Sync branch data from remote repos
`git pull`	git fetch + merge with local branch
`git push`	Push local branch data to remote repo

Forks and Merge Requests, Upstream and Origin



Merge-Request Based Development Processes

In large, collaborative software projects, it should **not** be common to simply merge developer branches into the master branch without further team interaction.

$\rightarrow$ Why?

Merge-Request Based Development Processes

In large, collaborative software projects, it should **not** be common to simply merge developer branches into the master branch without further team interaction.

→ Why?

Changes go into production without being checked:

- Does the code even compile?
- Do the changes actually correspond exactly to the original task?
- Are there still bugs that are not discovered by tests?
- Even with error-free code:
  - Is the code / architecture good?
  - Code style?
  - Documentation?
- ...

Merge-Request Based Development Processes

Significance of merge requests

A merge to master is usually the **last step into production**. If it contains bugs, the product contains these now too.

Merge-Request Based Development Processes

Significance of merge requests

A merge to master is usually the **last step into production**. If it contains bugs, the product contains these now too.

Solutions like GitHub, GitLab, Gitea, etc. provide workflows for forking repositories and creating merge requests:

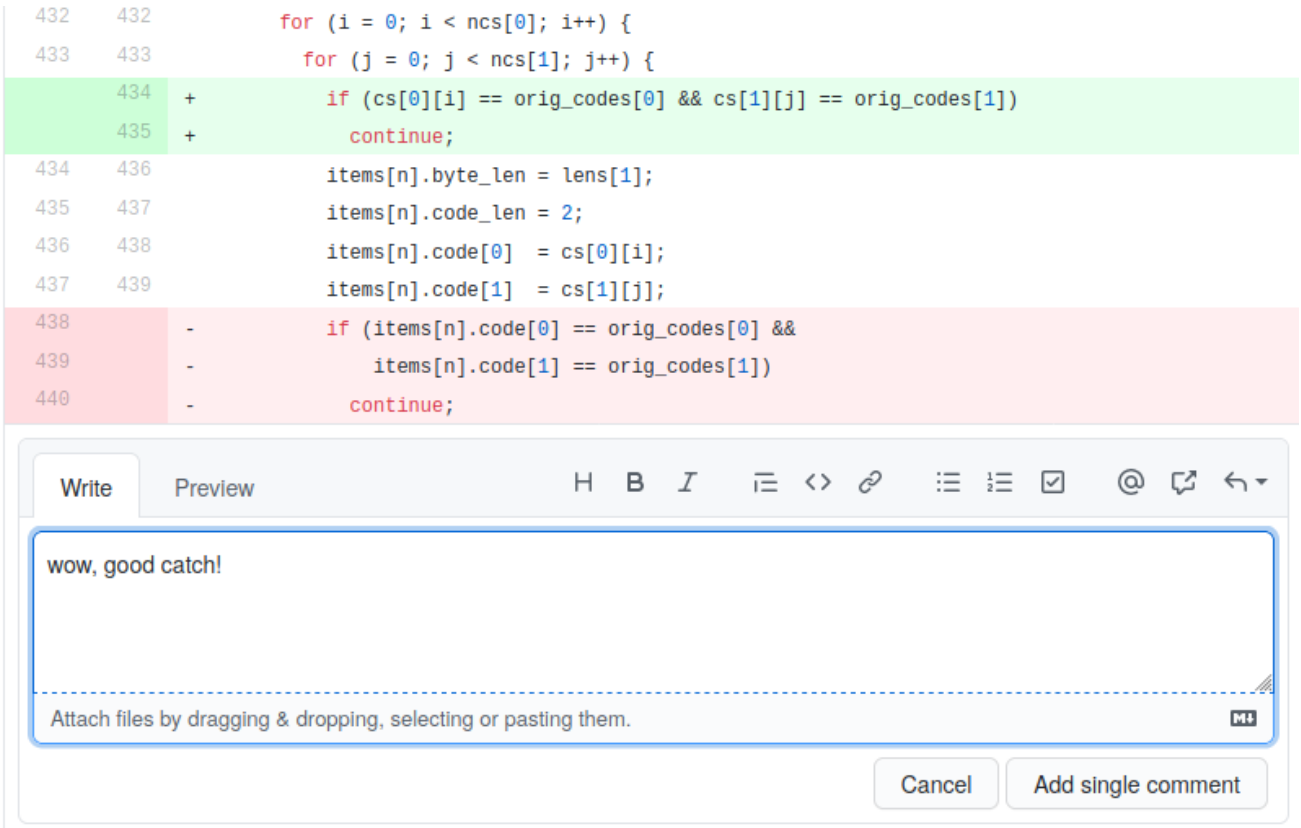
1. Developer forks project repo
2. Developer creates branches, writes features, fixes bugs
3. Developer creates a formal merge request:
   - Team reviews and discusses changes
   - Pipeline runs builds, tests, integration, ...
4. After incorporating review suggestions, code is merged to `master`
5. Developer can delete their old branches

Common Problems in the Industrial Practice

- CVS, SVN, Perforce, Clear Case. . . :
  - 1 commit = 1 push to server → suboptimal starting point for quality barrier based workflows
  - *"Locked files"*
  - *"Trunk locked"* phases
  - No/only few branches, because branches are hard
- *Force-pushes* to master or other long-living branches/tags
- Movements away from CVS/SVN/Perforce/Clear Case towards git are slowed/sabotaged by "culture clash", because long-established developers defend old workflows that cannot be mapped to git and cannot imagine the value of more modern workflows.

Code Review

Merge requests display code changes and give team members the chance to comment:



Code Review

Typical Features of repository hosting solutions regarding merge requests

- Developer can summarize reasons and context for certain changes
- Diff-View: Green and red highlighting for added and removed code lines
- Fellow developers can approve or deny changes with suggestions
  - Per-MR & Per-line discussion threads
  - . . . which can be marked as *resolved* after pushing fixes
- Only *maintainers* can decide that review is over and merge MRs
- . . . optionally after minimum number of *approvals*

Code Review: Ultimate Advantages

- Different experts find different problems
  - Security
  - Architecture
  - Programming language specific details
  - Domain expertize
- Certain classes of bugs are still best detected by humans
  - Code Style
  - Architectural improvement potential
  - Different types of security vulnerabilities
  - Duplication by failing to reuse existing library functionality
  - Missing test cases (due to lack of phantasy, perspective, . . .)
- Mentoring

