



# SOFTWARE ENGINEERING

CO3001

## LECTURE 2&3 – REQUIREMENTS ENGINEERING

Anh Nguyen Duc  
Jingyue Li  
Daniela Soares Cruzes

# TOPICS COVERED

- ✓ Requirements engineering & Requirements
- ✓ Requirements elicitation
- ✓ Requirements specification
- ✓ Requirements validation
- ✓ Requirements management
- ✓ Goal oriented requirement engineering
- ✓ Quality of requirements
- ✓ Non-functional requirements
- ✓ Requirement traceability

# TOPICS COVERED

- ✓ Requirements engineering & Requirements
- ✓ Requirements elicitation
- ✓ Requirements specification
- ✓ Requirements validation
- ✓ Requirements management
- ✓ Goal oriented requirement engineering
- ✓ Quality of requirements
- ✓ Non-functional requirements
- ✓ Requirement traceability



# REQUIREMENTS

---

# EFFECTS OF INADEQUATE REQUIREMENTS DEVELOPMENT – AIRBUS

- ✓ **Requirement:** Reverse thrust may only be used when the airplane is landed.
- ✓ **Translation:** Reverse thrust may only be used while the wheels are rotating.
- ✓ **Implementation:** Reverse thrust may only be used while the wheels are rotating fast enough.

- **Situation:** Rainstorm – aquaplaning
- **Result:** Crash due to overshooting the runway!
- **Problem:** Erroneous in the requirement phase



# The Ariane 5 accident - 1

- Single root cause failure!
- **The "bug": attitude deviation**  
stored as 2-byte integer (max value 65,535) instead of 4-byte (max value 4,294,967,295)
- SW module was reused from Ariane 4
- Insufficient V&V of detailed requirements: larger attitude deviation tolerated in Ariane 5 than in Ariane 4
- Ariane 5 production cost 10 years and \$7 billion; luckily, no victims because it was unmanned.

