CS3213 Project – Week 12

Summary | 06-04-2022

- ❑ Survey on “Interactive Repair”
- ❑ Recap: All Topics (Requirements to Integration)
- ❑ Aspects of Version Control

The slides contain additional comments in such yellow boxes.

Topics

The shown topics have been discussed throughout the lecture's project part and in the labs. The following slides will recap them and emphasize on important aspects.

- ❑ Requirements Elicitation – Analysis Techniques (Questions, Interviews, …)
- ❑ Requirements Modeling – GORE, Use Case, Activity Diagram, etc.
- ❑ Architecture: Architectural Drivers, Structures (Static + Dynamic), Architectural Styles
- ❑ Module Design: Design Pattern, Design Principles
- ❑ Project Planning: Work Packages, Gantt-Charts, PERT, Milestone Trend Analysis
- ❑ Implementation: Clean Code
- ❑ Testing: Foundations + Equivalence Class Partitioning, JUnit, TDD, Testable Code, Clean Tests, Code/Test Coverage, Static Analyzers (Checkstyle, Spotbugs, PMD)
- ❑ Debugging: TRAFFIC, Reducing the input (delta debugging), Reducing the program (slicing), SBFL/SFL, Interactive Debugging
- ❑ Software Integration Strategies and Integration Testing

Assignments

- ❑ A1 – Requirements Analysis & Elicitation
- ❑ A2 – Requirements Modeling
- ❑ A3 – Behavioral Modeling & Architectural Drivers
- ❑ A4 – Module Design / Strategy Plan
- ❑ A5 – Project Planning
- ❑ A6 – Intermediate Deliverable
- ❑ A7 – Unit Testing
- ❑ A8 – Presentation + Final Code
- ❑ A9 – Final Report

With the assignments, we covered many important aspects. The common mistakes have been discussed in the labs (you can check these slides separately), and individual feedback has been given via LumiNUS.

Topics

- ❑ **Requirements Elicitation – Analysis Techniques (Questions, Interviews, …)**
- ❑ Requirements Modeling – GORE, Use Case, Activity Diagram, etc.
- ❑ Architecture: Architectural Drivers, Structures (Static + Dynamic), Architectural Styles
- ❑ Module Design: Design Pattern, Design Principles
- ❑ Project Planning: Work Packages, Gantt-Charts, PERT, Milestone Trend Analysis
- ❑ Implementation: Clean Code
- ❑ Testing: Foundations + Equivalence Class Partitioning, JUnit, TDD, Testable Code, Clean Tests, Code/Test Coverage, Static Analyzers (Checkstyle, Spotbugs, PMD)
- ❑ Debugging: TRAFFIC, Reducing the input (delta debugging), Reducing the program (slicing), SBFL/SFL, Interactive Debugging
- ❑ Software Integration Strategies and Integration Testing

Requirements

We started the project with **requirements**, in particular, with their elicitation in our customer interview session. Note that ”getting the requirements right” is one of the key difficulties in software engineering. It requires proper Requirements **Analysis** & **Elicitation.**

“The hardest single part of building a software system is **deciding** precisely **what** to build. No other part of the conceptual work is as difficult as establishing the detailed **technical requirements** ... No other part of the work so cripples the resulting system if done wrong. **No other part is as difficult to rectify later**.”

➡ **Requirements Analysis & Elicitation**

Brooks, F. P., “No silver bullet – essence and accidents of software engineering” in IEEE Computer, Vol. 20 (4), 10-19, 1987.

Requirement Analysis Techniques

Main Focus

Analysis Technique | "As-Is" State | "To-Be" State | Innovation Impact

Analysis of existent data and documents

Observation

Survey with { closed / structured / open } questions

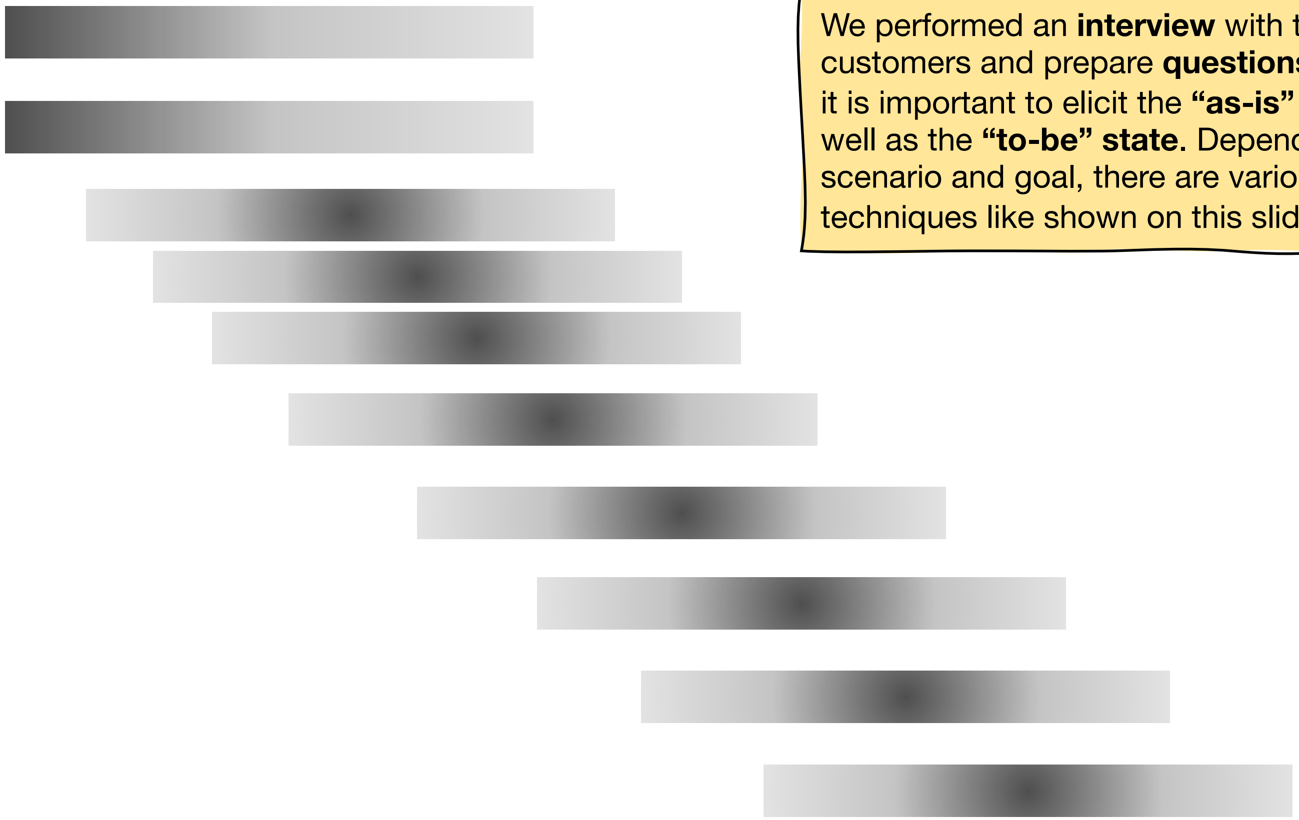
Interview

Modelling

Experiments

Prototyping

Participative Development (wrt analysis)



We performed an **interview** with the customers and prepare **questions.** Note that it is important to elicit the **"as-is" state** as well as the **"to-be" state**. Depending on the scenario and goal, there are various analysis techniques like shown on this slide.

How to find good questions?

- Which topics need to be covered in the requirement specification?
  - Purpose of the software
  - Functional Requirements
  - Requirements to External Interface
  - Requirements Regarding Technical Data
  - General Constraints and Requirements
  - Product Quality Requirements

1. Introduction
- 1.1 Purpose
- 1.2 Scope
- 1.3 Product overview
  - 1.3.1 Product perspective
  - 1.3.2 Product functions
  - 1.3.3 User characteristics
  - 1.3.4 Limitations
- 1.4 Definitions

2. References

3. Requirements
- 3.1 Functions
- 3.2 Performance requirements
- 3.3 Usability requirements
- 3.4 Interface requirements
- 3.5 Logical database requirements
- 3.6 Design constraints
- 3.7 Software system attributes
- 3.8 Supporting information

4. **Verification**
- (parallel to subsections in Section 3)

5. Appendices
- 5.1 Assumptions and dependencies
- 5.2 Acronyms and abbreviations

SRS outline (IEEE 29148:2018)

Good questions can be found by looking into the relevant aspects of a **specification.** The result of the requirements engineering phase will be the requirement specification, so it makes sense to think about which questions need to be answered by the stakeholders to write such a specification.

ISO/IEC/IEEE 29148:2018, ISO/IEC/IEEE International Standard - Systems and software engineering -- Life cycle processes -- Requirements engineering, 2018, DOI: 10.1109/IEEESTD.2018.8559686 → page 67 to 74

Requirement Elicitation – Closing Remarks

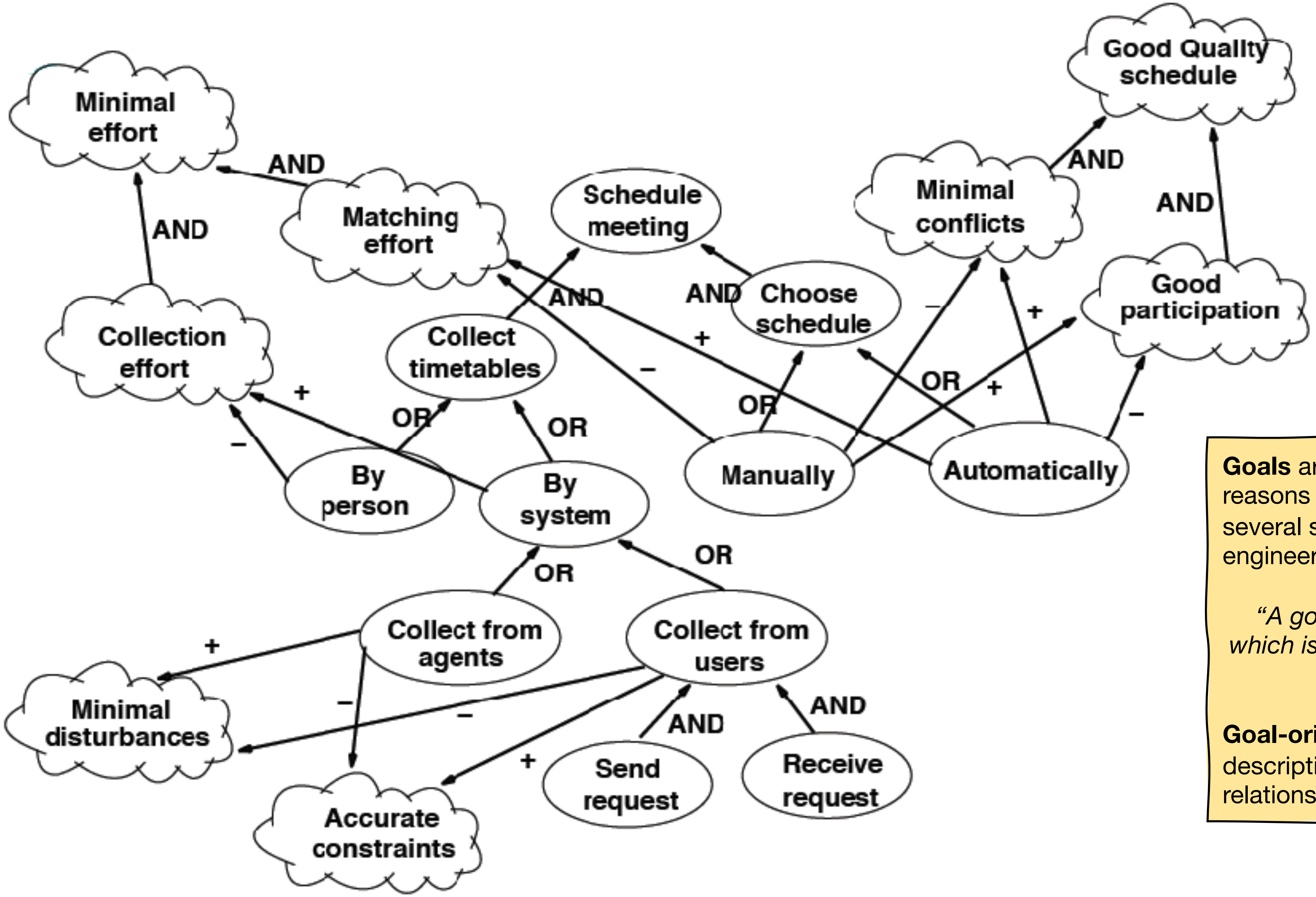
These remarks have been discussed after our requirement elicitation session with the customer.

- ❑ begin **gentle** and proceed with **caution**
- ❑ prepare your **catalogue of questions** and ask **systematically**
- ❑ reveal **contradictions**
- ❑ **special cases** usually require more effort as the default case – you need to explore **all eventualities** in the system **with** the customer
- ❑ do not forget the **„as-is“ state**
- ❑ Jewish motherhood (example of the *door access system*)

Topics

- ❑ Requirements Elicitation – Analysis Techniques (Questions, Interviews, …)
- ❑ **Requirements Modeling – GORE, Use Case, Activity Diagram, etc.**
- ❑ Architecture: Architectural Drivers, Structures (Static + Dynamic), Architectural Styles
- ❑ Module Design: Design Pattern, Design Principles
- ❑ Project Planning: Work Packages, Gantt-Charts, PERT, Milestone Trend Analysis
- ❑ Implementation: Clean Code
- ❑ Testing: Foundations + Equivalence Class Partitioning, JUnit, TDD, Testable Code, Clean Tests, Code/Test Coverage, Static Analyzers (Checkstyle, Spotbugs, PMD)
- ❑ Debugging: TRAFFIC, Reducing the input (delta debugging), Reducing the program (slicing), SBFL/SFL, Interactive Debugging
- ❑ Software Integration Strategies and Integration Testing

GORE Modeling



Goals are introduced for pragmatic or engineering reasons – they help accomplish the objectives of several specific subtasks of requirements engineering.

"A goal is a desirable state lying in the future, which is not reached automatically but by specific actions."

Goal-oriented analysis focuses on the description and evaluation of alternatives and their relationship to the organizational objectives.