Criteria for Good Merge Requests and Code Review

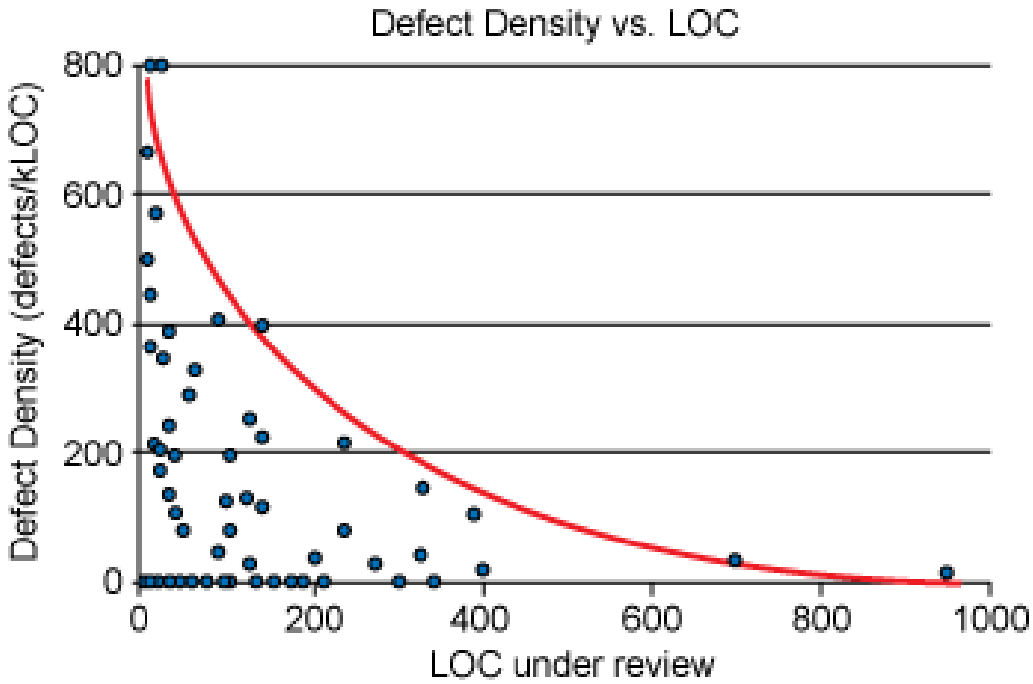
- Good summary (fellow coders really get the *big picture*)
- Readable code
- Semantic commits
- not too many changes per MR
- Code review must be **rigorous** - regardless of *whose* MR it is
- Comments must be **actionable**: Receiver must immediately understand what is wrong and what to do now

Perspective of the Reviewer

- **Reject bad code**: Assume the role of a *cooperative opponent* who defends the project's quality
- Be **kind**!
- Creating the **best quality possible** for the project is even more important than some individual's feelings
- Look at the changes with specific questions in mind:
  - Does the code *really* do what the summary says?
  - Is the code correct?
  - Is there potential to remove code by reusing existing libs?
  - Does the code follow all team conventions? (Style, documentation, naming, ...)
  - Does the code contain undocumented assumptions?
  - Do all test cases make sense? Are there test gaps?
  - If there are new compile- or runtime depencencies: Are they really necessary?

Typical Problems with Merge Requests and Code Review

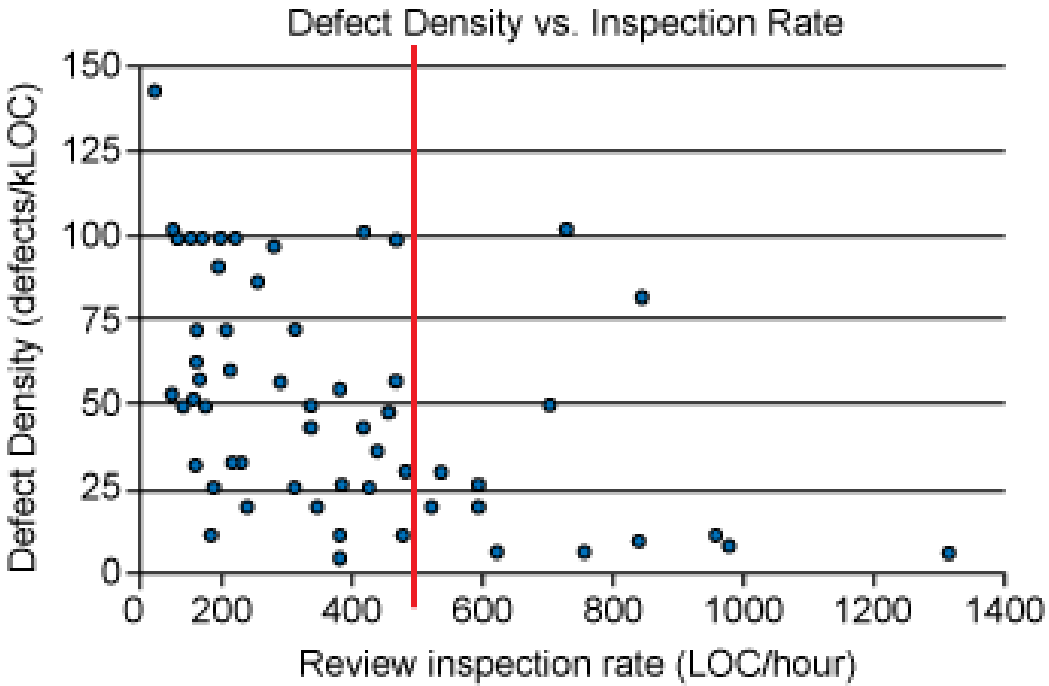
Too many changes per merge request



Source: smartbear.com

Typical Problems with Merge Requests and Code Review

Too many code reviews per time



Source: smartbear.com

Bikeshedding Effect

Complex Merge Requests

In MRs with enormously complex changes, discussions often focus on less important details.

There are *psychological reasons*:

Complex Change "The person who wrote this is much smarter than me. Some detail looks wrong, but I don't want to risk looking stupid."

Simple Change "I want impact. On *this* detail i can finally demonstrate my value with professional opinion."

Bikeshedding Effect

Complex Merge Requests

In MRs with enormously complex changes, discussions often focus on less important details.

There are *psychological reasons*:

Complex Change "The person who wrote this is much smarter than me. Some detail looks wrong, but I don't want to risk looking stupid."

Simple Change "I want impact. On *this* detail i can finally demonstrate my value with professional opinion."

- Bikeshedding has strongly *annoying* aspects, especially near deadlines
- Often typical candidates: New team mates who did not have many chances to prove themselves, yet

Merge Buddies

Social dynamics might emerge constellations where few team members regularly do each other the favor of *waving* their code through.

→ Use multi-approval settings

Programmer Ego

Even the most constructive criticism can be taken personally.

Good/Back Feedback Bias

Perfect code tends to be merged without comment. Merge requests are therefore mostly used for *negative feedback*.

Developer perspective: "That was 3 weeks of hard work - and now I can't save myself from criticism!".

Typical symptoms and impact:

- Defensive behavior slows down progress
- Sometimes suggestions are ignored - reviewers then have to doublecheck

→ No technical solutions. Try to talk in person, be understanding, and helpful.

Common Problems in the Industrial Practice

- All previously mentioned phenomenona are real
- *No* review happens
  - Some companies have no or bad review platforms
  - Deadlines are too tight for quality time
- During release phases, critisizm is sometimes cast off
- No clear definition of maintainer roles with exclusive merge rights and implied responsibilities
- Rude review culture can demotivate individuals $\rightarrow$ lost potential
- Management often thinks a feature is *done* as soon as the MR is open and applies pressure.
- Merge request *jams*

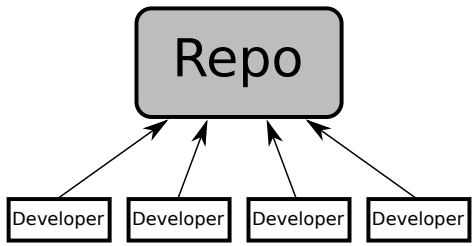
Division of Repositories

The *divide and conquer* approach is useful for big projects that consist of multiple modules/libraries.