$65,535 = 00000000 \text{ } 00000000 \text{ } 11111111 \text{ } 11111111$

$65,536 = 00000000 \text{ } 00000001 \text{ } 00000000 \text{ } 00000000$



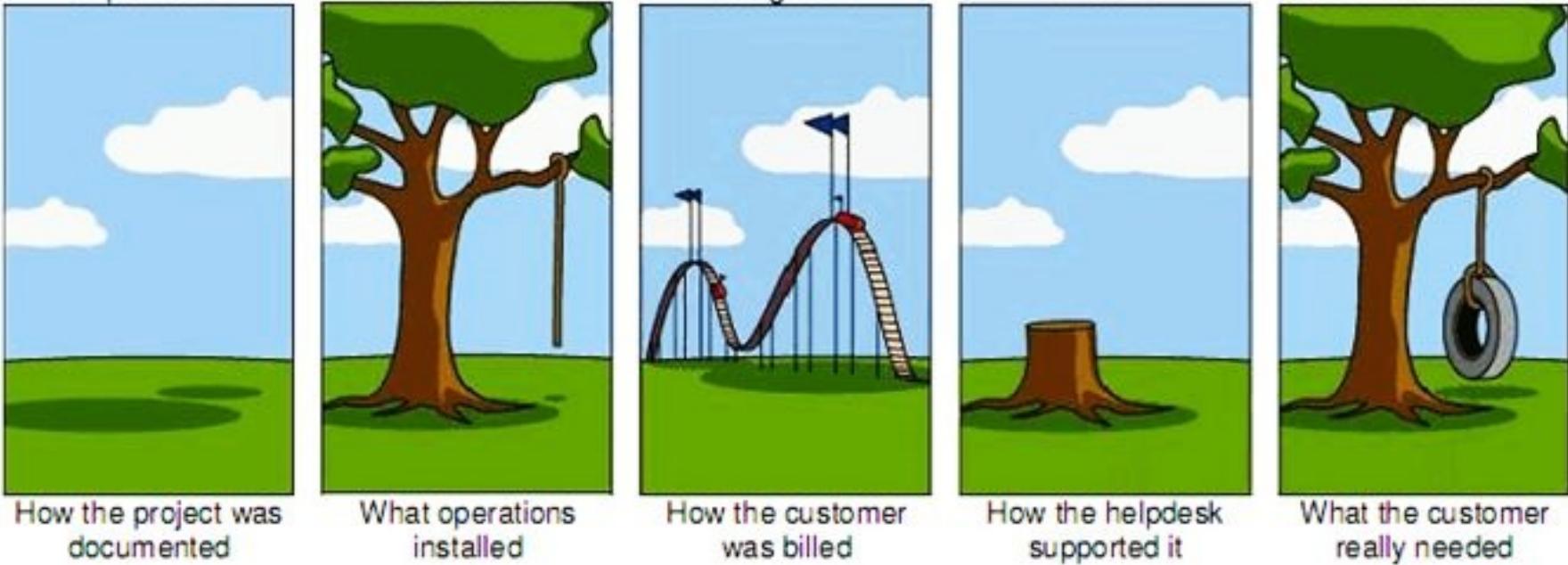
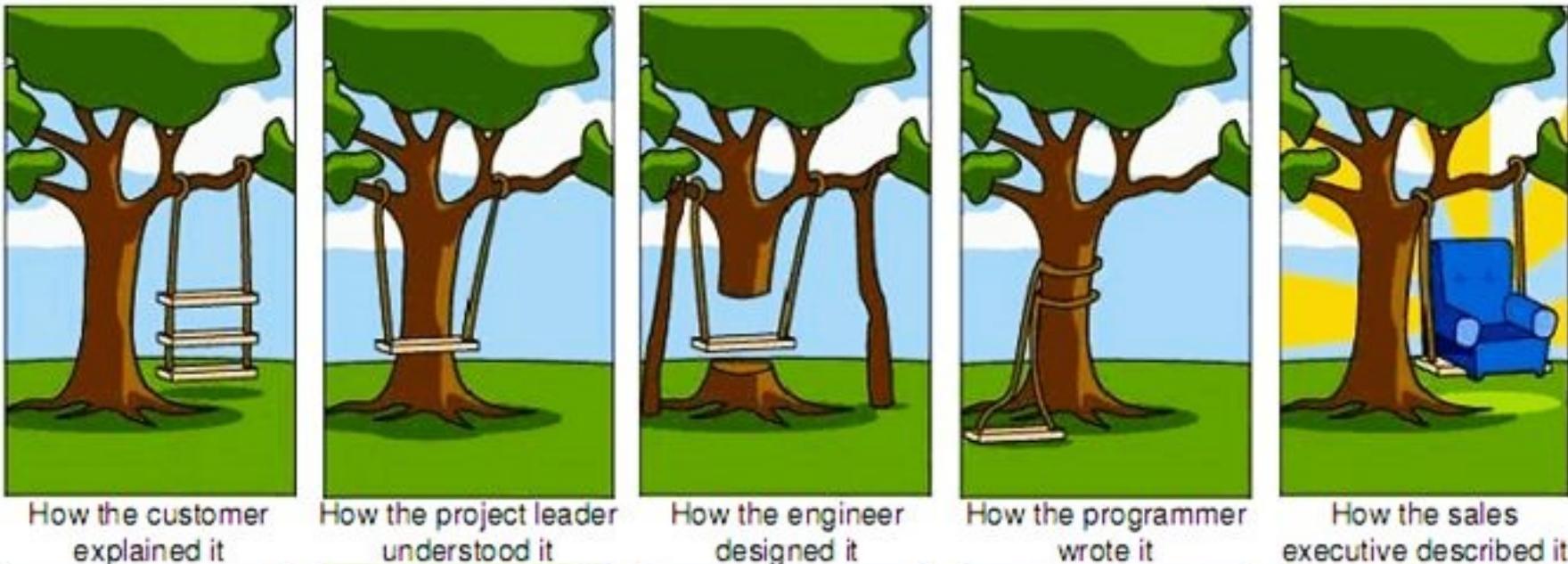
# OTHER SOFTWARE RELATED ACCIDENTS & INCIDENTS