Common Modeling Purposes

- clarifying requirements
  - modeling techniques need to support "why" and "how else" types of reasoning analysis
  - incremental process
- provide traceability of rationales
- management of change
- verification of achievement of requirements
- support of reuse

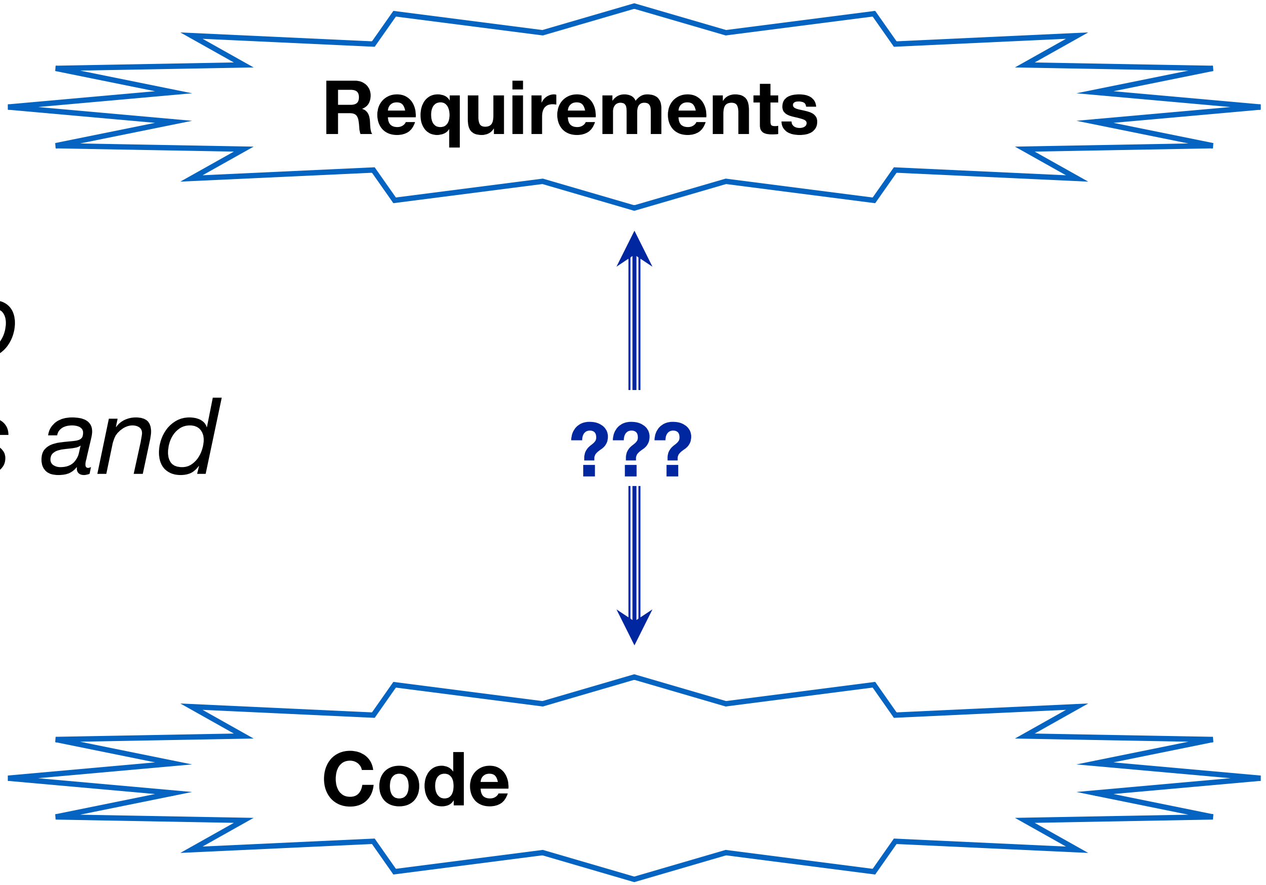
Remember: ***Different models have different purposes***. We looked into use case models, goal models, activity models and more. You should be aware of all these models, their syntax, semantics and purpose.

Topics

- ❏ Requirements Elicitation – Analysis Techniques (Questions, Interviews, …)
- ❏ Requirements Modeling – GORE, Use Case, Activity Diagram, etc.
- ❏ **Architecture: Architectural Drivers, Structures (Static + Dynamic), Architectural Styles**
- ❏ Module Design: Design Pattern, Design Principles
- ❏ Project Planning: Work Packages, Gantt-Charts, PERT, Milestone Trend Analysis
- ❏ Implementation: Clean Code
- ❏ Testing: Foundations + Equivalence Class Partitioning, JUnit, TDD, Testable Code, Clean Tests, Code/Test Coverage, Static Analyzers (Checkstyle, Spotbugs, PMD)
- ❏ Debugging: TRAFFIC, Reducing the input (delta debugging), Reducing the program (slicing), SBFL/SFL, Interactive Debugging
- ❏ Software Integration Strategies and Integration Testing

Comments to Software Architecture

How to bridge the gap between requirements and code?



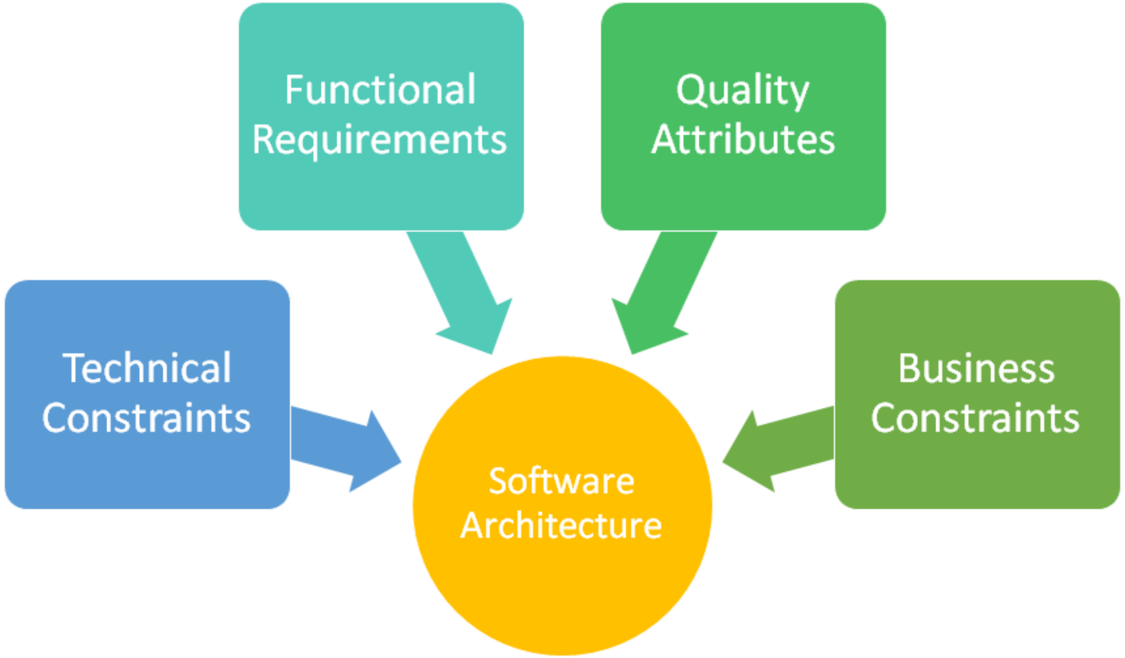
After various aspects of requirements engineering we looked briefly into the problem of **software architecture** and later also discussed some **architectural styles** in more detail.

The architecture becomes the **bridge** between requirements and implementation. Without proper architecture, the implementation is unpredictable and costly. Architecture becomes the tool for managing the **complexity**.

Architectural Drivers

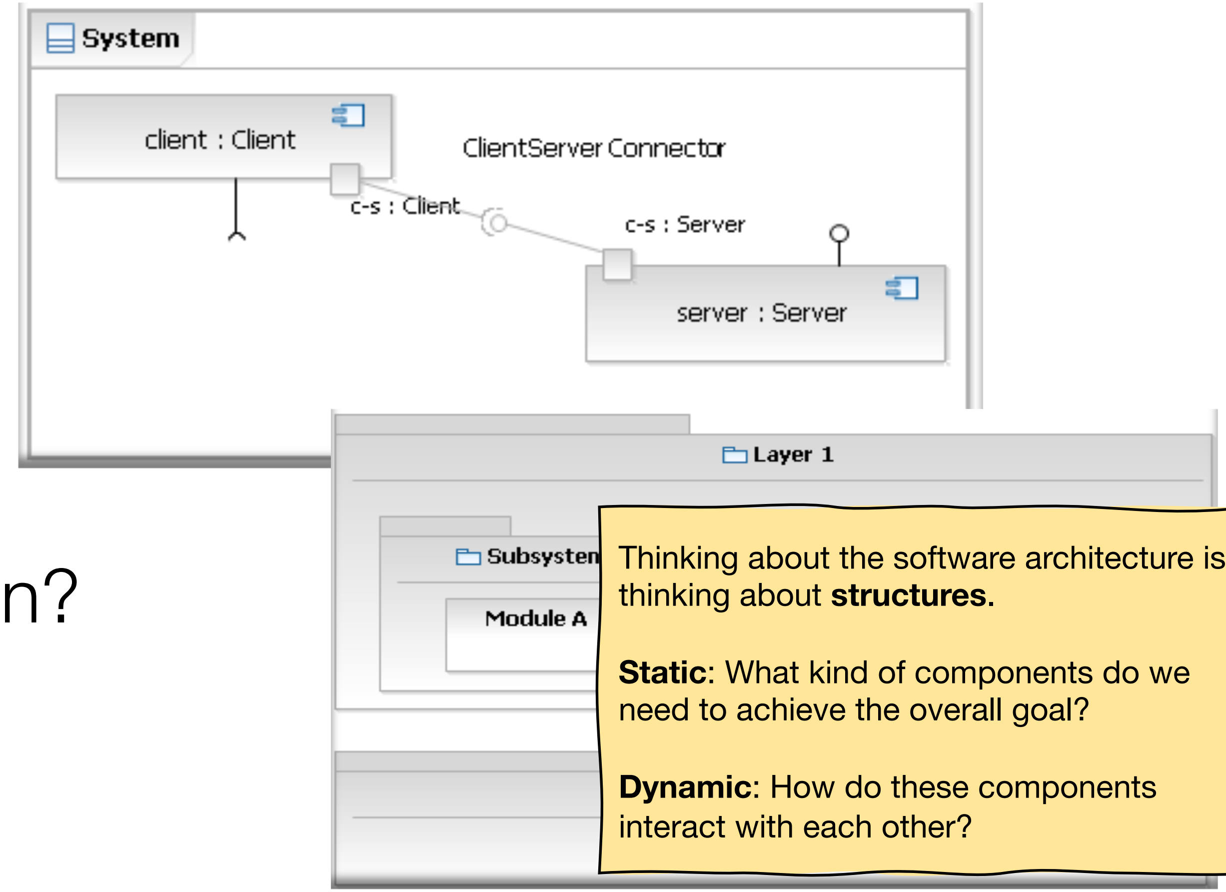
Before thinking about concrete architecture, you should think about the **key requirements** that may **drive** the **architecture**. Such requirements require specific architectures to be supported or put general constraints on the solution space.

- ❑ **Business goals**
  - ❑ Customer organization
  - ❑ Developing organization
- ❑ **Quality attributes**
- ❑ **Key functional requirements**
  - ❑ Unique properties
  - ❑ Make system viable
- ❑ **Constraints**
  - ❑ Organizational and technical
  - ❑ Cost and time



https://medium.com/@janerikfra/architectural-drivers-in-modern-software-architecture-cb7a42527bf2

Stuctures!



- ❑ What elements are there?
- ❑ How are they interconnected?
- ❑ What does the connection mean?

Thinking about the software architecture is thinking about **structures**.

Static: What kind of components do we need to achieve the overall goal?

Dynamic: How do these components interact with each other?

Overall conceptual idea:
→ Each part can be built fairly independently of the other parts
→ However, these parts must be put together to solve the larger problem in the end

Architecture Essentials – Design Principles

Architectural Design Principles and **Styles** are important aspects for the design of the overall system. You should be aware of the most common principles and styles so that you can join **discussions** in practice.

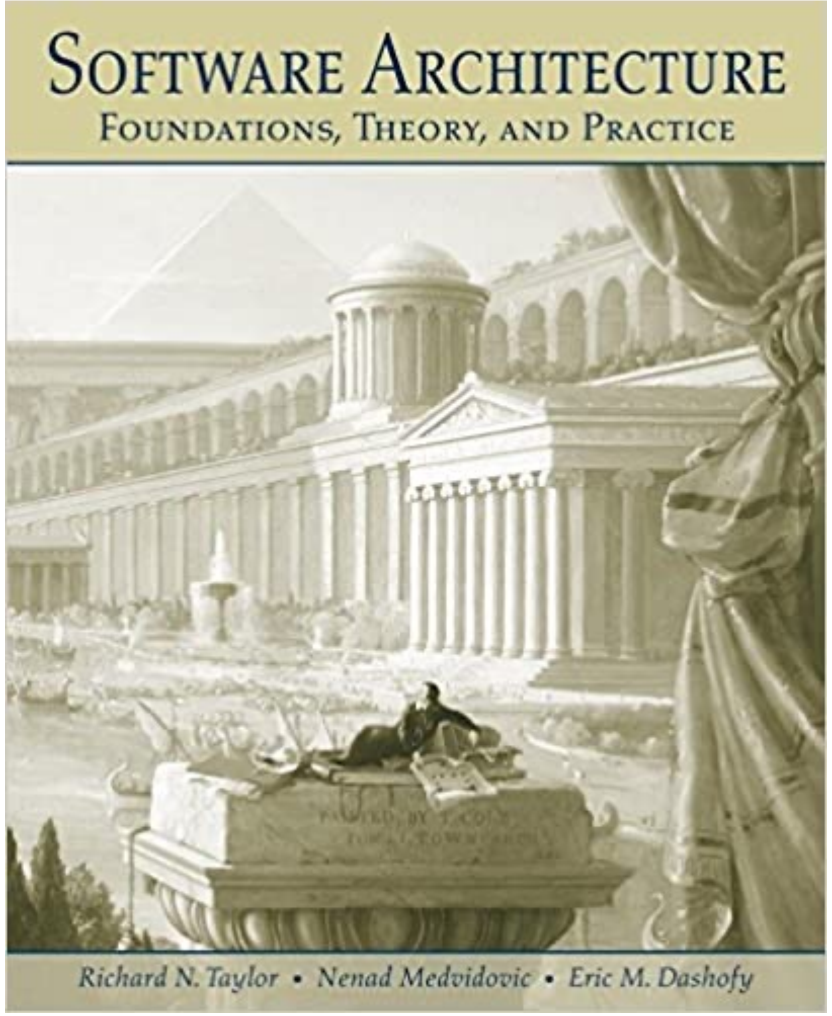
- ❑ Abstraction
- ❑ Separation of Concerns
- ❑ Decomposition: divide & conquer
- ❑ Modularization: coupling & cohesion
- ❑ Encapsulation: information hiding
- ❑ Well-Defined Interfaces
- ❑ Architectural Styles
  - Pipe-and-Filter
  - Shared-Data
  - Publish-Subscribe
  - Client Server Style
  - Peer-to-Peer Style
  - Communicating-Processes Style

There is more!