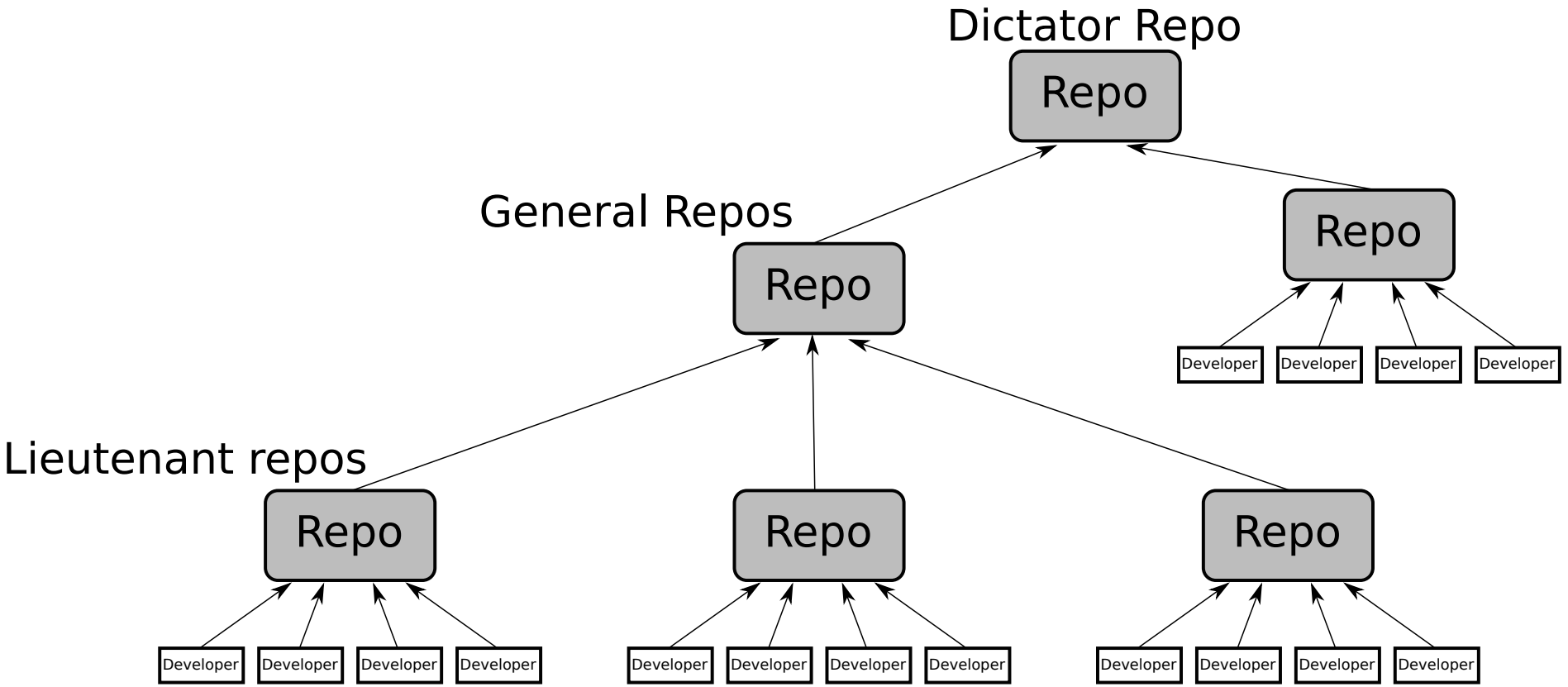
Repos can be divided in different ways

- Segmentation of projects in individual libraries/modules (one per repo)
  - Individual development cycles/sprints, toolchain, reviews, QA, ...
- One monorepo can be forked across teams that concentrate on subfolders with their development
  - Allows for domain-oriented maintenance
  - Well-known example: Linux kernel

Single Repo, Multiple Developers



Upstream/Downstream Repos



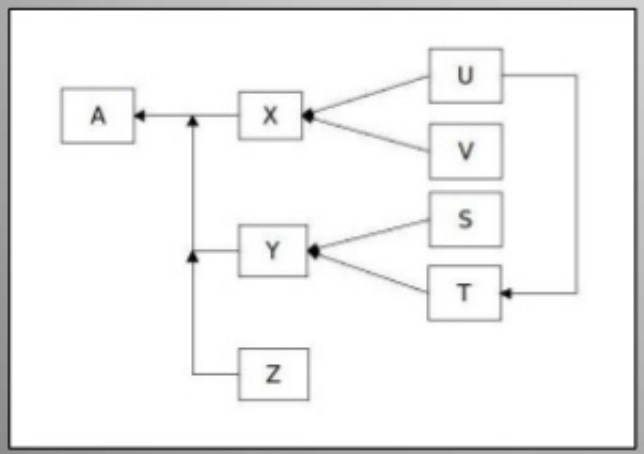
Section Content

Enter the World of Distributed Software Modules

Welcome to Dependency Hell

"A term for the **frustration** *of software users who have installed* **software packages** *which have* **dependencies** *on specific* **versions** *of other software packages."*

Source: Wikipedia



www.visug.be

VISUG

What is a Software *Build*?

Theoretical approach:

$$f(\ldots?\ldots) \rightarrow \text{binary app or lib}$$

What is a Software *Build*?

Theoretical approach:

$$f\left(\begin{array}{c}\text{source code}\\ \text{dependency libs}\\ \text{compiler}\\ \text{build system}\\ \text{build files}\\ \text{shell}\\ \text{environment vars}\\ \text{script interpreters (with libs)}\\ \text{typical userland tools}\\ \text{date + time}\\ \text{Windows: Registry State}\\ \text{user privileges}\\ \ldots\end{array}\right) \rightarrow \text{binary app or lib}$$

f should be a **pure function**!

What is a Pure Function?

Pure Function Properties

1. Its return value is the same for the same arguments
2. Its evaluation has no side effects

No variation with/mutation of local static variables, non-local variables, mutable reference arguments or input streams from I/O devices.

Why View SW Builds as Pure Functions?

What are the problems with impurities?

Why View SW Builds as Pure Functions?

What are the problems with impurities?

- Failing builds
  - on different machines
  - at different times
- Unknown/unwanted dependencies
  - Potential license implications
- Same build process on different machines/times produces different binaries
  - Does not always work
    - Different errors in different binaries? Debugging-nightmare!
  - No bit-identity

Purity of Tests

Unit and integration tests can also be modeled like builds:

$$f\left(\begin{array}{c}\text{app binaries}\\ \text{run time dependencies}\\ \text{test input}\\ \text{system configuration}\\ \text{shell}\\ \text{environment vars}\\ \text{script interpreters (with libs)}\\ \text{typical userland tools}\\ \text{date + time}\\ \text{Windows: Registry State}\\ \ldots\end{array}\right) \rightarrow \text{Test result: Success/fail}$$

This is trivial for unit tests - and anything *but* trivial for integration tests.

What Exactly Are Dependencies?

The most *obvious* dependencies are:

- Libraries and frameworks
  - . . . and recursively, their dependencies
- Compiler or intepreter
- Build system and build files

Less Obvious Dependencies

- Bash
  - Mind environment vars, maybe set by other tools on the system
- Userland tools (binutils, coreutils, patch, . . .)
- Operating system (kernel)
- libc
- If tools like `curl`, `wget`, `scp` are used to download content/dependencies:
  - Installed certificates
  - Stored credentials
  - openSSL version
- Versions of *all* mentioned apps and libraries
- Language settings
- Random generators
  - Does the system have enough entropy?
- Date and time
- Windows: Registry state!
- . . .