## ■ Accidents & incidents

- Alarm flooding, **power distribution** failure, BP Grangemouth Scotland, 29th May - 10th June 2000
- failure in a data bus + faulty logic in the software => engine power loss, **Airbus** A340-642, 2005
- failed accelerometer + software bug => Faulty airspeed metering, **Boeing** 777-200, 2005
- Software bug => shutdown of radio communication between **ATC and aircraft**, 2004. disrupted 800 flights
- Safety-related software flaws => recall of 200,000 **pacemakers** in 1990-2000
- **radiotherapy machines** attacked by computer **viruses**, 2005
- Buggy software incorporate computer + connection between corporate and control systems networks => shutdown of the **nuclear power plant**, USA 2008
- Many software failures are actually "bugs" in the **detailed requirement specifications**, i.e., poor understanding of the very detailed requirements



Korean Air 747 in Guam, 200 deaths (1997): incorrect configuration of "minimum altitude" warning system



# REQUIREMENTS ENGINEERING

- ✓ The process of establishing the services that the customer requires from a system and the constraints under which it operates and is developed.

# WHAT IS A REQUIREMENT?

Requirement engineering = establishing the **services** that the customer requires from a system and the **constraints** under which it operates and is developed.

- ✓ Requirement = the descriptions of
  - the system services
  - and constraints
- ✓ It may range
  - from a high-level abstract statement
  - to a detailed mathematical functional specification.
- ✓ May serve a dual function
  - The basis for a bid for a contract - must be open to interpretation;
  - The basis for the contract itself - must be in detail;

# STAKEHOLDERS

- ✓ Any person or organization who is affected by the system in some way and so who has a legitimate interest
- ✓ Stakeholder types
  - End users
  - System managers
  - System owners
  - External stakeholders

# FUNCTIONAL AND NON-FUNCTIONAL REQUIREMENTS

## ✓ Functional requirements

- Statements of services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations.
- May state what the system should not do.

## ✓ Non-functional requirements

- Constraints on the services or functions offered by the system such as timing constraints, constraints on the development process, standards, etc.
- Often apply to the system as a whole rather than individual features or services.

### □ Domain requirements

- Constraints on the system from the domain of operation

# NON-FUNCTIONAL REQUIREMENTS IMPLEMENTATION

- ✓ Non-functional requirements may be more critical than functional requirements.
  - If these are not met, the system may be useless.
- ✓ Non-functional requirements may affect the overall architecture of a system
  - rather than the individual components
  - cross-cutting concern
- ✓ A single non-functional requirement
  - may generate a number of related functional requirements
  - and may also generate requirements that restrict existing requirements.

# NON-FUNCTIONAL REQUIREMENTS

- ✓ Define system properties and constraints
  - Properties: reliability, response time and storage requirements.
  - Constraints: I/O device capability, system representations, etc.
  
- ✓ Non-functional requirements may be more critical than functional requirements.
  - If these are not met, the system may be useless.

# NON-FUNCTIONAL REQUIREMENTS IMPLEMENTATION

- ✓ Non-functional requirements may affect the overall architecture of a system
  - └ rather than the individual components
  - └ cross-cutting concern
  
- ✓ A single non-functional requirement
  - └ may generate a number of related functional requirements
  - └ and may also generate requirements that restrict existing requirements.



# REQUIREMENTS ENGINEERING PROCESS

---

# REQUIREMENTS ENGINEERING PROCESSES

- ✓ Processes to “generate” all requirements
- ✓ Generic activities common to all processes
  - Requirements elicitation;
  - Requirements analysis;
  - Requirements validation;
  - Requirements management.
- ✓ In practice, RE is an iterative activity



*"The client kept changing the requirements on a daily basis, so we decided to freeze them until the next release."*