Note: We only covered a **small portion** of **software architecture literature**.

- **Pipe-and-Filter**
- Shared-Data
- **Publish-Subscribe**
- **Client Server Style**
- Peer-to-Peer Style
- Communicating-Processes Style

"Software Architecture: Foundations, Theory, and Practice"
by Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; 2008 John Wiley & Sons, Inc.

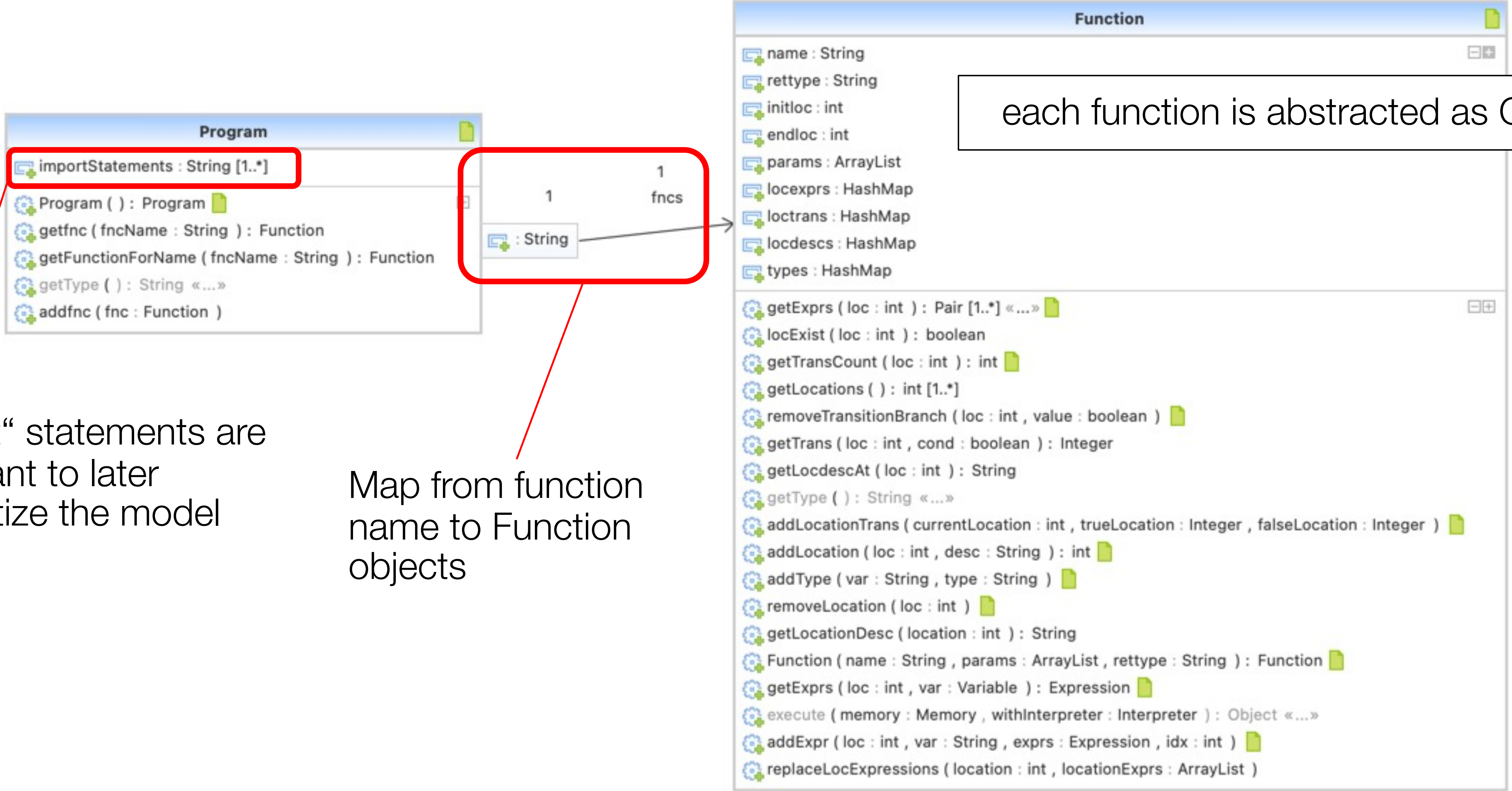


Topics

- ❑ Requirements Elicitation – Analysis Techniques (Questions, Interviews, …)
- ❑ Requirements Modeling – GORE, Use Case, Activity Diagram, etc.
- ❑ Architecture: Architectural Drivers, Structures (Static + Dynamic), Architectural Styles
- ❑ **Module Design: Design Pattern, Design Principles**
- ❑ Project Planning: Work Packages, Gantt-Charts, PERT, Milestone Trend Analysis
- ❑ Implementation: Clean Code
- ❑ Testing: Foundations + Equivalence Class Partitioning, JUnit, TDD, Testable Code, Clean Tests, Code/Test Coverage, Static Analyzers (Checkstyle, Spotbugs, PMD)
- ❑ Debugging: TRAFFIC, Reducing the input (delta debugging), Reducing the program (slicing), SBFL/SFL, Interactive Debugging
- ❑ Software Integration Strategies and Integration Testing

its-core: Program model (1/3)

We looked into the **module design** of our ITS baseline.

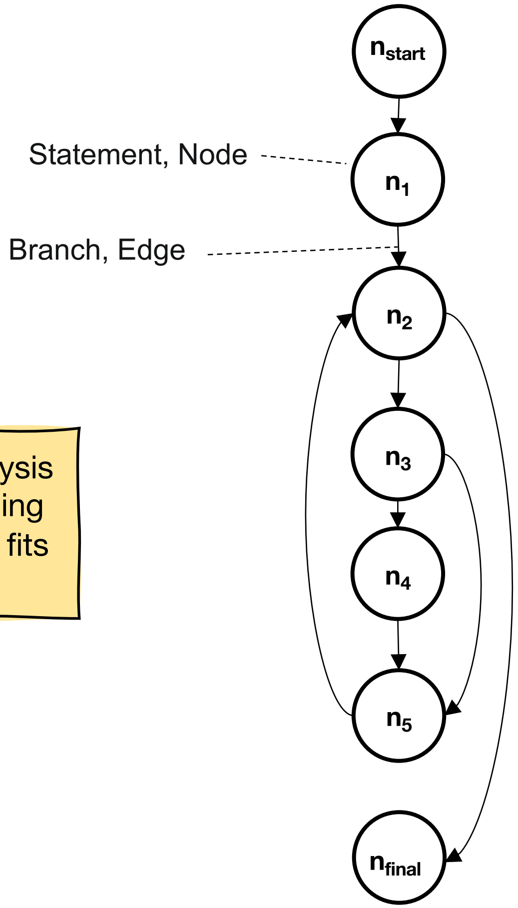


each function is abstracted as CFG

„Import“ statements are important to later concretize the model

Map from function name to Function objects

Control Flow Graph (CFG) Example



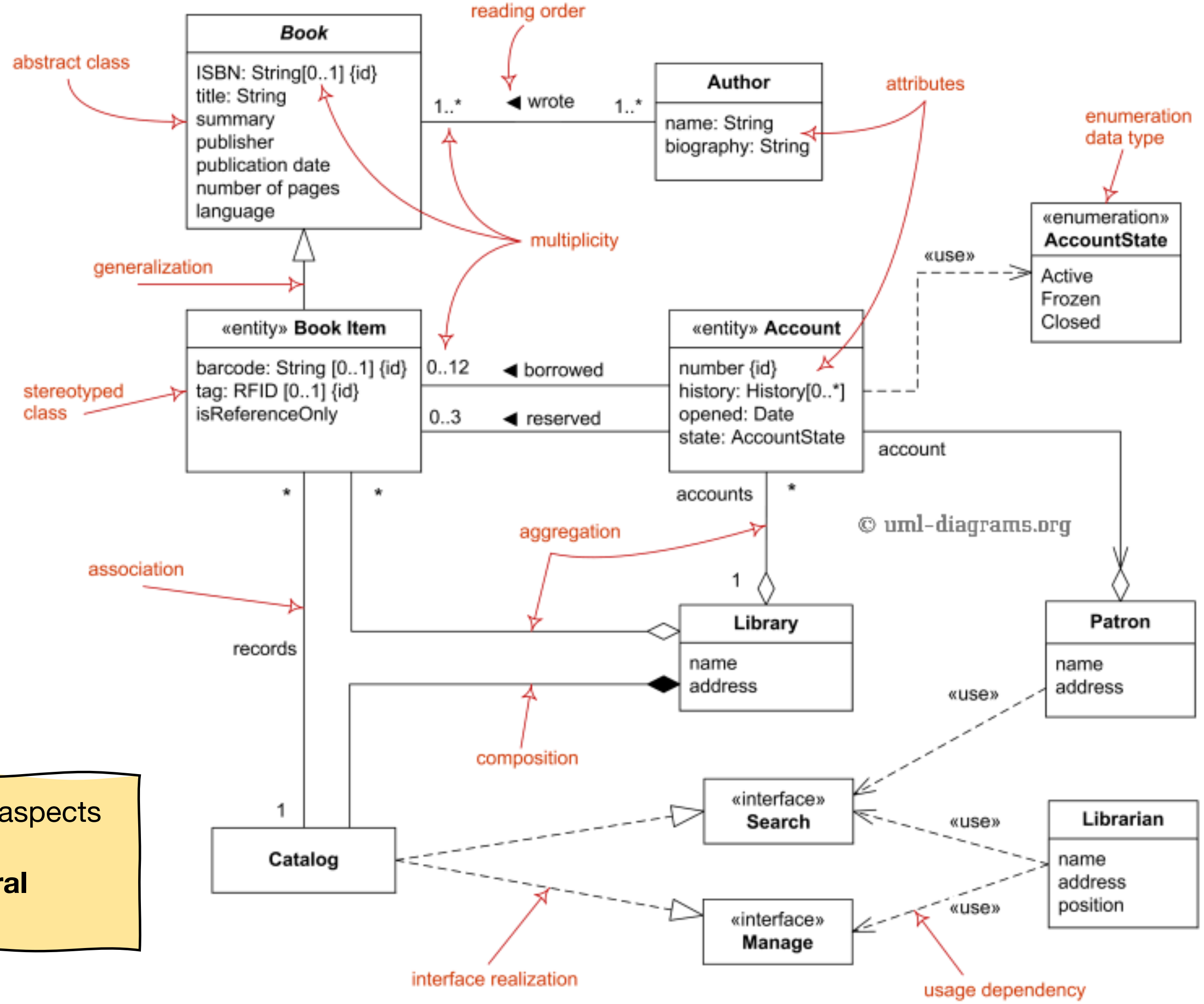

```
void CountChar (int &ACount, int &TotalCount)
{

  char c;
  cin >> c;


  while ((c >= 'A') && (c <= 'Z') && (TotalCount < INT_MAX))
  {


    TotalCount = TotalCount + 1;
    if ((c == 'A'))
    {

      ACount = ACount + 1;

    }

    cin >> c;
  }


}
```

We discussed **basics** in program analysis like the **control flow graph**, the meaning **single static assignment**, and how it fits to our **internal program model.**

Discussion: Module Designs (1/2)

- Be aware of the syntactical elements in UML Class diagrams.
- Class diagrams, on the macro level, have:
  - Classes
  - Associations
  - Aggregations ('has-a' relationship)
  - Compositions ('part-of' relationship)
  - Inheritances
- Class diagrams, on the micro level, have:
  - Attributes / Fields
  - Operations
  - Abstract and concrete operations/classes

We recapped important aspects of **module design** with **structural** and **behavioral diagrams.**



Design Principles

- SOLID
  - S – Single Responsibility Principle
  - O – Open/Closed Principle
  - L – Liskov Substitution Principle
  - I – Interface Segregation Principle
  - D – Dependency Inversion Principle

One class should have one and only one responsibility.

Software components should be open for extension, but closed for modification

Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.

Keep interface as small as possible.

Depend on abstractions, not on concretions.

- GRASP – General Responsibility Assignment Software Principles e.g., high cohesion & low coupling
- DRY vs WET "don't repeat yourself" vs "write everything twice"/"waste everyone's time"
- KISS: "Keep it simple, stupid" and YAGNI: "You aren't gonna need it"

Design principles are general advices that help you to keep your design clean from typical mistakes.

Design Patterns (Gamma et al., 1995)

We covered the basics for **design patterns** and covered some of them in the lecture as well as in the labs. Note that it is not the goal to include *as many patterns as possible* but to use them **appropriately**. It is about to **know** them and to use them correctly in **discussions**.

- **Creational** Patterns
  - Instantiation of objects
  - Example: Singleton
- **Structural** Patterns
  - Solution of distinct structuring problems
  - Example: Composite
- **Behavioral** Patterns
  - Solution of specific behavior aspects
  - Example: Visitor

Pattern Description Language/Structure

1. *Intent*
2. *Motivation*
3. *Applicability*
4. *Structure*
5. *Participants*
6. *Collaborations*
7. *Consequences*
8. *Implementation / Sample Code*
9. *Related Patterns*

When is the module design **finished**?
You can ask yourself:
- Can you start coding now?
- Can the work effort by distributed in you team?