Kinds of Dependencies

Which *kinds* of dependencies can we sensibly subdivide?

Kinds of Dependencies

Which *kinds* of dependencies can we sensibly subdivide?

At least the following two kinds:

- **Compile time** dependencies
  - Toolchain (must be built for build host)
  - Library (must be buildable for target by build host's toolchain)
- **Run time** dependencies
  - Dependency libraries (must be available already or be deployed with app)

Different degrees of reproducibility

- It builds
- It works
- Results (binaries and/or output) are bit for bit identical
  - Very interesting feature
  - Absence of bit identity is (nowadays still) Considered normal

Different degrees of reproducibility

- It builds
- It works
- Results (binaries and/or output) are bit for bit identical
  - Very interesting feature
  - Absence of bit identity is (nowadays still) Considered normal

Reproducibility requires:

- It's reproducible on different machines
- It's reproducible at different times

Common Problems in the Industrial Practice

- In many projects, function f is neither pure, nor really specified but an historically grown mess
- Your first day at work = Whole-day manual setup of build environment
  - Often takes even longer
  - Sometimes it's not possible to replicate build setups
- Unclarity: Does the build fail because i did a mistake *again* or are there really problems in the code?
- "Don't touch a running build environment!"
  - Toolchain/dependency updates are avoided because they would result in chaos
- Code from e.g. *last year* does not build any longer, but client needs **this version** fixed

Avoiding Effects of Non-Reproducibility

Can impurities still occur if we take a *fresh system* for every build?

- Install only one compiler
- Install only necessary libraries
- Install only one Python/Ruby/. . . interpreter
- Install only one version of the build system
- . . .

Avoiding Effects of Non-Reproducibility

Can impurities still occur if we take a *fresh system* for every build?

- Install only one compiler
- Install only necessary libraries
- Install only one Python/Ruby/. . . interpreter
- Install only one version of the build system
- . . .

This is a good start, but won't give us bit-identity. We will also have different experience on e.g. different Linux distros. Some systems don't even let us decide such basic things.

→ Let's ignore bit-identity for a while.

Creating Defined, Isolated Environments

There are several ways to create isolated environments which have their own rules and settings. We will concentrate on the following:

- **Virtual Machines**
- **Container** environments

Creating Defined, Isolated Environments

There are several ways to create isolated environments which have their own rules and settings. We will concentrate on the following:

- **Virtual Machines**
- **Container** environments

Let's see how these work out in different use cases:

- Daily developer use case (Editing and *incremental* building)
- Automatic build infrastructure use case (Fresh build everytime)

Virtual Machines

- KVM + Qemu, Virtualbox, Parallels, VMWare, ...
- Completely free configuration of a full-blown computer in a computer
- Free operating system choice
- System can be deployed on the desktop or in the intranet
  - Access via RDP, VNC, SSH, ...
- Most virtual machine monitor solutions provide *snapshots*
  - Snapshots can be made of running and/or shutdown VMs
  - Resetting a VM for every build can be very useful to avoid *bitrotting*

Pros and Cons of VM Environments

- ⊕ Free OS choice
- ⊕ Quick Snapshot reset for every build
- ⊕ Usually easy to duplicate for fellow developers
- ⊕ We can simulate *networks* of machines in order to test a product
- ⊖ Reduced comfort
- ⊖ Overhead: How well do 100 or 1000 VMs *scale* on a build server?
- ⊖ Overhead: Many VMs = Many disk space

Container Environments

- Docker, LXC, VServer, OpenVZ, ...
- We don't fake hardware but new process space, file system, network, etc.
- The Linux kernel provides so-called **Namespaces** to achieve this
- Very quick *ad-hoc* creation of environments
  - We can easily compose environments from shared parts like file system trees etc.

Namespaces - Lightweight Process Virtualization

Linux provides different categories of namespaces to isolate processes selectively on different surfaces:

- Cgroups
- IPC (Interprocess Communication)
- Network
- Mount
- PID (Process ID)
- User
- UTS (Unix Time Sharing)

Network Namespace Example

Terminal 1

```
# ip netns add foonet
# ip netns exec foonet ip link set lo up
# ip netns exec foonet python -m http.server
```

Terminal 2

```
# curl http://localhost:8000
curl: (7) Failed to connect to localhost port 8000:
# nsenter --net=/proc/<PID>/ns/net curl http://loca
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "
<html>
...
```

Pros and Cons of Container Environments

- ⊕ Selective choice over different types of isolation
- ⊕ Less overhead compared to VMs
- ⊕ Less disk consumption as filesystems can be composed from shared parts
- ⊕ More possibilities for communication between containers
- ⊖ Only the running OS's kernel can be used
- ⊖ Containers are less secure compared to VMs
- ⊕⊖ Namespaces are very complicated to use, need for wrappers

Bocker

Open source mini-project "bocker"

"Docker implemented in around 100 lines of bash."

→ `https://github.com/p8952/bocker`

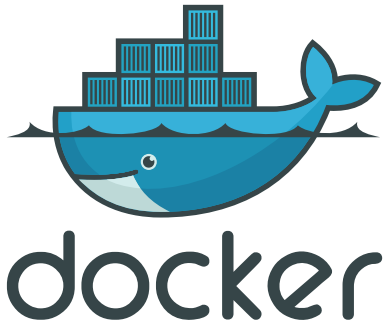
Container Solutions: Docker and Nix

In the following we learn about the FOSS technologies **Docker** and **nix**:

Docker is widely used in the industry and is a *good* solution for isolation and deployment of complex applications.

nix is relatively unknown, but provides a *next-generation* solution for isolation and reproducibility for building, testing, and deploying complex software.

Docker



- Linux-namespace based containers
- First release in 2013
- `https://docker.com`
- Containers are defined in *Dockerfiles* that can reference other Dockerfiles
- A running container can be an environment for multiple processes
- Containers can be switched together using virtual networks
- Very popular in software development and server administration
  - Devs isolate their toolchains and make them reproducible
  - DevOps define pipelines
  - Admins deploy complex multi-container services

Creating Docker Images

1. Create a Dockerfile

```
FROM debian:testing-slim

RUN apt-get update && \
    apt-get -y install python3 python3-requests

COPY ./ /root/
WORKDIR /root

CMD ./test.sh
```

2. Build and run

```
$ docker build -t integration_test .
$ docker run integration_test
```

→ see also: Dockerfile Reference

Anatomy of Docker Images

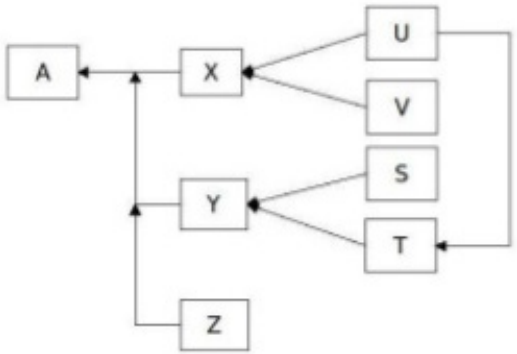
Image Content

```
$ docker history integration_test
IMAGE          CREATED          CREATED BY                                      SIZE
de0d0daab636   29 minutes ago   /bin/sh -c #(nop)  CMD ["/bin/sh" "-c" "./te...   0B
779a03d03e79   29 minutes ago   /bin/sh -c #(nop) WORKDIR /root                   0B
a40275931545   29 minutes ago   /bin/sh -c #(nop) COPY dir:5f8759f758c347294...   2.07kB
6db13abc346a   29 minutes ago   /bin/sh -c apt-get update &&     apt-get -y ...   59.2MB
b51cdf7e1ca1   3 weeks ago      /bin/sh -c #(nop)  CMD ["bash"]                   0B
<missing>      3 weeks ago      /bin/sh -c #(nop) ADD file:672f2811061b3df0a...   69MB
```

- Every layer is an *immutable* UnionFS
- Images can only grow, not shrink
- Common layers can be shared between Docker images

Limitations of Docker

- Building A requires Docker image with proper toolchain and deps X, Y, Z
- Building X requires Docker image with different toolchain and deps U, V
- ...



