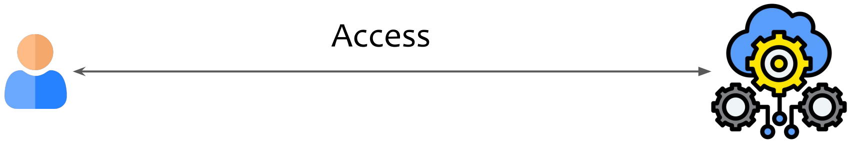
# L10 Software Quality and Project Management

Prof. Pramod Bhatotia

Systems Research Group

https://dse.in.tum.de/

# Roadmap



Access

Clients

Cloud software

**Our focus:**

How to analyze, design, implement, test, and deploy cloud software systems?

L03:
Cloud Design
Requirements

→

L04 + L05 + L06:
System Design and
Implementation

→

L07 + L08:
Software Testing and
Program Analysis

→

L09:
Software Management,
Deployment, & Monitoring

L10: Software Quality and Program management

L03 + L04: Cloud Software Architectures

L02: Cloud Systems Engineering

# Today’s learning goals

- **Part I:** Software quality
  - Software quality management
  - Reviewing
  - // Comments
  - Code refactoring
  - Trustworthy software systems
    - Formal verification
    - Code compliance
- **Part II:** Project management
  - Project management
  - Work breakdown structure
  - Team organization
  - Communication mechanisms

# L10a: Software Quality

Prof. Pramod Bhatotia

Chair of Distributed Systems and Operating Systems

https://dse.in.tum.de/

# Outline

- **Part I: Software quality**
  - Software quality management
  - Reviewing
  - // Comments
  - Code refactoring
  - Trustworthy software systems
    - Formal verification
    - Code compliance

# Software quality

- Simply, **quality** means a **product should meet its specification**

- “Conformance to explicitly stated **functional and nonfunctional requirements**, explicitly documented **development standards**, and **implicit characteristics** that are expected of all professionally developed software"

– Software Engineering: A Practitioner's Approach

# Software quality management

- **Software quality management** ensures that the required level of quality is achieved in a software product
  - Involves **defining appropriate quality standards and procedures**
  - Ensures that these **procedures are followed**
  - Aims to **develop a quality culture** where quality is seen as everyone’s responsibility

# Challenges for quality in software systems

- **Software specifications** are usually incomplete and often inconsistent
  - Some **quality requirements** are difficult to specify in an unambiguous way

- **Tension** between customer quality requirements (efficiency, reliability, etc.) and developer quality requirements (maintainability, reusability, etc.)

- **Quality requirements** change over time

# Best practices for software quality

Review

Comments

Code refactoring

Trustworthy
software systems

# Best practices for software quality

Review

Comments

Code refactoring

Trustworthy software systems

# Reviewing

- **Reviews** in software engineering:
  - A **process or meeting** during which an artifact is examined to **discover any flaws** and **ensure a certain level of quality is met**
- **Major review types**
  - Requirements analysis review
  - Design review
  - Code review

# Best practices for software quality

Review

Comments

Code refactoring

Trustworthy
software systems

# Requirements analysis reviews

- **Objective:** To make sure that the requirements specification is correct, complete, consistent, unambiguous, realistic and verifiable
    - Developers should be prepared to discover errors and make changes to the specification
    - The review can be facilitated by a checklist or a list of questions