Some design parts will likely change during development… Interfaces should not.

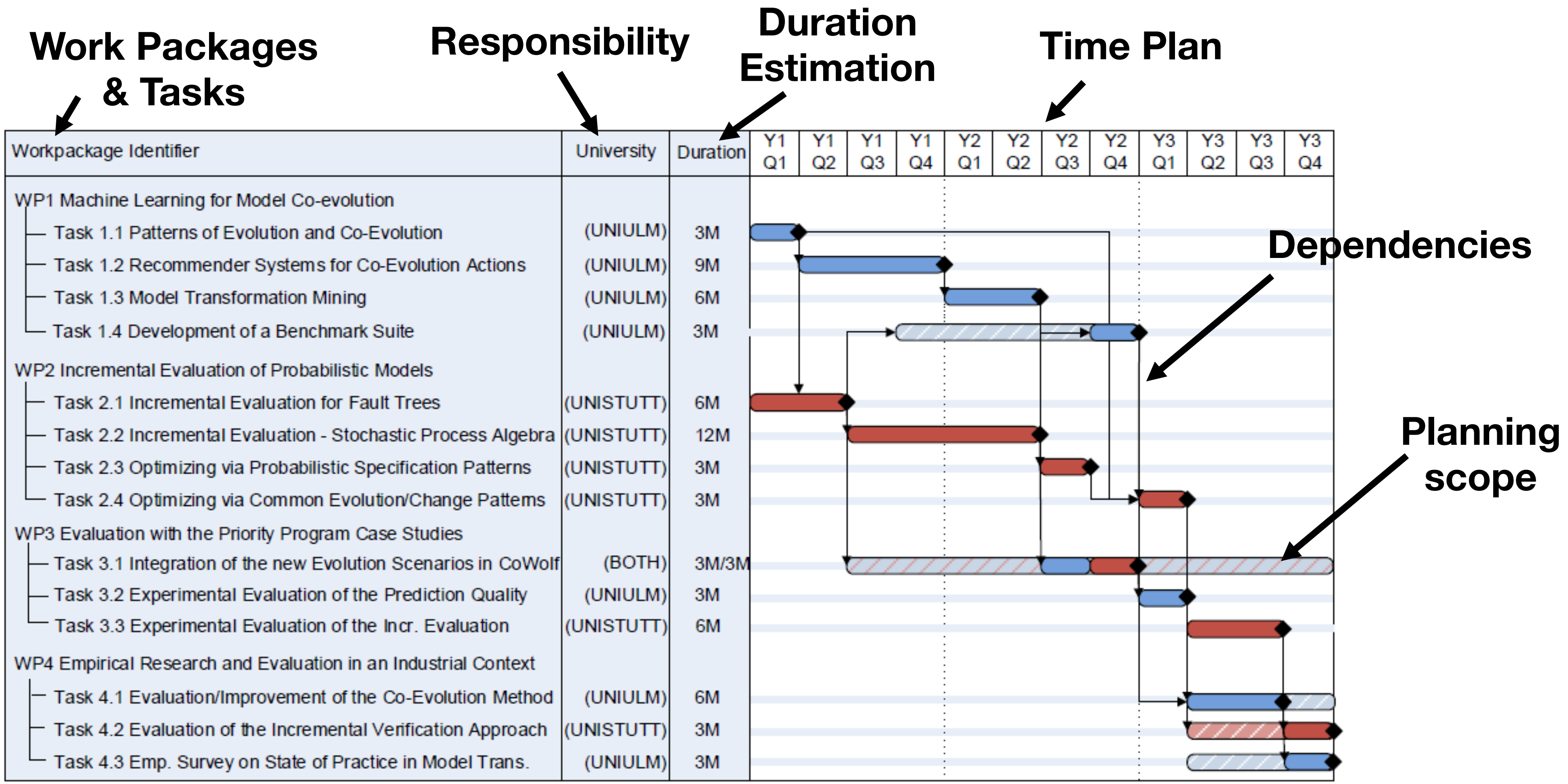
Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. "Design Patterns. Elements of Reusable Object-Oriented Software." Addison-Wesley Publishing Company (1995).

Topics

- ❑ Requirements Elicitation – Analysis Techniques (Questions, Interviews, …)
- ❑ Requirements Modeling – GORE, Use Case, Activity Diagram, etc.
- ❑ Architecture: Architectural Drivers, Structures (Static + Dynamic), Architectural Styles
- ❑ Module Design: Design Pattern, Design Principles
- ❑ **Project Planning: Work Packages, Gantt-Charts, PERT, Milestone Trend Analysis**
- ❑ Implementation: Clean Code
- ❑ Testing: Foundations + Equivalence Class Partitioning, JUnit, TDD, Testable Code, Clean Tests, Code/Test Coverage, Static Analyzers (Checkstyle, Spotbugs, PMD)
- ❑ Debugging: TRAFFIC, Reducing the input (delta debugging), Reducing the program (slicing), SBFL/SFL, Interactive Debugging
- ❑ Software Integration Strategies and Integration Testing

Gantt-Charts (Example)

Aspects of **project planning** are **task** planning, **time** planning, and **resource** planning. The **Gantt-Chart** helps you to list your identified **work packages** and show their **dependencies** and the **time plan** with your project **milestones**.

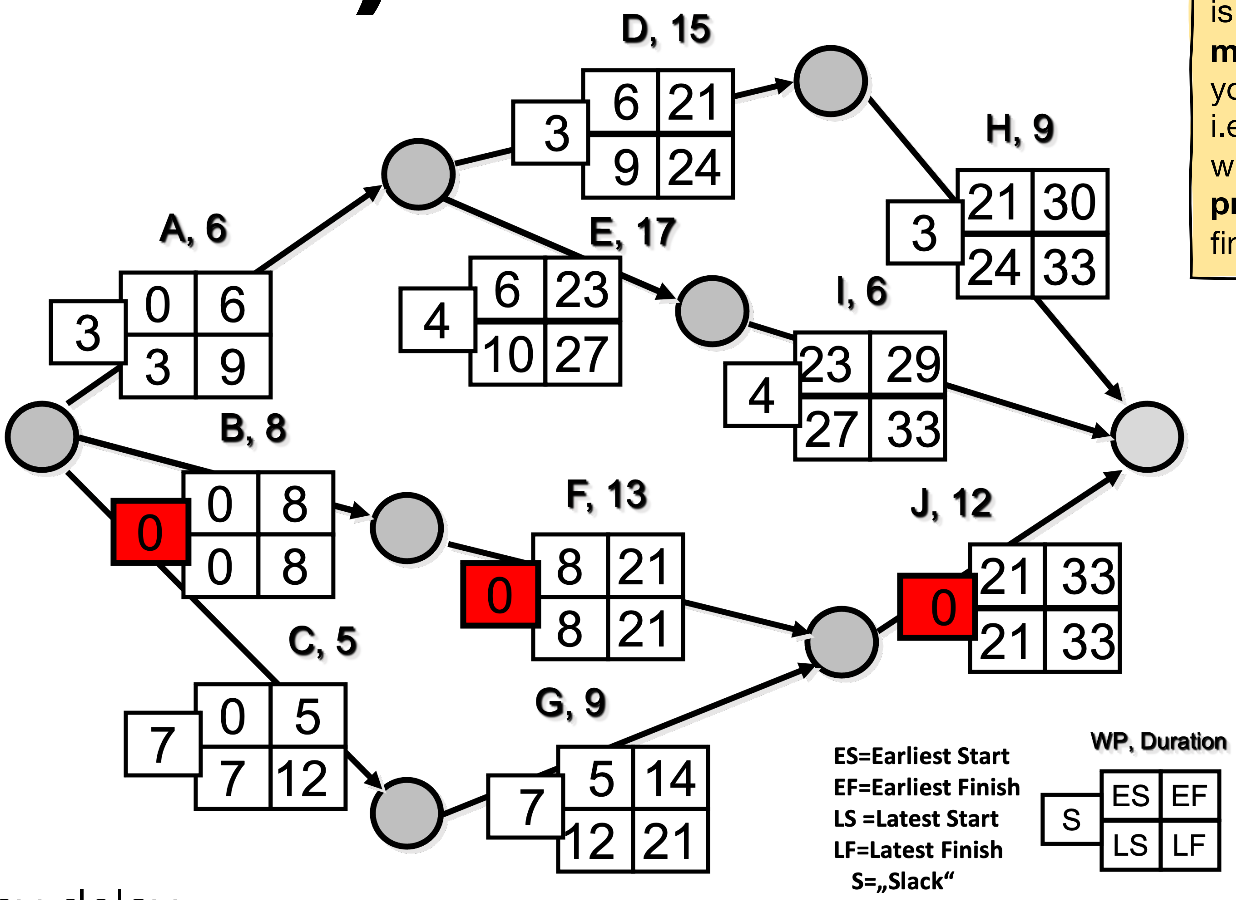


Workpackage Identifier	University	Duration
WP1 Machine Learning for Model Co-evolution		
Task 1.1 Patterns of Evolution and Co-Evolution	(UNIULM)	3M
Task 1.2 Recommender Systems for Co-Evolution Actions	(UNIULM)	9M
Task 1.3 Model Transformation Mining	(UNIULM)	6M
Task 1.4 Development of a Benchmark Suite	(UNIULM)	3M
WP2 Incremental Evaluation of Probabilistic Models		
Task 2.1 Incremental Evaluation for Fault Trees	(UNISTUTT)	6M
Task 2.2 Incremental Evaluation - Stochastic Process Algebra	(UNISTUTT)	12M
Task 2.3 Optimizing via Probabilistic Specification Patterns	(UNISTUTT)	3M
Task 2.4 Optimizing via Common Evolution/Change Patterns	(UNISTUTT)	3M
WP3 Evaluation with the Priority Program Case Studies		
Task 3.1 Integration of the new Evolution Scenarios in CoWolf	(BOTH)	3M/3M
Task 3.2 Experimental Evaluation of the Prediction Quality	(UNIULM)	3M
Task 3.3 Experimental Evaluation of the Incr. Evaluation	(UNISTUTT)	6M
WP4 Empirical Research and Evaluation in an Industrial Context		
Task 4.1 Evaluation/Improvement of the Co-Evolution Method	(UNIULM)	6M
Task 4.2 Evaluation of the Incremental Verification Approach	(UNISTUTT)	3M
Task 4.3 Emp. Survey on State of Practice in Model Trans.	(UNIULM)	3M

(Example taken from a Research Project)

Program Evaluation and Review Technique (PERT)

Work Package (WP)	Duration (e.g., days)	Depends on
A	6	–
B	8	–
C	5	–
D	15	A
E	17	A
F	13	B
G	9	C
H	9	D
I	6	E
J	12	F, G



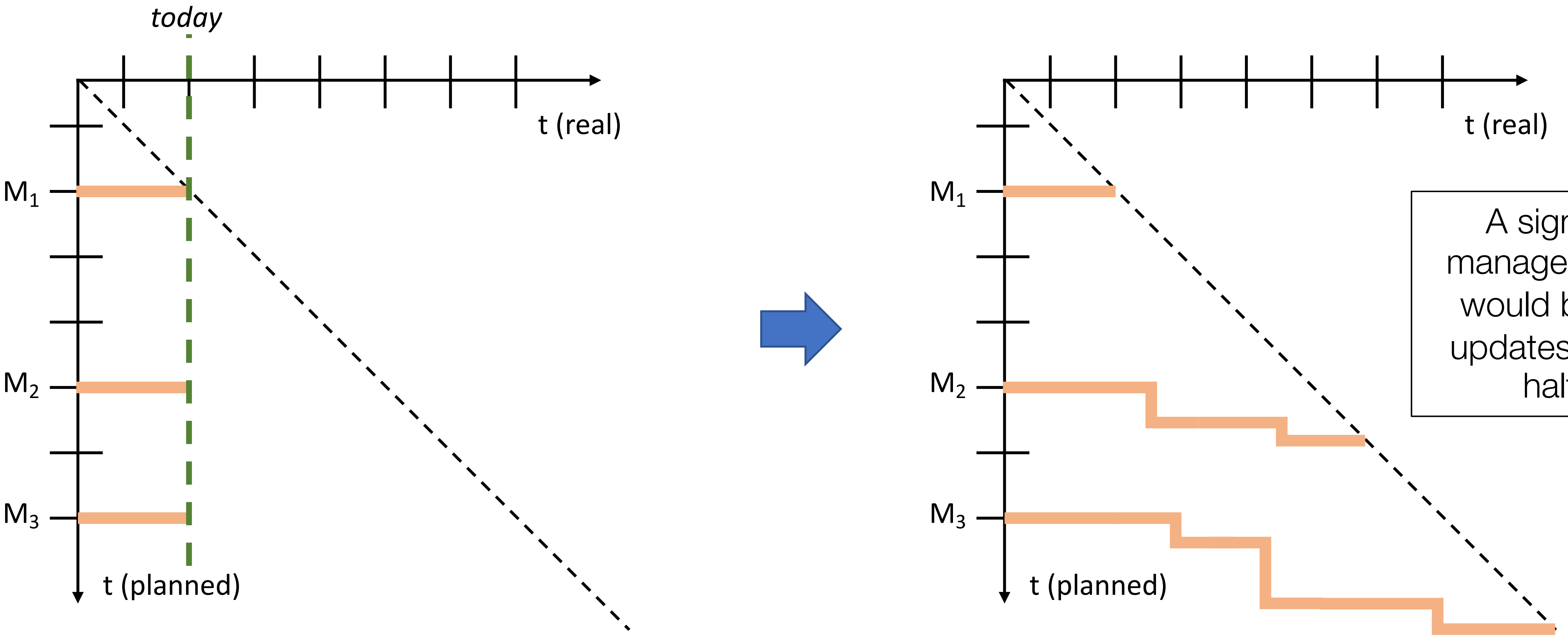
One crucial step in planning is to **identify risks** and **mitigate** them. **PERT** helps you identify the **critical path**, i.e., the work packages that will **delay** the **complete project** if they are not finished on time.