This sequence requires a *meta*-buildsystem that builds Docker images from inputs of other Docker images.

Further requirements:

- Change detection for incremental builds
- Image management/garbage collection

Non-Reproducible Dockerfiles

What's wrong with this build recipe?

```
FROM debian:testing-slim

RUN apt-get update && \
    apt-get -y install python3 python3-requests
```

Non-Reproducible Dockerfiles

What's wrong with this build recipe?

```
FROM debian:testing-slim

RUN apt-get update && \
    apt-get -y install python3 python3-requests
```

- `testing-slim` is a moving target
- Even fixed version numbers of images can change
- Package versions will change from time to time

Nix

- Nix is scripting language, package manager, and Linux Distribution (NixOS) at the same time
- `https://nixos.org/nix`
- FOSS result of a PhD thesis at TU Delft in 2003 about reproducible builds (PDF)
- Research hypothesis:
  - It must be possible to describe the whole process to build a program/system down to the smallest detail as a reproducible *recipe*
  - Also: Recipes should be composable and overridable
- Nix models the pure build function f using namespaces

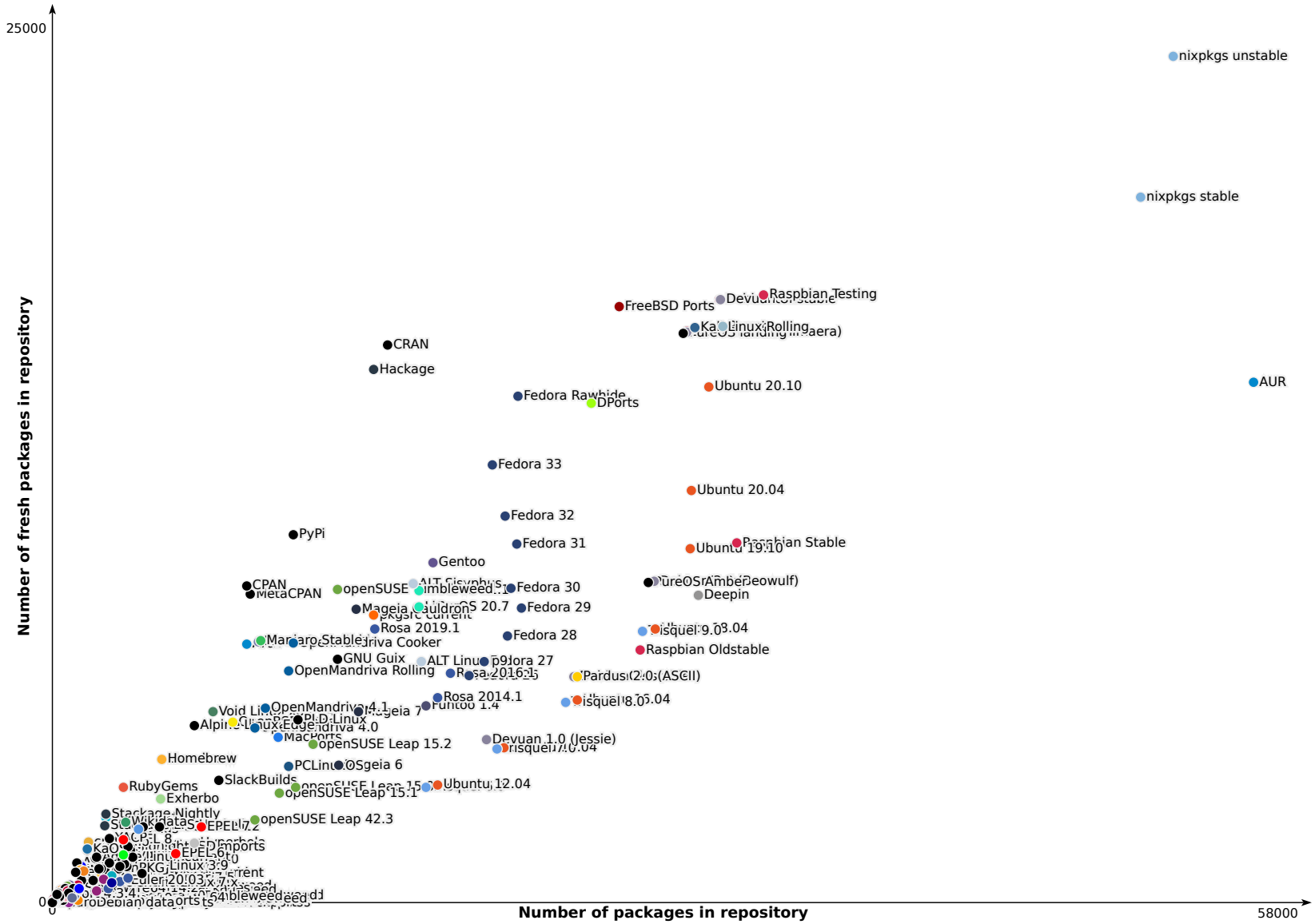
Nix Specialties

- nix language is purely functional $\rightarrow$ composability
- nix has two workflows:
  - `nix-shell` spawns a shell in the development environment for incremental builds
  - `nix-build` performs clean build of the end-product (e.g. app, PDF, VM image, package, ...)
- nix hashes all inputs of *f*
  - it stores multiple versions of packages without clashes
  - Hashes allow for caching identically configured builds
- Packages are at the same time source-based and binary-based
  - If a package cannot be obtained from remote caches, it can simply be rebuilt
  - In fact, nix package builds still work after many *years*

NixOS

- Complete GNU/Linux operating system
- Whole system configuration is a nix expression
- We will regard it as key component for very complex, but still composable integrationtests with multiple VMs in networks
- Package ecosystem is another large nix expression: `https://github.com/nixos/nixpkgs`
- The nix community is extremely awesome - consider contributing here if you are looking for (potentially paid by EU) OSS contributions

NixOS Package Ecosystem



Source: `https//repology.org/repositories/graphs`

Nix Build Basics

- After evaluation, `.nix` expressions are translated to `.drv` *derivations*
- A derivation is a machine-readable, standardized cooking-recipe-like description without any dynamic parts left
- A *realized* derivation results in a read-only output path with build products: `/nix/store/<hash>-<packagename>`

Evaluation/realization phases

`.nix` $\rightarrow$ `.drv` $\rightarrow$ `/nix/store/<hash>-<packagename>`

A Minimal Nix Derivation

File `builder.sh`

```
declare -xp # Show environment vars
echo "foobar" > $out
```