## The RE process:

### Requirements Engineering

#### Requirements Development

Elicitation

Analysis

Specification

Validation

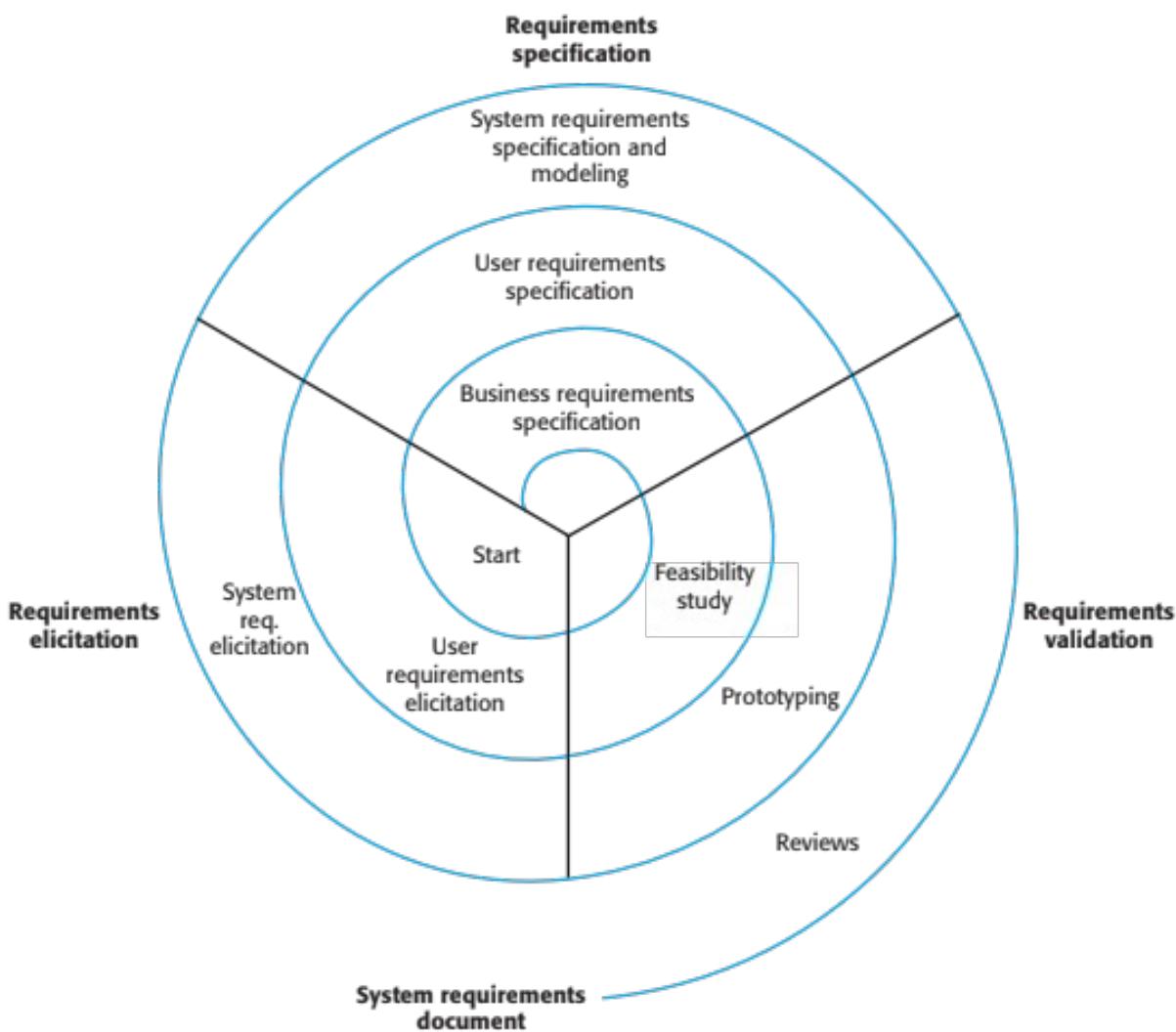
#### Requirements Management

Traceability

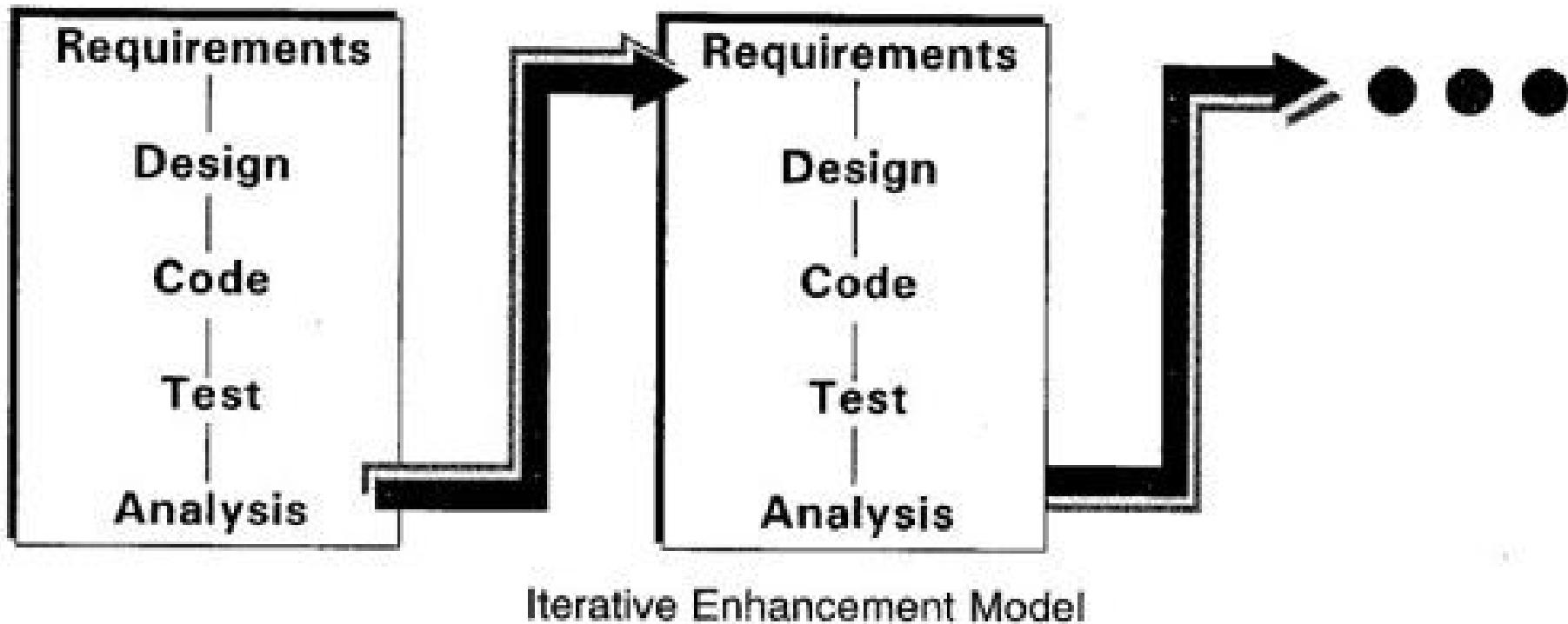
Change Management

Verification

# A SPIRAL VIEW OF THE REQUIREMENTS ENGINEERING PROCESS



# A AGILE VIEW OF THE REQUIREMENTS ENGINEERING PROCESS





# REQUIREMENT ELICITATION

---

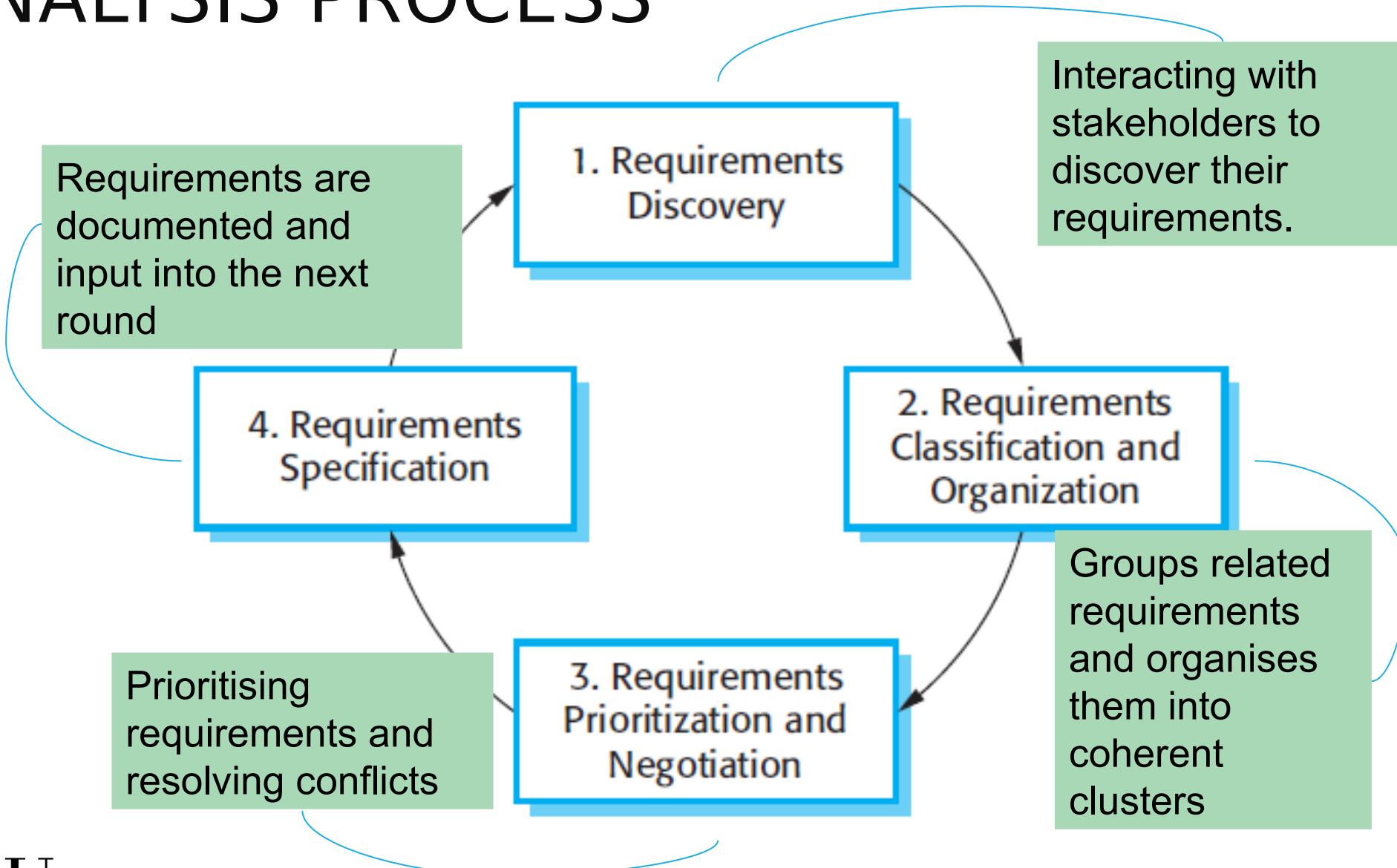
# REQUIREMENTS ELICITATION AND ANALYSIS

- ✓ ~ requirements elicitation or requirements discovery.
- ✓ Work with customers to find out:
  - I the application domain, the services and the operational constraints (system performance, hardware constraints, etc.).
- ✓ May involve
  - I end-users, managers, engineers involved in maintenance, domain experts, trade unions, etc. These are called stakeholders.

# PROBLEMS OF REQUIREMENTS ELICITATION

- ✓ Stakeholders don't know what they really want.
- ✓ Stakeholders express requirements in their own terms.
- ✓ Different stakeholders may have conflicting requirements.
- ✓ Organisational and political factors may influence the system requirements.
- ✓ The requirements change during the analysis process.

# THE REQUIREMENTS ELICITATION AND ANALYSIS PROCESS



# REQUIREMENTS DISCOVERY

- ✓ To gather information about the required and existing systems and distil the user and system requirements from this information.
  
- ✓ Main concerns:
  - Stakeholders
  - Discovery techniques/approaches/...

# DISCOVERY TECHNIQUE - INTERVIEWING



- ✓ Part of most RE processes.
- ✓ Types of interview
  - Closed vs Open => mixed?
- ✓ Be effective
  - Be open-minded, avoid pre-conceived ideas about the requirements and are willing to listen to stakeholders.
  - Prompt the interviewee to get discussions going using a springboard question, a requirements proposal, or by working together on a prototype system.