→ Identify the ***critical path***, i.e., any delay along this path will delay the complete project

Planning & Retrospective

To **improve planning**, it is important to **keep track of decisions** and eventually perform a proper **retrospective**. The **Milestone Trend Analysis** supports you for that.

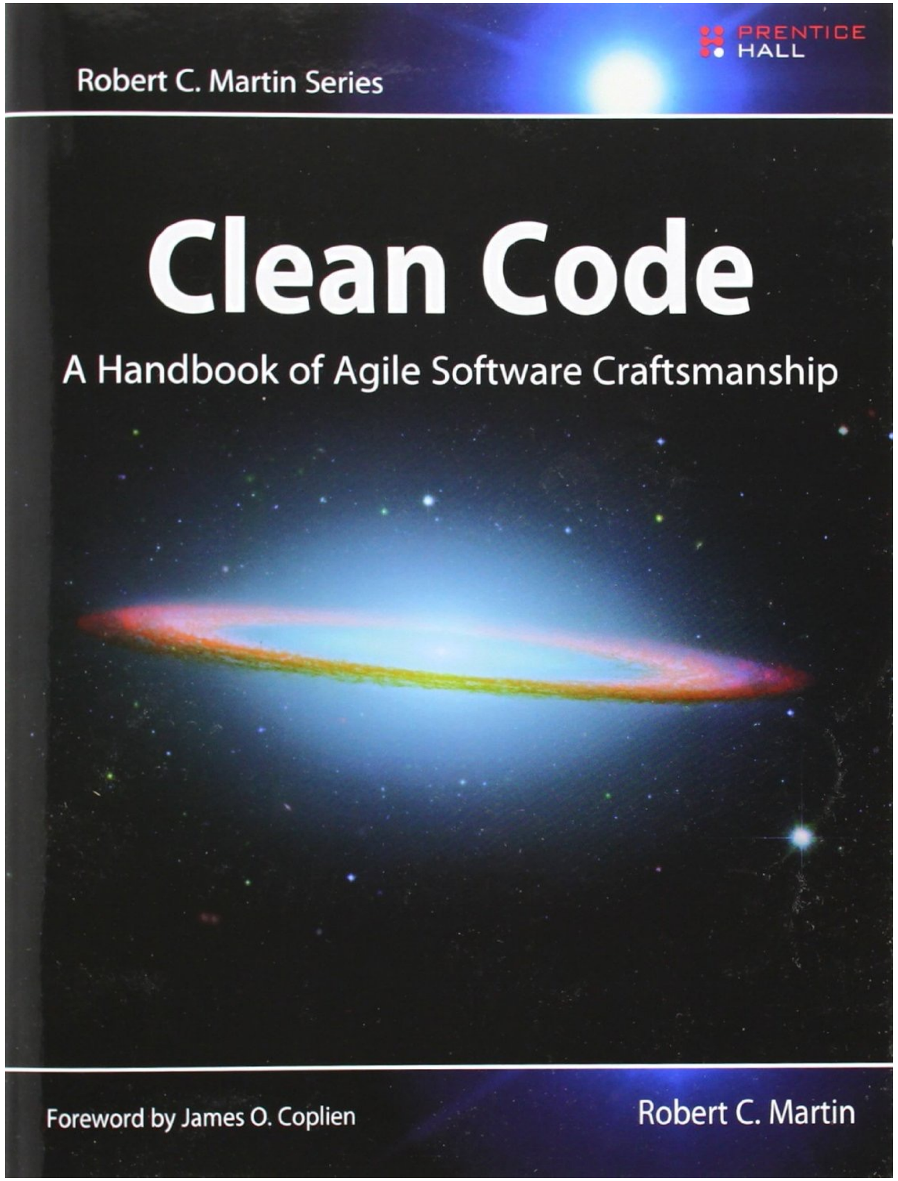
→ **Milestone Trend Analysis (MTA)**, continuous task in project planning



A sign of bad management skills would be a lot of updates along the half-line.

Topics

- ❑ Requirements Elicitation – Analysis Techniques (Questions, Interviews, …)
- ❑ Requirements Modeling – GORE, Use Case, Activity Diagram, etc.
- ❑ Architecture: Architectural Drivers, Structures (Static + Dynamic), Architectural Styles
- ❑ Module Design: Design Pattern, Design Principles
- ❑ Project Planning: Work Packages, Gantt-Charts, PERT, Milestone Trend Analysis
- ❑ **Implementation: Clean Code**
- ❑ Testing: Foundations + Equivalence Class Partitioning, JUnit, TDD, Testable Code, Clean Tests, Code/Test Coverage, Static Analyzers (Checkstyle, Spotbugs, PMD)
- ❑ Debugging: TRAFFIC, Reducing the input (delta debugging), Reducing the program (slicing), SBFL/SFL, Interactive Debugging
- ❑ Software Integration Strategies and Integration Testing



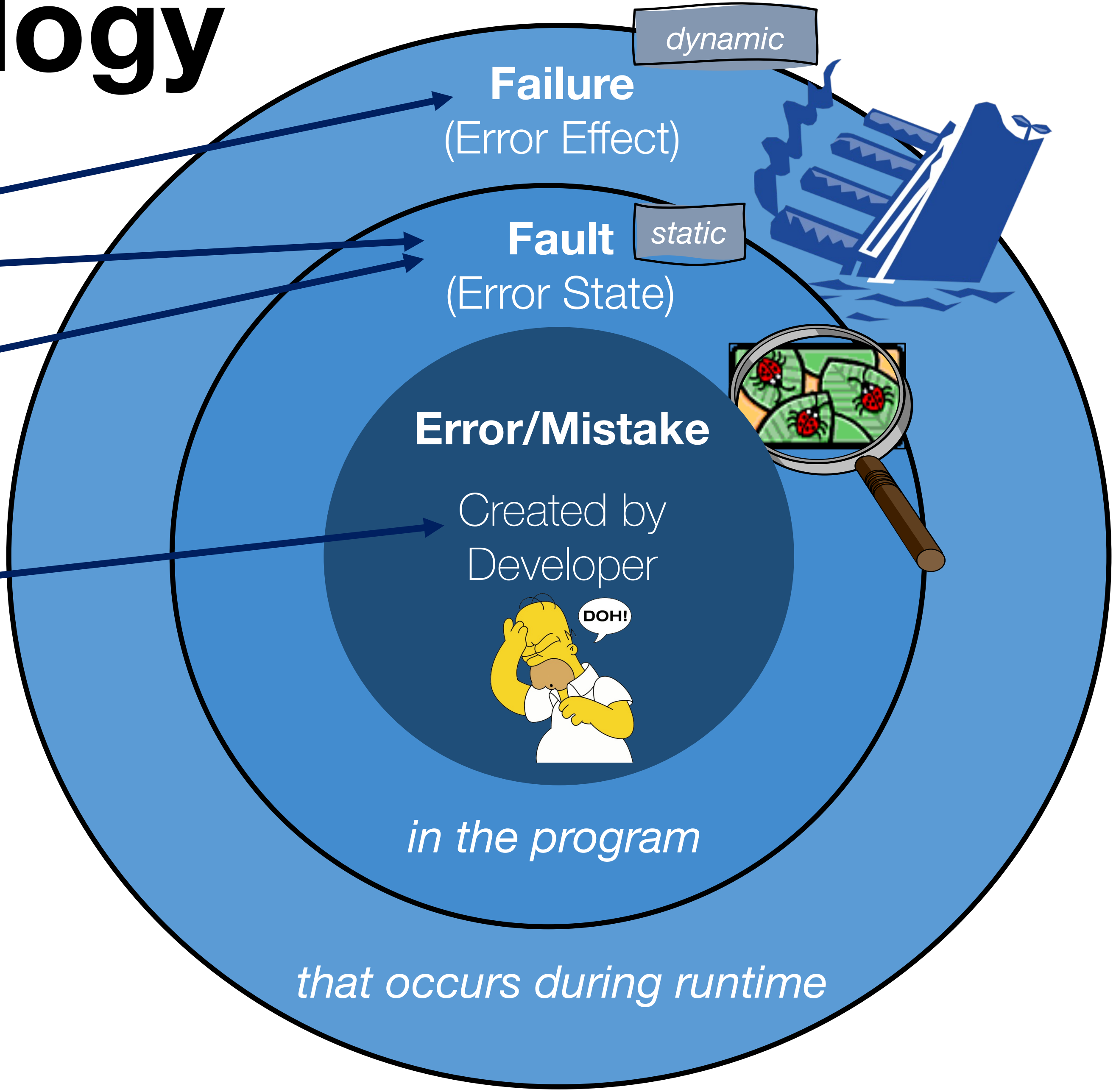
Robert C. “Uncle Bob” Martin:
Clean Code: A Handbook of Agile Software Craftsmanship
Prentice Hall, 2008

Every software developer should be aware of **clean code**. Therefore, we covered some relevant aspects: meaningful names/identifiers, clean functions and comments, code formatting, clean objects and data structures, clean error handling and clean unit tests.

Topics

- ❑ Requirements Elicitation – Analysis Techniques (Questions, Interviews, …)
- ❑ Requirements Modeling – GORE, Use Case, Activity Diagram, etc.
- ❑ Architecture: Architectural Drivers, Structures (Static + Dynamic), Architectural Styles
- ❑ Module Design: Design Pattern, Design Principles
- ❑ Project Planning: Work Packages, Gantt-Charts, PERT, Milestone Trend Analysis
- ❑ Implementation: Clean Code
- ❑ **Testing: Foundations + Equivalence Class Partitioning, JUnit, TDD, Testable Code, Clean Tests, Code/Test Coverage, Static Analyzers (Checkstyle, Spotbugs, PMD)**
- ❑ Debugging: TRAFFIC, Reducing the input (delta debugging), Reducing the program (slicing), SBFL/SFL, Interactive Debugging
- ❑ Software Integration Strategies and Integration Testing

Testing Terminology



The **chain of error**:
error → ***fault*** *(error state)* → ***failure*** *(error effect)*

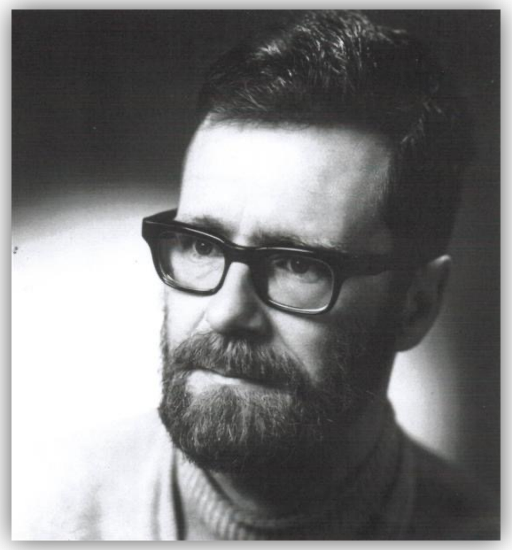
With testing we trying to find failures and subsequently we want to debug them by eliminating the fault.

Foundations of Testing

- Some fundamentals have been established over the last 50 years.

  »Program testing can be used to show the presence of bugs, but never to show their absence!«

  Edsger W. Dijkstra, 1970



  »Complete testing is not possible«
  »Start as early as possible with testing«

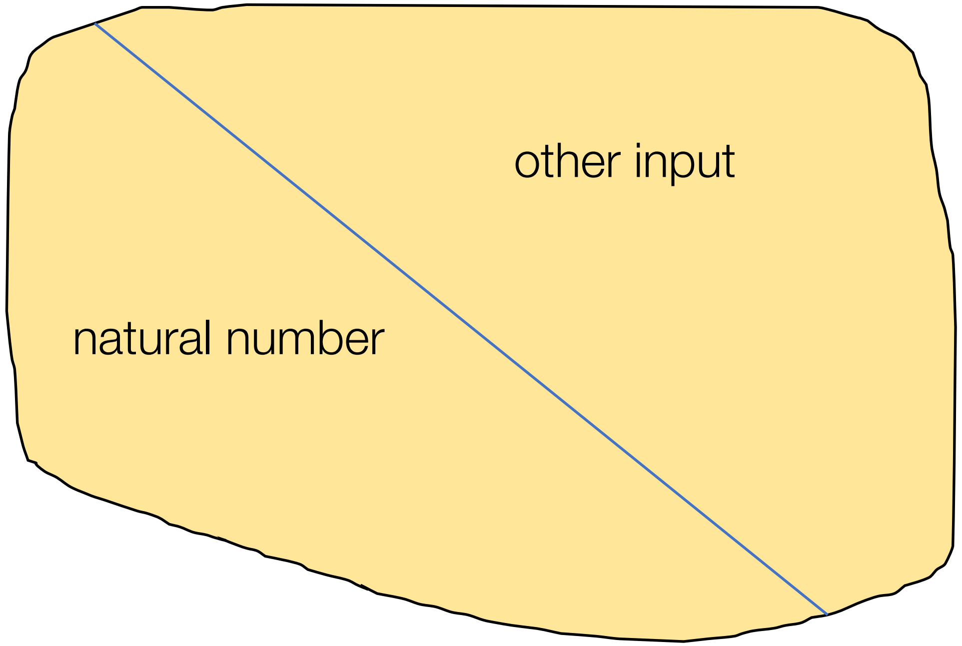
- Testing is not a late phase of the development process, but should be included as early as possible. The sooner errors are found, the lower the costs.

The **foundations** of testing have been established over the last 50 years, and **still**, it is a **problem in practice.** The research is working on automated testing techniques.