# Design reviews

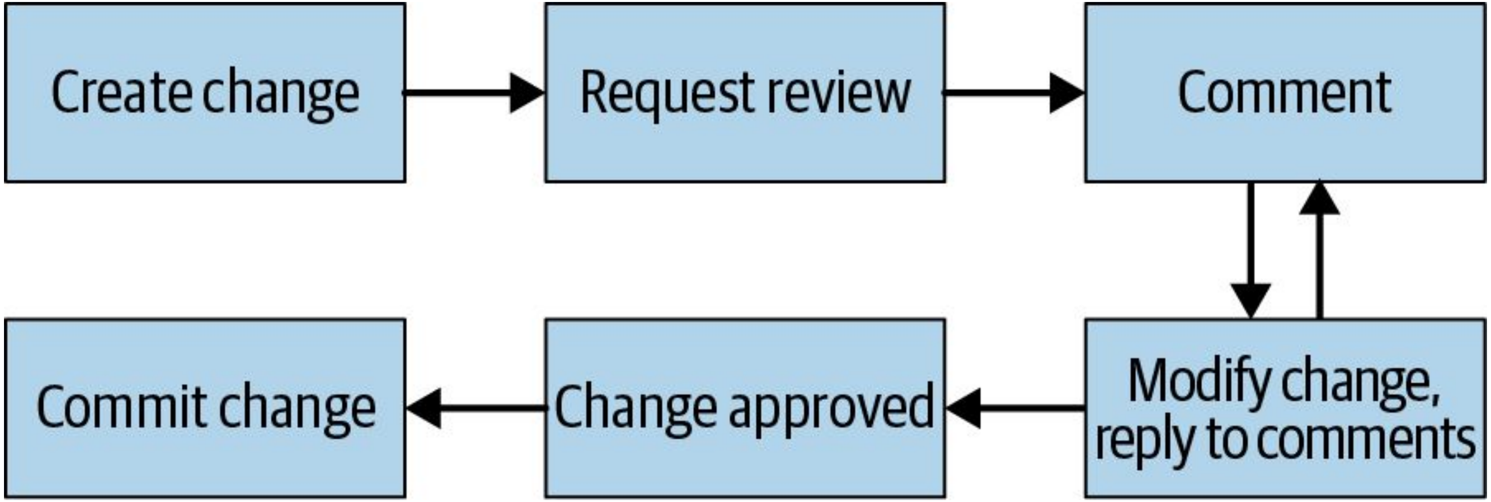
- **Objectives:** To ensure that the system design is correct, complete, consistent, realistic, and readable
  - **Review the design goals** that were identified during system design

- Ask the following questions to determine if the system design is correct
  - Can every subsystem be traced back to a use case or a nonfunctional requirement?
  - Can every use case be mapped to a set of subsystems?
  - Can every design goal be traced back to a nonfunctional requirement?

# Code reviewing

- **Code review** is a process in which code is reviewed by someone other than the author, often before the introduction of that code into a codebase
- **Main goals of code review:**
  - To improve the readability and maintainability of the code base

# Code review workflow



Create change

Request review

Comment

Modify change, reply to comments

Change approved

Commit change

# Code reviews

**Advantages:**

- Improved code quality
- Knowledge transfer
- Improved developer communication and culture

**Disadvantages:**

- Higher costs
- Slower development

**Note:**

- **Static code analysis** can automate repetitive aspects of code reviews
- However, **manual code reviews** are still needed and useful
- **Best practice:** only review code manually if all automatic checks have passed

# References

- Software Engineering at Google
  - Chapter 9: Code Review
    - https://abseil.io/resources/swe-book/html/ch09.html
  - Chapter 19: Critique: Google’s Code Review Tool
    - https://abseil.io/resources/swe-book/html/ch19.html

# Best practices for software quality

Review

Comments

Code refactoring

Trustworthy
software systems

# Why are comments needed?

- **Code alone can't represent cleanly all the information** in the mind of the designer
  - Even if information could be deduced from code, it might be time-consuming
- **Comments provide clarity** that reduces complexity
  - E.g., make abstractions more clear
- **Comments are still controversial (!)**
  - A significant fraction of all commercial code is uncommented!

# The four excuses

- **"This code is self-documenting"**
  - A myth!
- **"Comments get out of date and become misleading"**
  - Just remember to update them!
- **"I don't have time to write comments"**
  - Future investment
- **"The comments I have seen are worthless; why bother?"**
  - Learn to write good comments

# //Comments: The golden rule!

**Comments should describe things that aren’t obvious from the code**

- **Mistake #1:** Comments duplicate code
- **Mistake #2:** Non-obvious info is not described