Continue in the REPL: `nix repl '<nixpkgs>'`

```
:b derivation {
  name = "myDerivation";
  builder = "${pkgs.bash}/bin/bash";
  args = [ ./builder.sh ];
  system = builtins.currentSystem;
}
```

- Attributes are later accessible as environment vars
- The `pkgs` attribute comes from nixpkgs and contains ∼50k packages
- Our derivation is technically another package

Basic nix Commands

Command	Effect
`nix-instantiate`	Create derivation from nix expression
`nix show-derivation`	Show a derivation
`nix-store -r`	Realize a derivation
`nix-build`	One-step evaluation and realization
`nix-shell`	Create a shell with a specific environment
`nix repl '<nixpkgs>'`	open a REPL with `pkgs` in scope

Nix REPL

Experimentation in the REPL

```
$ nix repl '<nixpkgs>'
Welcome to Nix version 2.1.3. Type :? for help.

Loading '<nixpkgs>'...
Added 9427 variables.

nix-repl> :b pkgs.hello

this derivation produced the following outputs:
  out -> /nix/store/gdh8165b7rg4y...mbbw89f9-hello-2.10

nix-repl>
```

- `:b x` builds derivation x and prints its output path(s)
- Note that the REPL has tab-completion

Nix Syntax

Mini Crash-Course on Attribute-Sets and Functions

```
nix-repl> x = { a = 1; b = 2; }

nix-repl> x
{ a = 1; b = 2; }

nix-repl> f = a: b: a + b

nix-repl> f 1 2
3

nix-repl> f = {a, b}: a + b

nix-repl> f x
3
```

Nix PATH

<nixpkgs> is expanded with a path that is looked up from NIX_PATH

Content of the NIX_PATH

```
$ echo $NIX_PATH
/home/tfc/.nix-defexpr/channels:ssh-config-file=/nix/store/
    nnkv5sx0xwkzwz7k0kb4adcbdkg9xqqk-ssh_config:nixpkgs=/nix
    /var/nix/profiles/per-user/root/channels/nixos/nixpkgs:
    nixos-config=/etc/nixos/configuration.nix:cbspkgs=/root/
    cbspkgs:/nix/var/nix/profiles/per-user/root/channels

$ nix repl
Welcome to Nix version 2.1.3. Type :? for help.

nix-repl> <nixpkgs>
/nix/var/nix/profiles/per-user/root/channels/nixos/nixpkgs
```

- The nixpkgs from here are simply a git clone of the original repo
- See also Nix Manual, Chapter 18, Common Environment

A Typical Nix Derivation

A Standard Derivation with Standard Helpers

```
with import <nixpkgs> {};

stdenv.mkDerivation { # Generic Derivation Builder function
  name = "myPackage";
  src = ./.;

  # if no other attributes are provided, mkDerivation will
  # employ scripts that automatically work with projects
  # that use autoconf, GNUMake, CMake, etc.

  installPhase = ''
    mkdir -p $out/bin
    cp meineBinary $out/bin/
  '';
}
```

See also Nix Manual, Section 6.1

Package Installation

If a derivation's output contains binaries in its `$out/bin` folder, it's an installable package.

Installing and Uninstalling a Package

```
$ nix-env -i $(nix-build)
installing 'hello'

$ hello
Hello World

$ nix-env -e hello
uninstalling 'hello'
```

Pinning expressions

To make nix expressions build *forever*, it's possible to pin *all* its inputs.

Pinned nixpkgs

```
$ cat nixpkgs.nix
builtins.fetchTarball {
  url = "https://github.com/nixos/nixpkgs/archive/859
      ce47b023d80ccbcf446a112c813972a5949b6.tar.gz";
  sha256 = "14rmrpzbxbf8p...m9qhnwlg4pjkzf7ya6n697g8yd1wx";
}

$ nix repl
Welcome to Nix version 2.1.3. Type :? for help.

nix-repl> pkgs = import (import ./nixpkgs.nix) {}
```

- The hash references a specific git commit
- If the commit is so old that no binaries are available on `cache.nixos.org`, nix will simply rebuild the packages

Multi-Attribute Derivations

A project's repository can provide multiple packages.

Fictionary `release.nix`

```
{
  a = <derivation>;
  b = <derivation>;
  c = <derivation>;
}
```

- `nix-build -A a` would simply build package a
- With no further `-A` arguments, `nix-build` would build all the derivations

Garbage Collection

Nix has its own garbage collection for the nix store.

```
$ nix-collect-garbage -d
removing old generations of profile /nix/var/nix/profiles/
    per-user/tfc/channels
removing old generations of profile /nix/var/nix/profiles/
    per-user/tfc/profile
removing generation 94
...
deleting '/nix/store/jbchpg8...ac8m56vxs6j67-rpn_calc'
deleting '/nix/store/qpavjqr...9hgcpv1j1g2zi-rpn_calc'
deleting '/nix/store/trash'
deleting unused links...
note: currently hard linking saves 3683.55 MiB
2642 store paths deleted, 3129.22 MiB freed
```

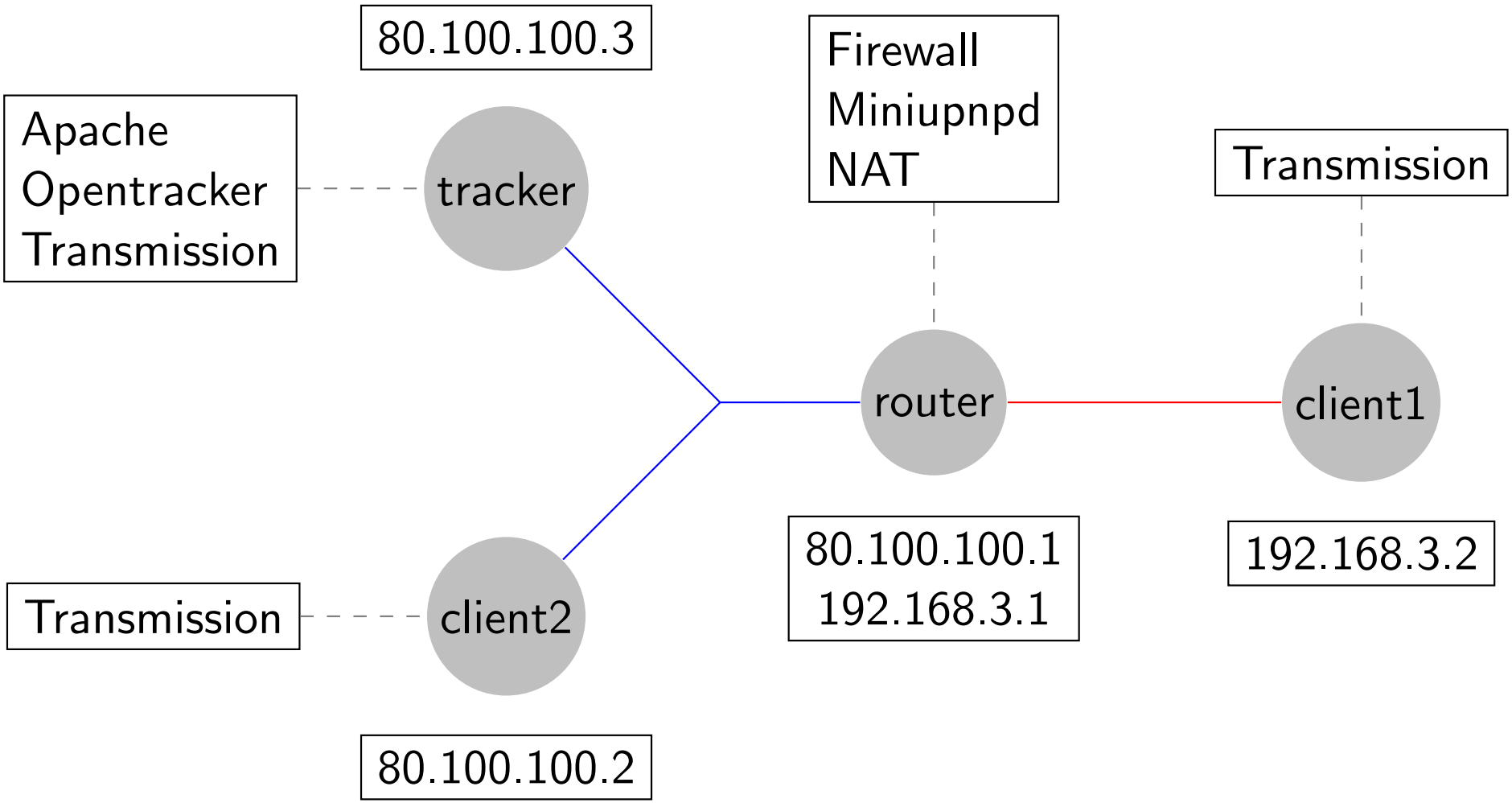
This deletes all derivation outputs in the nix store that are not directly or indirectly referenced on the system.