Equivalence Class Partitioning
(Example 2/4)

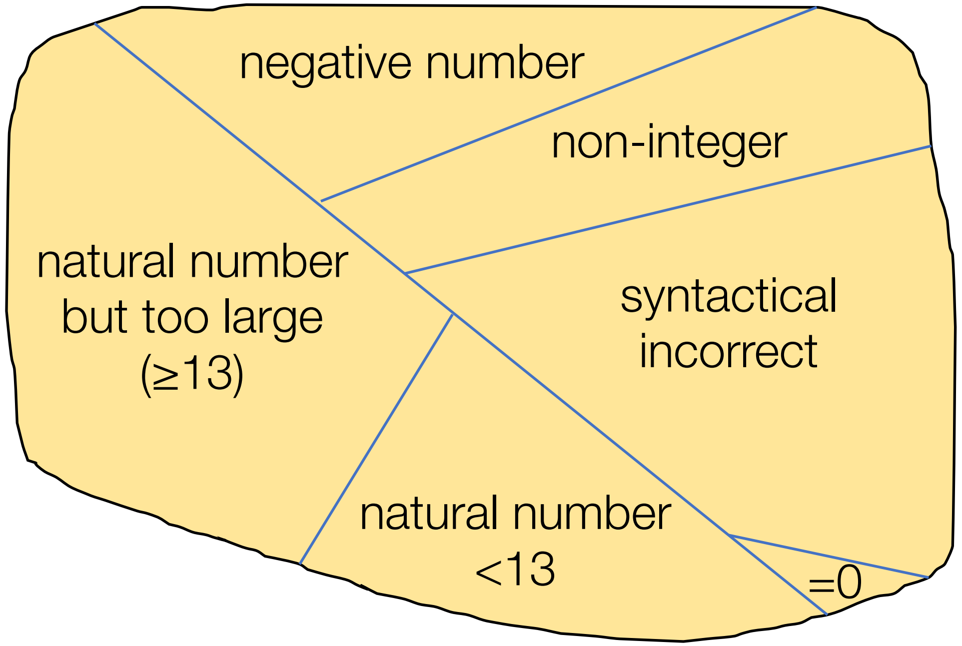
As a concrete way of **identifying test cases**, we look into the **equivalence class partitioning** method. Step by step, the partitions can be **refined** and improved. Finally, one chooses **representative values** from each class to **construct** test cases.

- A program to calculate the **factorial** of *n*.
- A program that is to calculate the factorial of *n* must reject (1) negative numbers, (2) real fractions, (3) numbers whose factorial is too large ($n \geq 13$), and (4) syntactically incorrect inputs. Special case: 0!



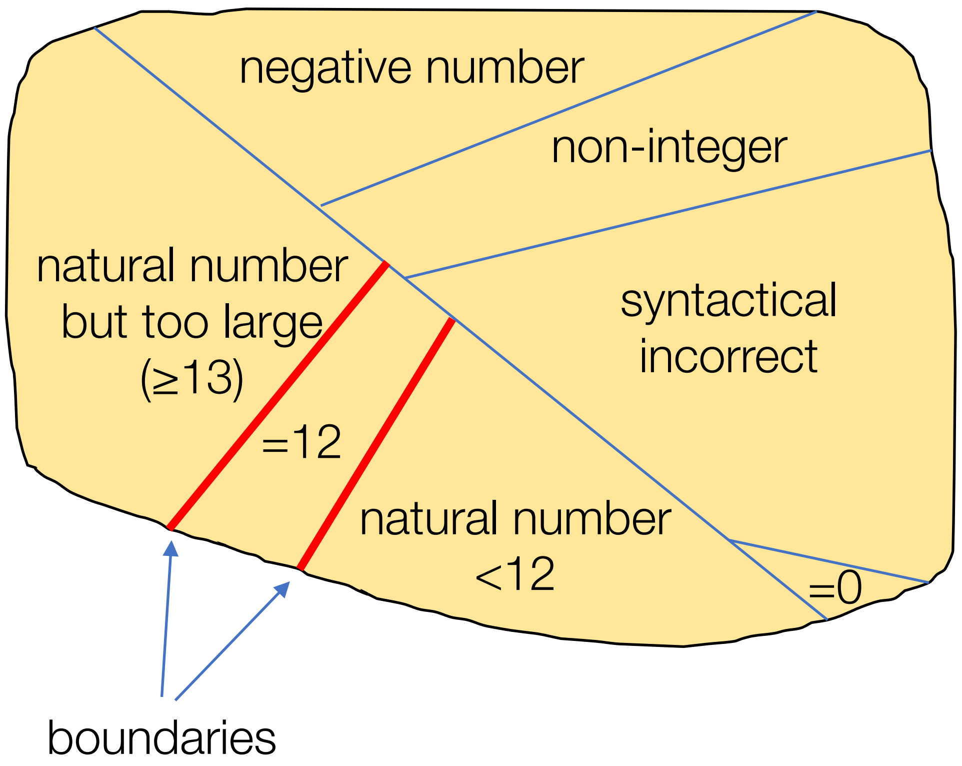
Refinement

results in 6 classes



Boundary Value Analysis
(Example 5/5)

The **boundary value analysis** explores the boundaries between the classes because these corner cases are a **typical location for programming mistakes.**



Class	Input	Expected Outcome
Negative number	-5	Error message
Non-integer	3.14	Error message
Too large number	100	Error message
Syntactical Incorrect input	“ABC”	Error message
Normal/expected input	7	5040
Zero	0	1
Boundary Value	12	479001600
Boundary Value -1	11	39916800
Boundary Value +1	13	Error message

Unit Testing

Unit testing probes components in **isolation.**

"In this test, individual, manageable program program units are tested, depending on the programming language, e.g., functions, subroutines or classes."

Ludewig/Lichter, 2007

- ❑ Each component is tested individually, in isolation.
- ❑ Implemented software units are tested systematically.
- ❑ Error conditions can be clearly traced back to the source.
- ❑ Components can be interconnected, this is not considered in unit testing and only the component in itself is tested.
- ❑ Unit tests are based on the component specification, the code and all related documents.

Unit Testing in Java with JUnit

- JUnit is a unit testing framework for Java.
- It is now considered the standard for unit testing in Java.
- Originally developed by Kent Beck and Erich Gamma.
- Current version JUnit 5: http://junit.org/

Unit testing of Java programs is supported by the **JUnit framework**. It usually is highly supported by IDEs, which makes it easy to integrate unit testing in the development process.

Motto: „Keep the bar green to keep the code clean!“

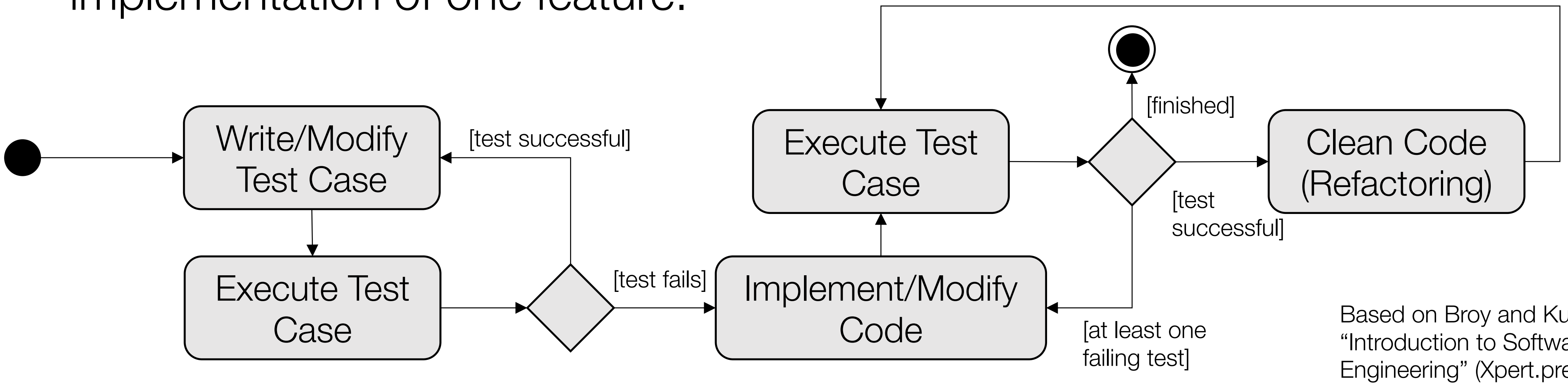
Visualization by means of colored bar: if the test finds no errors, the bar turns **green**; a **red** bar indicates errors.

Examples and project-related workflow will be shown in the Lab!

Test Driven Development (TDD)

- Refers to a **style** of software development that focuses on testing.
- The three core tasks of **coding**, **testing** and **design** are carried out in an interactive manner.
- The procedure described below maps the simple rules of Test-Driven Development in an incremental/iterative process for the implementation of one feature.

TDD is a common practice for merging coding and testing. The workflow on this slide shows the basic steps to follow TDD for the **incremental** implementation of one feature.



Based on Broy and Kuhrmann. "Introduction to Software Engineering" (Xpert.press), 2021.

Three Laws of TDD (by Kent Beck)

Rule 1:
You may not write production code until you have written a failing unit test.

Rule 2:
You may not write more of a unit test than is sufficient to fail, and not compiling is failing.

Rule 3:
You may not write more production code than is sufficient to pass the currently failing test.

TDD comes with simple **rules** that should be followed. They set the boundaries for **what should be done** and **what should *not* be done.**

K. Beck. “Test Driven Development: By Example.” Addison-Wesley Longman, 2002.

Testing – Best Practices (1/2)

A typical problem in practice are **flaky tests** which can be caused by dependent test cases.

- Test cases should be **independent**!
  - The JUnit execution model executes test cases in arbitrary order (unless explicitly defined).
  - Use @Before.. Annotations to define test case preparations! Do not assume that another test case already created some sort of test data or program state.
  - Dependent test cases can cause **flaky tests**: sometimes they pass, sometimes they fail, depending on the test execution order. General reasons for flaky tests:
    - an issue with the test itself
    - some external factor compromising the test results
    - an issue with the newly-written code

August Shi, Wing Lam, Reed Oei, Tao Xie, and Darko Marinov. *"IFixFlakies: a framework for automatically **fixing order-dependent flaky tests**".* In Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), 2019.

Wing Lam, Reed Oei, August Shi, Darko Marinov and Tao Xie, "*iDFlakies: A Framework for **Detecting** and Partially **Classifying Flaky Tests***".12th IEEE Conference on Software Testing, Validation and Verification (ICST), 2019.

Testing – Best Practices (2/2)

As for production code, **test code** should follow established **best practices.** Most of them are concerned about **readability.**

- ❑ One test case for one feature (→ Single Responsibility for Tests). Keep things simple!
- ❑ 5LOC Rule: Strive to write test cases 5LOC long.
- ❑ Choose meaningful test method names!
- ❑ Use same package structure as for source code.
  → Test code is separate, but you can access methods with package accessibility
- ❑ Test cases should have the end user or defined requirements in mind.
- ❑ Peer review is important!

(un)Testable Code (1/3)

Based on "Guide: Writing Testable Code" written by Google developers
http://misko.hevery.com/code-reviewers-guide/

Flaw #1 – Constructor does Real Work

"When your constructor has to instantiate and initialize its collaborators, the result tends to be an inflexible and prematurely ***coupled design****. Such constructors shut off the ability to inject test collaborators when testing."*

- ❑ violates the **S**ingle Responsibility Principle
- ❑ testing directly is **difficult**

Another aspect is that the code itself should support proper **testability**. We discussed several scenarios and flaws in code and how to improve it to enable better testing.

Topics

- ❑ Requirements Elicitation – Analysis Techniques (Questions, Interviews, …)
- ❑ Requirements Modeling – GORE, Use Case, Activity Diagram, etc.
- ❑ Architecture: Architectural Drivers, Structures (Static + Dynamic), Architectural Styles
- ❑ Module Design: Design Pattern, Design Principles
- ❑ Project Planning: Work Packages, Gantt-Charts, PERT, Milestone Trend Analysis
- ❑ Implementation: Clean Code
- ❑ Testing: Foundations + Equivalence Class Partitioning, JUnit, TDD, Testable Code, Clean Tests, Code/Test Coverage, Static Analyzers (Checkstyle, Spotbugs, PMD)
- ❑ **Debugging: TRAFFIC, Reducing the input (delta debugging), Reducing the program (slicing), SBFL/SFL, Interactive Debugging**
- ❑ Software Integration Strategies and Integration Testing

Debugging – From Error to Failures

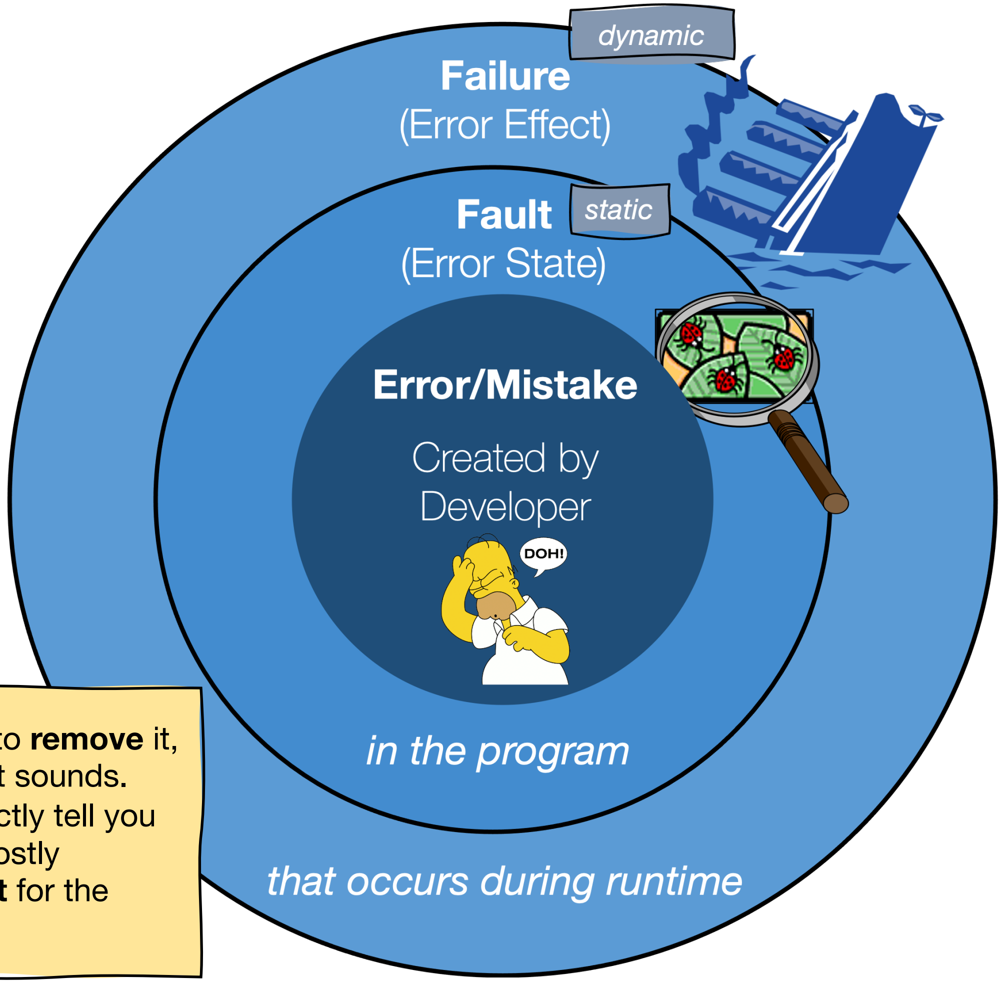
The issue of *debugging* is to

- *relate* an observed failure to a fault/defect and
- to *remove* the defect such that the failure no longer occurs.