# What should go in comments?

- **Higher-level information (capture the intuition):**
  - **Abstractions:** a higher-level description of what the code is doing
  - **Rationale for the current design:** why the code is this way?
  - **Invariants to preserve:** What are the invariants?
- **Lower level details (especially for variables, arguments, return values):**
  - Exactly what is this thing? What are the units?
  - Boundary conditions: Does "end" refer to the last value, or the value after the last one?
  - Is a null value allowed? If so, what does it mean?
  - If memory is dynamically allocated, who is responsible for freeing it?
  - How to choose the value of a configuration parameter

# Best practices for //comments

- **Document each thing exactly once**
  - Don't duplicate documentation (it won't get maintained)
  - Use references rather than repeating documentation: "See documentation for xyz method"
- **Put documentation as close as possible to the relevant code**
  - Next to variable and method declarations
  - Push in-method documentation down to the tightest enclosing context
- **Don't say anything more in documentation than you need to**
  - E.g., don't use comments in one place to describe design decisions elsewhere
  - Higher-level comments are less likely to become obsolete
- **Look for "obvious" locations** where people can easily find the documentation

# References

- Book: A philosophy of software design by John Ousterhout
  - **Chapter 12:** Why write comments? The four excuses
  - **Chapter 13:** Comments should describe things that aren't obvious from the code
  - **Chapter 15:** Write the comments first
- Lecture notes:
  - https://web.stanford.edu/~ouster/cgi-bin/cs190-spring16/lecture.php?topic=comments

# Best practices for software quality

Review

Comments

Code refactoring

Trustworthy software systems

# Code smells, please refactor it to make it cleaner!

- **Code smells**, please refactor it!
- **Clean code:**
    - Obvious for other programmers
    - Doesn’t contain duplication
    - Contains a minimal number of classes and other moving parts
    - Passes all tests
    - Easier and cheaper to maintain!

# When to refactor?

- **Rule of three**
  - When you’re doing something **for the first time, just get it done**!
  - When you’re doing something similar **for the second time, cringe at having to repeat** but do the same thing anyway
  - When you’re doing something **for the third time, start refactoring!**
- **More broadly: Take every chance!**
  - When adding a new feature
  - When fixing a bug
  - When doing a code review

# Checklist for refactoring

- **The code should become “cleaner”**
  - See the next slides for “Code smells”
- **New functionality shouldn’t be created during refactoring**
  - Don’t mix refactoring and direct development of new features
  - Try to separate these processes at least within the confines of individual commits.
- **All existing tests must pass after refactoring**
  - Don’t break things!

# Examples of a smelly code!

- **Bloaters**
  - Bloaters are code, methods and classes that have increased to such gargantuan proportions that they are hard to work with
- **Change Preventers**
  - Change in one place leads to make make many changes in other places too!
- **Dispensables**
  - Something pointless and unneeded → Can be removed for clean, efficient, and easier to understand code
- **Couplers**
  - Excessive coupling between classes/subsystems

# References

- Code refactoring:
  - https://refactoring.guru/refactoring

# Best practices for software quality

Review

Comments

Code refactoring

Trustworthy software systems

# Trustworthy software systems

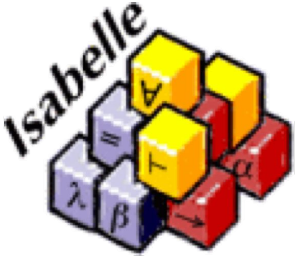
- Ubiquitous use of complex software in cyber-physical systems
  - Software quality is of at most importance in this domain
  - **Reliable and secure software systems**
- **Why testing is not the definitive answer?**
  - Testing shows the presence of errors, in general, not their absence
  - How to test for the unexpected? Rare cases?
- **Software verification: An approach for building “correct-by-construction” systems**
  - Increasingly adopted in the software industry!