Further Documentation

- `https://nixos.org/learn.html`
- Nix pills articles: `https://nixos.org/guides/nix-pills/`
- `https://nix.dev/`
- NixOS Wiki: `https://nixos.wiki`
- Nix Manual: `https://nixos.org/nix/manual/`
- Nixpkgs Manual: `https://nixos.org/nixpkgs/manual/`
- NixOS Manual: `https://nixos.org/nixos/manual/`
- NixOS Forum: `https://discourse.nixos.org/`
- IRC: `#nixos`, `#nixos-de` on Freenode

NixOS Integration Test Example: Bittorrent Service

Complex Network&Service Definitions in ∼50 LOC



Section Content

Central Automation of Builds, Tests, and Deployment

Frequent Reassembly of Complex Software Stacks

Complex software products consist of many modules that are developed in parallel. In order to guarantee the full stack's functionality **every day**, they must be frequently reassembled from their parts and tested.
CI-pipelines do this.

Central Automation of Builds, Tests, and Deployment

Frequent Reassembly of Complex Software Stacks

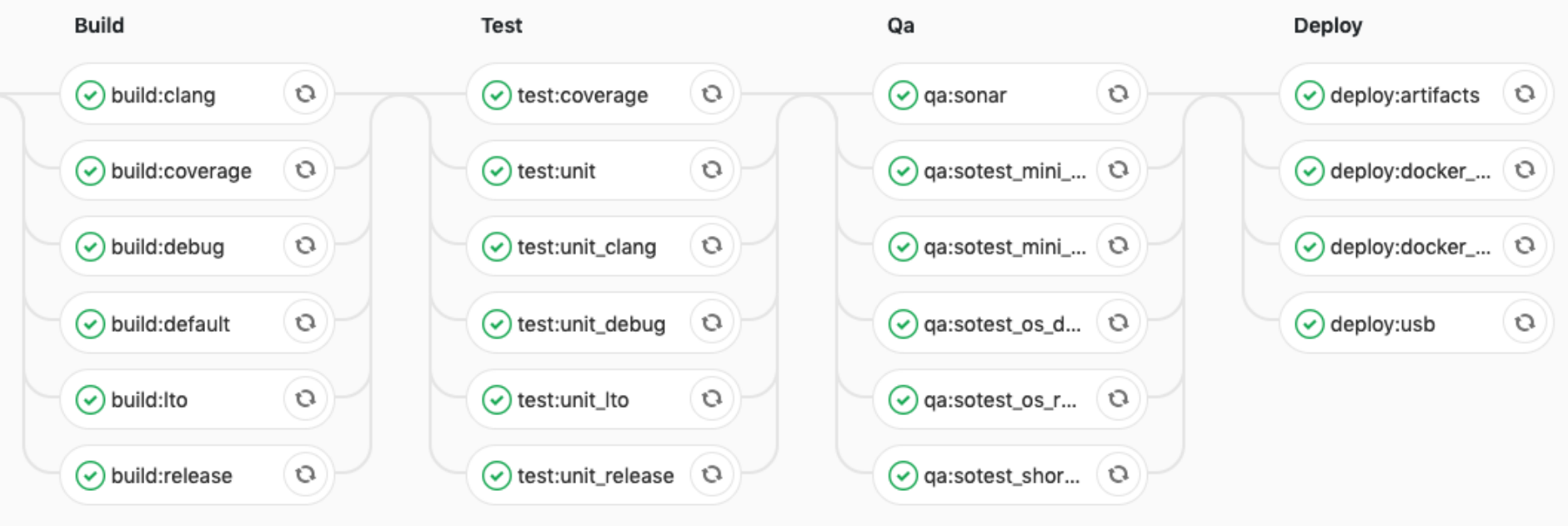
Complex software products consist of many modules that are developed in parallel. In order to guarantee the full stack's functionality **every day**, they must be frequently reassembled from their parts and tested.
CI-pipelines do this.

Typical steps that should be performed by a CI pipeline:

- Build the code, through **multiple** compilers/OSes
- Build and run unit tests with sensible sanitizer configurations
- Perform static analysis, run code metrics
- Execute **all** integration tests possible
- Generate customer-ready images/packages for delivery

Real-Life CI Example

Gitlab Docker Pipeline



Fundamental Project Facts

Automation of Important Tests

Everything that is not automated, will not happen regularly.
Everything that does not happen regularly will break unnoticed.
Everything that breaks unnoticed will become a bad surprise.

Fundamental Project Facts

Automation of Important Tests

Everything that is not automated, will not happen regularly.
Everything that does not happen regularly will break unnoticed.
Everything that breaks unnoticed will become a bad surprise.

Inevitable Quality Gates

Any error that goes through the CI unnoticed will go to QA. QA finds bugs shortly before release $\rightarrow$ over-hours and frustration incoming.
Any error that goes through QA unnoticed will hit the customer. Most QA departments are already at their capacity limit and have problems covering the most basic customer scenario tests.

Fundamental Project Facts

Automation of Important Tests

Everything that is not automated, will not happen regularly.
Everything that does not happen regularly will break unnoticed.
Everything that breaks unnoticed will become a bad surprise.

Inevitable Quality Gates

Any error that goes through the CI unnoticed will go to QA. QA finds bugs shortly before release → over-hours and frustration incoming.
Any error that goes through QA unnoticed will hit the customer. Most QA departments are already at their capacity limit and have problems covering the most basic customer scenario tests.

The Beyonce Rule (Google)

"If you liked it, you should have put a CI test on it."
→ If an untested change caused an incident, it is not the change's fault.