Debugging Steps:

- Execution of tests!
- ***Fault Localization!***
- Identify possible fixes.
- Choose the best fix.
- Implement the best fix!

After finding a bug, we also want to **remove** it, which may be more difficult than it sounds. Knowing the **failure** does not directly tell you the **actual fault**. **Debugging** is mostly concerned about **finding the fault** for the observed failure.



Debugging – Difficulties

Debugging can be **frustrating** but also has its **rewards** when the bug is finally eliminated.

- Symptom and failure cause can be far apart.
- Symptoms of one error may be hidden by other errors.
  - Fault masking: “An occurrence in which one defect prevents the detection of another [IEEE 610]
- Symptoms of one error may disappear or change due to correction of another error.

„Debugging is one of the more frustrating parts of programming. It has elements of brain teasers, coupled with the annoying recognition that you have made a mistake.“

B. Shneiderman: Software Psychology. Winthorp Publishers, 1980.

TRAFFIC Principle

Debugging can be **time consuming** and should be done **systematically. TRAFFIC** presents such a systematic way on a high level.

Debugging should follow the TRAFFIC principle:

- ❑ **T**rack the problem
- ❑ **R**eproduce – Requires control over data and environment.
- ❑ **A**utomate – Write a simple test case that exercises the problem.
- ❑ **F**ind Origins – Where does the failure originate? Locate likely fault locations.
- ❑ **F**ocus – Focus your effort on the most likely origin.
- ❑ **I**solate – Isolate the fault (see scientific method of debugging, next slide).
- ❑ **C**orrect – Fix the fault and verify that the failure no longer occurs.
  Check for regression errors.

Debugging – Techniques

In the lecture and in the lab, we discussed several concrete **debugging techniques**, which are shown here on the slide. They are concerned on either **reducing** the **input** or with **reducing** the **program**.

- **Reduce the input:** ***Delta Debugging***
  Simplifying and Isolating Failure-Inducing Input
- **Reduce the program:** ***Program Slicing***
  Isolating the relevant program statements/locations to focus debugging effort.
  - Dynamic Slicing
  - Static Forward and Backward Slicing
  - Relevant Slicing
- **Identify faulty statements:** ***Statistical Fault Localization***
  Ranking suspicious program statements.

Dynamic Slicing

- Slice backward from the erroneous output of the program
- Dynamic slice includes the closure of
  - **Data dependencies** and
  - **Control dependencies**

For our test case with **a=2**, the value of variable **x** printed in line 6 is **unexpected**.

```
1   b=2;
2   y=1;
3   If (a>1){
4       if (b>1){
5           x=2;
        }
    }
6   printf ("%d", x);
```

Control Dependence

Data Dependence

Slicing Criterion

Dynamic slicing tries to **shrink** the program to the set of **instructions** that **influence** a specific **value** in the program for a specific **test input.**

Dependency Graphs

Example based on lecture by
Michael Pradel (University of Stuttgart):
https://www.youtube.com/watch?v=flkYsAkc8rA

Input: 1

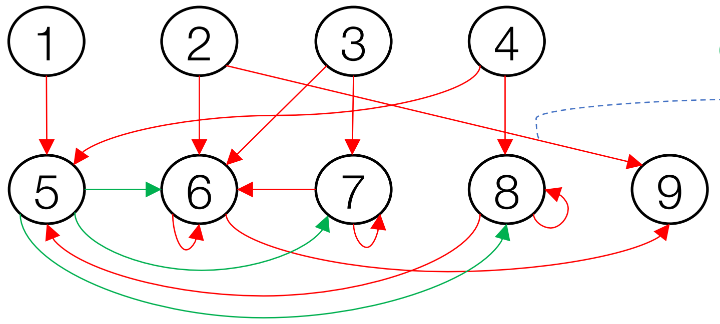
```
1  int x = read();
2  int z = 0;
3  int y = 0;
4  int i = 1;
5  while (i <= x) {
6     z = z + y;
7     y = y + 1;
8     i = i + 1;
   }
9  printf("%d", z);
```

Slicing Criterion

The **resulting dynamic slice depends** on the type of used **dependency graph.**

How do we calculate the data dependencies?

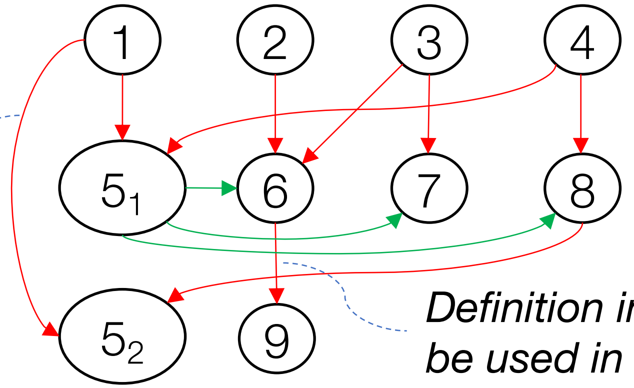
- Static dependency graph



Data dependencies
Control dependencies

Definition in 2 may be used in 9.

- Dynamic dependency graph



Definition in 1 will be used in 5_2.

Definition in 6 will be used in 9.

Static Slicing (2/2)

```
read(max);
int a = 1;
int b = 1;
int i = 1;
int fak = 1;
do {
    if (i > 2) {
        int tmp = a;
        b = b + a;
        a = tmp;
    }
    fak = fak * i;
    i++;
} while (i <= max);

print(n);
print(b);
print(fak);
```

Forward Slice

What is affected by this assignment?

read(max);
int a = 1;
~~int b = 1;~~
~~int i = 1;~~
~~int fak = 1;~~
do {
~~if (i > 2) {~~
int tmp = a;
b = b + a;
a = tmp;
}
~~fak = fak * i;~~
~~i++;~~
~~} while (i <= max);~~

~~print(n);~~
print(b);
~~print(fak);~~

Backward Slice

What influenced the value of this variable?

read(max);
~~int a = 1;~~
~~int b = 1;~~
int i = 1;
int fak = 1;
do {
~~if (i > 2) {~~
~~int tmp = a;~~
~~b = b + a;~~
~~a = tmp;~~
}
fak = fak * i;
i++;
} while (i <= max);

~~print(n);~~
~~print(b);~~
print(fak);

Static slicing is not concerned with any test input and provides slices in two ways: **forward slice** (what is affected) and **backward slice** (what influenced the value).

Potential Dependence (2/2)

Relevant Slice

input: a=2

```
1   b=1;
2   x=1;
3   If (a>1){
4       if (b>1){
5           x=2;
        }
    }
6   printf (“%d”, x);
```

Potential Dependence

Dynamic Data Dependence

Another interesting aspect is the **relevant slice**, which also takes into account the **potential dependencies** in the program.

Visualizing Fault Localization (2/3)

Test Cases

	3,3,5	1,2,3	3,2,1	5,5,5	5,3,4	2,1,3
mid(){ int x,y,z,m;						
read("Enter 3 numbers:", x,y,z);	●	●	●	●	●	●
m = z;	●	●	●	●	●	●
if(y < z)	●	●	●	●	●	●
if(x < y)	●	●			●	●
m = y;		●				
else if (x< z)	●				●	●
m = y; // bug	●					●
else			●	●		
if (x > y)			●	●		
m = y;			●			
else if (x > z)				●		
m = x;						
print("Middle number is:", m);	●	●	●	●	●	●
}						
Pass Status	P	P	P	P	P	F

Statistical Fault Localization assigns **suspiciousness scores** to statements (or another spectrum level) based on their occurrence in failing and passing test cases.
The results can be **highlighted** in the code to **guide** the developer to potentially faulty statements.

Tool Support for Software Quality Assurance

To complete the unit testing discussion, we explored several **static analyzers** and their integration into IDEs.

https://checkstyle.sourceforge.io
https://plugins.jetbrains.com/plugin/1065-checkstyle-idea

https://spotbugs.github.io
https://plugins.jetbrains.com/plugin/14014-spotbugs
https://plugins.jetbrains.com/plugin/3847-findbugs-idea

https://pmd.github.io
https://plugins.jetbrains.com/plugin/1137-pmdplugin

Topics

- ❑ Requirements Elicitation – Analysis Techniques (Questions, Interviews, …)
- ❑ Requirements Modeling – GORE, Use Case, Activity Diagram, etc.
- ❑ Architecture: Architectural Drivers, Structures (Static + Dynamic), Architectural Styles
- ❑ Module Design: Design Pattern, Design Principles
- ❑ Project Planning: Work Packages, Gantt-Charts, PERT, Milestone Trend Analysis
- ❑ Implementation: Clean Code
- ❑ Testing: Foundations + Equivalence Class Partitioning, JUnit, TDD, Testable Code, Clean Tests, Code/Test Coverage, Static Analyzers (Checkstyle, Spotbugs, PMD)
- ❑ Debugging: TRAFFIC, Reducing the input (delta debugging), Reducing the program (slicing), SBFL/SFL, Interactive Debugging
- ❑ **Software Integration Strategies and Integration Testing**

Integration Testing

Integration is concerned with **combining** several components to the **overall system**. During this system, it can come to various issues, mostly about **consistency problems** between the **components**.

Testing in which software components, hardware components, or both are combined and tested to **evaluate the interaction** between them.

[IEEE Std 610.12 (1990)]

- ❑ Integration tests serve for the (syntactical and semantic) **evaluation of the interfaces**.
- ❑ It is less concerned with the errors of the individual components (unit testing) but with **consistency problems** between the components.
- ❑ When everything is integrated, the system test can follow.

Overview: Integration Strategies

	Core Idea	Pro	Con
Top-Down	Start point: Component that only depends on others, but has no incoming dependency. Other components are replaced by placeholders.	Little or no drivers needed as high level components are used as test environment.	▪ Can be expensive ▪ Low level components must be replaced with stubs.
Bottom-Up	Start point: component that is not called. Larger sub-systems are created step by step.	No need for stubs.	Needs test drivers for high-level components.
Ad-Hoc	Start point: components are integrated as soon as they are ready.	No waiting times.	Needs both, stubs and drivers.
Big Bang	Everything is put together at once.		▪ All errors at once ▪ Difficult fault loalization ▪ Time until integration is wasted

We discussed several different **integration strategies**. We also conducted several exercises for planning integration testing by creating **integration tables** and **dependency graphs.**

Summary: Testing Strategies

You can find a summary of the major testing strategies occurring in software development.

Criteria	Unit Test	Integration Test	System Test	Acceptance Test
Testing Goal	Identify faults in software components that are tested in isolation.	Identify faults in interfaces and in the interaction between integrated components.	Checking whether the specified requirements (functional, non-functional) are met by the product.	Gain confidence in the system or in certain non-functional properties.
Testing Base	→ Component specification → Detailed design → Data model → Source code	→ Software architecture → Workflows → Use cases	→ Requirements specification → Use cases → Functional requirements → Business processes → Risk analysis report	→ User requirements → System requirements → Use cases → Business processes → Risk analysis report
Typical Test Subjects	Isolated source unit (class, package, module) → Components, programs → Data transformation or migration programs → Database modules	To be integrated individual components, sub systems or purchased standard software/components/libraries → Database implementation → Infrastructure → Interfaces → System configuration and configuration data	→ System and user documentation/handbooks → System configuration and configuration data	→ Business processess of the integrated system → Production and maintenance processes → User procedures → Forms → Reports → Configuration data
Testing Tools	IDE, Interactive Debugger, Static Analyzer, Unit Testing Frameworks like JUnit	Test monitoring to observe interaction (data exchange) of components	Test management tools, automated UI testing	
Testing Environment	Stubs, drivers, simulators	Reuse/extension of placeholders/stubs, drivers and simulators generated for unit testing.	Testing and production environment should be mostly the same.	Testing and production environment should be mostly the same.

Source: German Testing Board

Aspects of Version Control

Finally, we add here some additional material for your own study about version control systems. We will discuss some aspects in the lab.

Version Control (Overview)

Having **version control systems** is standard nowadays. They support the developers and take care of the source code **versioning** and **backup**.

- Reasons for version control of software:
  - Parallel editing of software by multiple persons
  - Change tracking
  - Undo of changes
  - Data backup of the source code
- Basic abilities of version control systems
  1. Version and release identifier
  2. Tracking of change history
  3. Independent, parallel development + merging
  4. Merge-Conflict handling
  5. (Efficient) memory management (deltas)
- Usage model:
  - check out → edit → check in

Configuration, Baseline, Release

- **Configuration**:
  Set of software units that together form a functioning (sub)system.

- **Baseline**:
  A stable configuration as reference point for further development.

- **Release**:
  Baseline that is delivered to customer.

There are some basic terms, which should be known: **Configuration**, **baseline** and **release.**

Version Control Systems

Type	Description	Example
Local	Local archiving of (mostly single) files.	SCCS, RCS
Central	Revisions are located on central server. Clients request updates, send changes.	CVS, SVN, Perforce, Visual SourceSafe
Distributed	Distributed repositories (with all known revisions) that can be synchronized.	Git, Mercurial, ClearCase

Version control systems come in different **types.** Historically the **centralized** version control system was first, but also with the rise of platforms like GitHub and GitLab, the distributed repositories are becoming the current standard. For a more detailed comparison you can check: https://www.perforce.com/blog/vcs/git-vs-svn-what-difference

Git Commands

We also use **Git** for the **projects** in this course, and you probably have searched online for some of its commands. Knowing the basics like **`git pull`**, **`push`**, **`commit`**, and **`status`** is **essential**, but for the more sophisticated (sequence of) commands a **cheat sheet** becomes handy. As a **software engineer** it necessary to **know** your toolset; it should not only include programming (languages) but also other (DevOps) tools like Git.