# Formal verification

- **Formal verification**
  - **Specify** the correctness of the system formally
  - **Prove** that the implementation conforms to the specification

- If the specification expresses your correctness property, then your system is correct, subject to the assumptions you have made during the proof

- Proof assistants:



https://isabelle.in.tum.de/

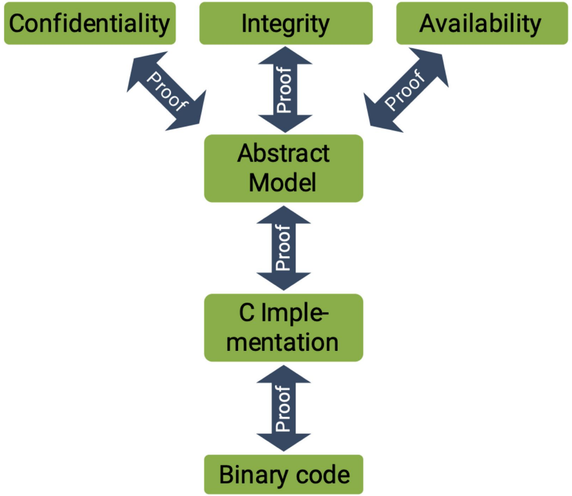
https://coq.inria.fr/

# seL4: A case-study

- **SeL4: A formally verified operating system**
    - Trustworthy foundation for safety- and security-critical systems
    - Project webpage: [https://sel4.systems/About/](https://sel4.systems/About/)
- What does it mean to be a formally verified OS?
    - **seL4's implementation** is formally (mathematically) proven correct (bug-free) against its specification seL4 comes with a formal, mathematical, machine-checked proof of implementation correctness

# seL4 verification process

- **Functional correctness**
  - Formal specification of OS kernel's functionality in higher-order logic (abstract model)
  - Functional correctness proof states that C implementation is a refinement of the abstract model, i.e., all possible behaviors of C code are a subset of the abstract model

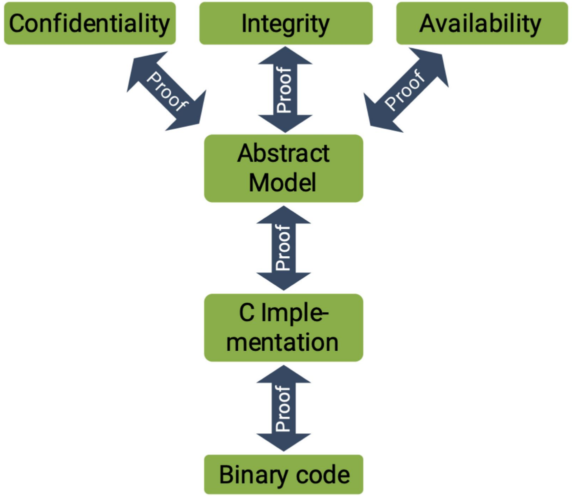


Reference: https://sel4.systems/About/seL4-whitepaper.pdf

# seL4 verification process

## Security correctness

- **Confidentiality:** seL4 will not allow an entity to read data without having been explicitly given access
- **Integrity:** seL4 will not allow an entity to modify data without having been explicitly given access
- **Availability:** seL4 will not allow an entity to prevent another entity's authorised use of resources



# Code compliance checkers

- **Formal verification has its limits**
  - Not scalable to large software systems
  - Also, very expensive!!
- **Cyber-physical systems require high-quality software**
  - Automotive, aerospace, medical devices, telecommunications, etc.
- **Code compliance** is a “best effort” approach to build trustworthy systems
  - For e.g., automotive industry standards: MISRA, AutoSar
    - A set of standards and guidelines for C and C++ programs
    - Reliable enough to run in safety-critical systems
    - Secure against common code exploits
    - Portable (reusable) throughout the supply chain
  - **Static analyzer for code-compliance**
    - PC-Lint: https://www.gimpel.com/
    - Cppcheck: http://cppcheck.sourceforge.net/

# References

- seL4: https://sel4.systems/
- Introduction to formal verification in software systems:
  - https://www.moritz.systems/blog/an-introduction-to-formal-verification/
- How Amazon Web services uses formal verification
  - https://cacm.acm.org/magazines/2015/4/184701-how-amazon-web-services-uses-formal-methods/abstract

# Summary

- **Part I:** Software quality
  - Software quality management
  - Reviewing
  - // Comments
  - Code refactoring
  - Trustworthy software systems
    - Formal verification
    - Code compliance