Ultimate Advantages of Continuous Integration

Core Value of a Good CI Setup

Every single commit on the master branch has release-quality

- Can be used as *automatic merge-blocking* **quality gate**
  - Code still builds with all compilers on all supported OSes
  - Unit tests run cleanly with all sanitizers
  - All integration tests still run
  - No code metric got worse
  - Product is in deployable state (Installer, package, . . .)
- CI can do all the things that developers don't have time for or can't do alone
- Infrastructure serves downloads for all intermediate and end products in every version
- Most valuable: Quick deployment after bug fix or security fix? It's only a matter of *hours*!

CI from Developer Perspective

- Project repository contains some kind of `.ci_config.yaml` or similar that describes the CI steps/phases
  - Every step runs in its own environment (e.g. different installed compiler/OS)
  - Every step is described by a series of recipe steps
  - One step's output can be the input of the next step
- Evironment definitions for CI steps differs between CI solutions
  - Some CIs simply have a few execution environments (e.g. VMs) to select, without any variability
  - Old-fashioned/bad CIs are mostly admin-controlled, New-fashioned/good CIs are mostly developer-configurable
  - Most important question is: Which kind of changes do you need to discuss with an admin?
  - If team can't simply change/update toolchains and add new environments, they are longterm unable to maintain quality

Common Problems in the Industrial Practice (1)

- Anything what's not automated will not be done reguarly
  - In many projects, automation is a priority until it's *good enough for now* - at some point it breaks and no one has time to repair it and it remains broken
  - What also happens often is that things are *half-automated* and need to be controlled by persons who might be too busy/in holiday when results are needed
- Changing the CI pipeline configuration is tedious
  - Admins/Ops are usually deadline driven themselsves and rarely have time
  - Simple updates of toolchains can quickly become an unrealistic request in such environments
- Many CIs are very slow, which results in slow builds and testing

Common Problems in the Industrial Practice (2)

- CI may become central "single truth" Build System
  - Individual developer can't reproduce anything locally
  - Errors in CI configuration may be unnoticed due to intransparency
  - Trying to innovate by experimenting with bold changes of the product may be difficult when only the CI can fully build it
  - Frustrating fix-commit-push-wait-retry workflow

Existing CI Solutions

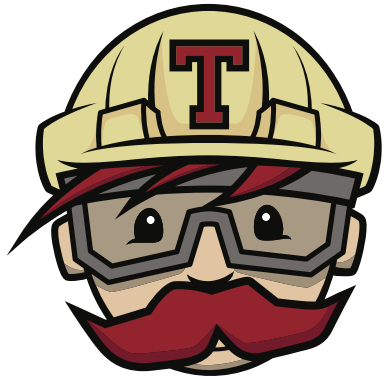
We will have a brief look at a few real-life CI solutions

GitHub	Commercial, free for FOSS projects
GitLab	Commercial, free for FOSS projects
Travis CI	Commercial, free for FOSS projects
Hydra	FOSS Nix community project

GitHub

- Homepage: `https://github.com`
- Proprietary and closed source: American Company founded in 2008, bought by Microsoft in 2018 †
- Nice Web-Interface for Project management, tickets, merge requests, code review, ...
- Provides a *marketplace* for plugin-like services (e.g. Code analysis, merge bots, etc.): `https://github.com/marketplace`
- Provides free public and private (with collaborator limitation) git repositories
- GitHub Enterprise can be licensed as *aaS or on-premise
- Provides CI/CD as *GitHub Actions*
  Developers configure these in their repo: `.github/workflows`

Travis CI



- Homepage: `https://travis-ci.org` for OSS / `https://travis-ci.com` for commercial
- FOSS, German Company founded in 2011
- Free to use for FOSS projects on e.g. GitHub, paid for commercial projects
- Travis CI is configured per-repo and reads and executes the content of `.travis.yml` in each repo
- A collection of programming language/environment specific facilities is provided

GitLab

- Homepage: `https://gitlab.com`
- FOSS, Dutch Company founded in 2014
- Looks like GitHub at first, but then appears as a more centralized *all-in-one* solution
- Provides a Community Edition (CE) and an Enterprise Edition (EE)
- Developers configure the CI via `.gitlab-ci.yml`
- GitLab's builtin CI/CD has strong focus on Docker: Every pipeline job bases on a Docker image and contains stepwise recipe descriptions

Hydra

- Homepage: `https://nixos.org/hydra`
- FOSS, driven by Nix community
- Only works with nix expressions
- Must be self-hosted
  - See also commercial rewrite Hercules CI: `https://hercules-ci.com/`
- `https://hydra.nixos.org` - The whole NixOS Linux distribution is built, tested, and distributed by a single Hydra build farm
- Hydra is **fully declarative**
  - plays a key role in maintaining and providing a whole Linux distribution from the repo `https://github.com/nixos/nixpkgs`
  - Contributor provide merge requests with new package descriptions or updates - after merge, Hydra builds them and they are part of the distro!

Declarativity in CIs

Most significant difference between nix & Hydra and other CI solutions is how they handle initial CI setup and infrastructure dependencies.

CI Configuration User Story

Can the CI be configured just by clicking "New CI project", providing a repo URL and that's it?

The difference becomes apparent via the following perspectives:

- How many steps are needed to configure the CI for a repo mirror somewhere else?
- Do the CI repo-local config files even allow to run the same CI somewhere else?
- Can the CI steps be reproduced locally on a developer's computer?

Which CIs Are Declarative?

- GitLab, GitHub, etc.: **No**
  - Environment description and build steps are separated.
  - Environments are referenced from some kind of *registry* which magically provides some kind of image
  - Example: Required Docker images might be part of the repo, but must be built and installed in the registry before they can be used
  - References to unhashed docker image tags are a also *build impurity* that might become error source for tricky to debug problems

Which CIs Are Declarative?

- GitLab, GitHub, etc.: **No**
  - Environment description and build steps are separated.
  - Environments are referenced from some kind of *registry* which magically provides some kind of image
  - Example: Required Docker images might be part of the repo, but must be built and installed in the registry before they can be used
  - References to unhashed docker image tags are a also *build impurity* that might become error source for tricky to debug problems
- Hydra: **Yes**
  - Environment *and* build steps are **both** described by nix expressions
  - This way a fresh and clean Hydra installation can simply be pointed to a repo's `release.nix` file and has all information
  - No dependency on any kind of external registry
  - Yet not elegantly solved problem: Repository access for private projects with credentials and rights management

The Innovative Value of Declarativity in Product Builds

A fully declarative description of both environment and build steps for builds and tests is the **manifestation of build function** f.

The Innovative Value of Declarativity in Product Builds

A fully declarative description of both environment and build steps for builds and tests is the **manifestation of build function** f.

This empowers developers to:

- Locally build and test anything just as the CI would
- Fully control what the CI does, without depending on Admins/Ops
- Design and test complex CI jobs locally
  $\rightarrow$ Execution in CI just works the same way

The Innovative Value of Declarativity in Product Builds

A fully declarative description of both environment and build steps for builds and tests is the **manifestation of build function** f.

This empowers developers to:

- Locally build and test anything just as the CI would
- Fully control what the CI does, without depending on Admins/Ops
- Design and test complex CI jobs locally
  $\rightarrow$ Execution in CI just works the same way

Hydra and Hercules CI are ahead of time: Innovations like the fully declarative and reproducible builds of `nix` are not widely available in other solutions.