Git Cheat Sheet

GIT BASICS

`git init <directory>`	Create empty Git repo in specified directory. Run with no arguments to initialize the current directory as a git repository.
`git clone <repo>`	Clone repo located at `<repo>` onto local machine. Original repo can be located on the local filesystem or on a remote machine via HTTP or SSH.
`git config user.name <name>`	Define author name to be used for all commits in current repo. Devs commonly use `--global` flag to set config options for current user.
`git add <directory>`	Stage all changes in `<directory>` for the next commit. Replace `<directory>` with a `<file>` to change a specific file.
`git commit -m "<message>"`	Commit the staged snapshot, but instead of launching a text editor, use `<message>` as the commit message.
`git status`	List which files are staged, unstaged, and untracked.
`git log`	Display the entire commit history using the default format. For customization see additional options.
`git diff`	Show unstaged changes between your index and working directory.

UNDOING CHANGES

`git revert <commit>`	Create new commit that undoes all of the changes made in `<commit>`, then apply it to the current branch.
`git reset <file>`	Remove `<file>` from the staging area, but leave the working directory unchanged. This unstages a file without overwriting any changes.
`git clean -n`	Shows which files would be removed from working directory. Use the `-f` flag in place of the `-n` flag to execute the clean.

REWRITING GIT HISTORY

`git commit --amend`	Replace the last commit with the staged changes and last commit combined. Use with nothing staged to edit the last commit's message.
`git rebase <base>`	Rebase the current branch onto `<base>`. `<base>` can be a commit ID, branch name, a tag, or a relative reference to HEAD.
`git reflog`	Show a log of changes to the local repository's HEAD. Add `--relative-date` flag to show date info or `--all` to show all refs.

GIT BRANCHES

`git branch`	List all of the branches in your repo. Add a `<branch>` argument to create a new branch with the name `<branch>`.
`git checkout -b <branch>`	Create and check out a new branch named `<branch>`. Drop the `-b` flag to checkout an existing branch.
`git merge <branch>`	Merge `<branch>` into the current branch.

REMOTE REPOSITORIES

`git remote add <name> <url>`	Create a new connection to a remote repo. After adding a remote, you can use `<name>` as a shortcut for `<url>` in other commands.
`git fetch <remote> <branch>`	Fetches a specific `<branch>`, from the repo. Leave off `<branch>` to fetch all remote refs.
`git pull <remote>`	Fetch the specified remote's copy of current branch and immediately merge it into the local copy.
`git push <remote> <branch>`	Push the branch to `<remote>`, along with necessary commits and objects. Creates named branch in the remote repo if it doesn't exist.

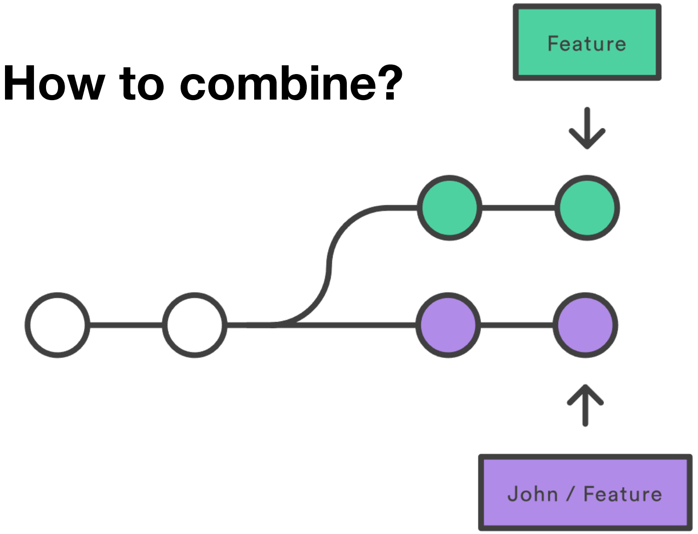
- ❑ knowing the git commands like `pull`, `push`, `commit`, `status` is essential
- ❑ you need to know your tools
- ❑ having a cheat sheet becomes handy

---

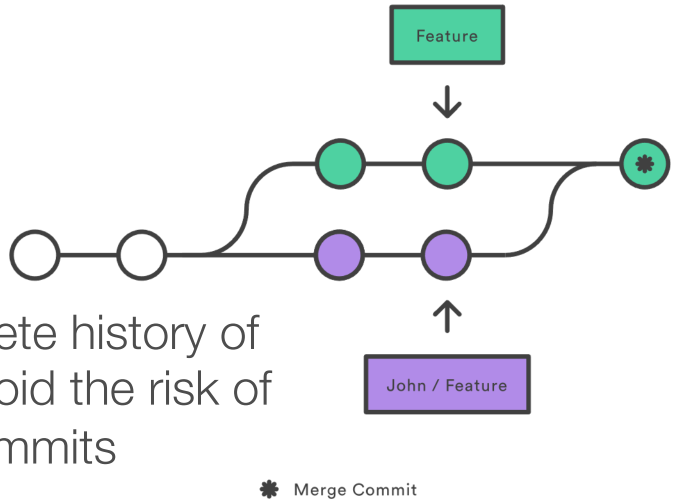
https://wac-cdn.atlassian.com/dam/jcr:e7e22f25-bba2-4ef1-a197-53f46b6df4a5/SWTM-2088_Atlassian-Git-Cheatsheet.pdf?cdnVersion=296

Merging vs. Rebasing

How to combine?



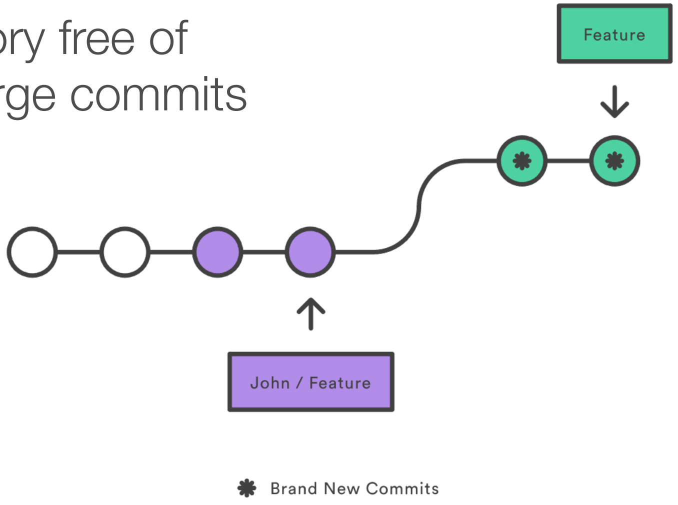
`git merge`



preserve the complete history of your project and avoid the risk of re-writing public commits

`git rebase`

clean, linear history free of unnecessary merge commits



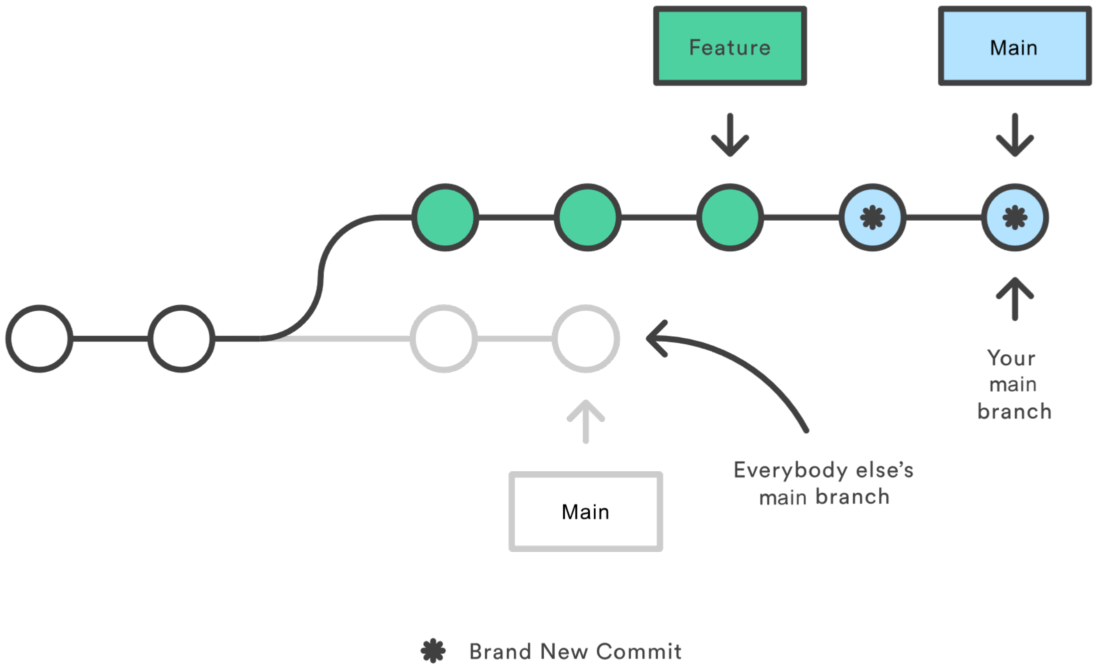
`git pull --rebase` can become handy to avoid unnecessary merge commits. The default for **`git pull`** is to use the `merge` concept.

Illustrations are taken from https://www.atlassian.com/git/tutorials/merging-vs-rebasing

The Golden Rule of Rebasing

However, **`rebase`** might not always be the best choice! Knowing the difference to merge is important! You need to adjust your practices to the organization rules.

“The golden rule of git rebase is to never use it on public branches.”



- ❑ Git will think that your main branch’s history has diverged from everybody else’s.
- ❑ Both main branches would need to be merged, resulting in an extra merge commit and two sets of commits that contain the same changes (the original ones, and the ones from your rebased branch).

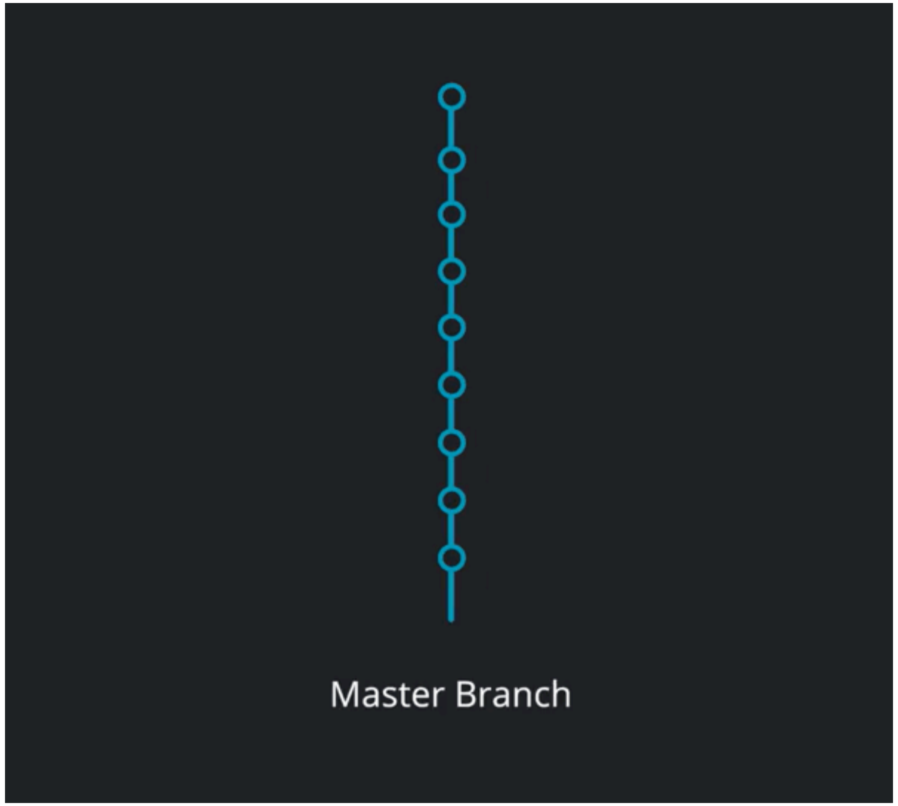
Better: First stash your changes, pull the latest main branch, apply your stash, and then push your changes.

Illustration taken from https://www.atlassian.com/git/tutorials/merging-vs-rebasing

git squash

Especially when you work on a **feature branch**, it may be useful to use `git squash` to **clean up** and simplify the commits history! Do not spoil the history with tiny commit sequences that actually belong together.

- ❑ rewrite your commit history
- ❑ this action helps to clean up and simplify your commit history before sharing your work with team members



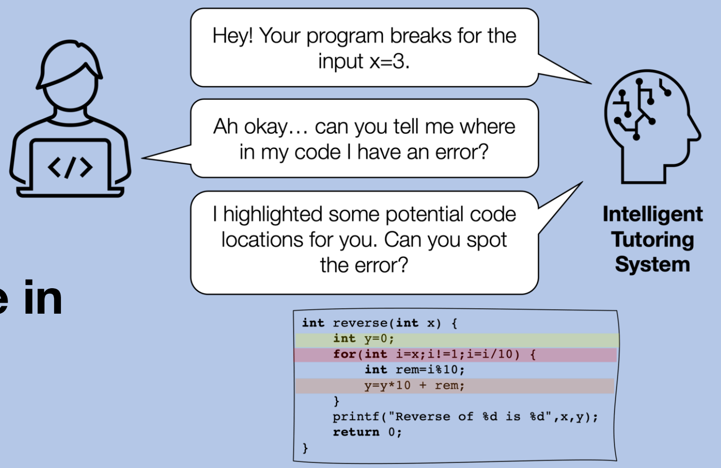
https://www.gitkraken.com/learn/git/git-squash

Conclusion

- ❑ Software Engineering is more than programming…
- ❑ **Good luck** with the final submissions and the final exam!

Next Week – Week 13:

- Wednesday → **Presentations**
- Thursday → **No labs anymore**



Please support our research and participate in our survey!

https://forms.office.com/r/DknsSTwVsP