# DISCOVERY TECHNIQUES

- ✓ Observational techniques
  - ❑ used to understand operations and derive support requirements
- ✓ How
  - ❑ A social scientist spends a considerable time observing and analysing how people actually work.
  - ❑ People do not have to explain or articulate their work.
  - ❑ Social and organisational factors of importance may be observed.



# STORIES AND SCENARIOS

- ✓ Scenarios and use cases tell stories of how a system can be used.
- ✓ Stories and scenarios help us understand how a system may be used.
- ✓ Because they are stories, stakeholders can more easily comment on their own story.

## TOBI DAY

FAKE CROW PERSONA TEMPLATE

**AGE** 26  
**OCCUPATION** Record Store Manager  
**STATUS** Single  
**LOCATION** New York, NY  
**TIER** Enthusiast  
**ARCHETYPE** The Maestro

**MOTIVATIONS**

Incentive	█
Fear	█
Achievement	█
Growth	█
Power	█
Social	█

**GOALS**

- To grow a strong industry reputation
- To build an audio-pro portfolio
- To keep track of everything

**FRUSTRATIONS**

- Slow download times
- Data crashes
- Poor communication

**BIO**

Tobi has a day job at a record store, but on the side she does all kinds of production work for up-and-coming artists. She never hesitates to learn something new and she often acts as tech support for her friends and clients. She is usually working on a dozen projects at a time and is trying to establish herself in the industry, so she hates data crashes or anything that makes her look bad. Because she works alone and in her home, collaboration is everything.

**PERSONALITY**

Extrovert	█	Introvert	█
Sensing	█	Intuition	█
Thinking	█	Feeling	█
Judging	█	Perceiving	█

**TECHNOLOGY**

IT and Internet	█
Software	█
Mobile Apps	█
Social Networks	█

**Logos**

Audi

Coca-Cola

SONY

PreSonus

# REQUIREMENT ELICITATION TECHNIQUES

	Interviews	Workshops	Focus groups	Observations	Questionnaires	System interface analysis	User interface analysis	Document analysis
Mass-market software	X		X	X				
Internal corporate software	X	X	X	X		X		X
Replacing existing system	X	X		X		X	X	X
Enhancing existing system	X	X				X	X	X
New application	X	X				X		
Packaged software implementation	X	X		X		X		X
Embedded systems	X	X				X		X
Geographically distributed stakeholders	X	X			X			

**Suggested elicitation techniques by project characteristics**



# REQUIREMENTS SPECIFICATION

---

# REQUIREMENTS SPECIFICATION

- ✓ The process of writing down the user and system requirements in a requirements document.
- ✓ Notes:
  - User requirements have to be understandable by end-users and customers who do not have a technical background.
  - System requirements are more detailed requirements and may include more technical information.
  - The requirements may be part of a contract for the system development

# WAYS OF WRITING A SYSTEM REQUIREMENTS SPECIFICATION

Notation	Description
Natural language	Sentences in natural language. Each sentence should express one requirement.
Structured natural language	Natural language statements on a standard form or template
Design description languages	Uses a language like a programming language, but with more abstract features
Graphical notations	Graphical models, supplemented by text annotations (best for functional requirements); UML use case and sequence diagrams are commonly used.
Mathematical specifications	Based on mathematical concepts such as finite-state machines or sets; Can reduce the ambiguity but hard to understand (and hard to check manually)

# NATURAL LANGUAGE SPECIFICATION

- ✓ Used for writing requirements because it is expressive, intuitive and universal.
  - The requirements can be understood by users and customers.
- ✓ Problems
  - Lack of clarity: Precision is difficult without making the document difficult to read.
  - Requirements confusion: Functional and non-functional requirements tend to be mixed-up.
  - Requirements amalgamation: Several different requirements may be expressed together.

## An insulin pump control system

An insulin pump is a medical system that simulates the operation of the pancreas (an internal organ). The software controlling this system is an embedded system, which collects information from a sensor and controls a pump that delivers a controlled dose of insulin to a user.

People who suffer from diabetes use the system. Diabetes is a relatively common condition where the human pancreas is unable to produce sufficient quantities of a hormone called insulin. Insulin metabolises glucose (sugar) in the blood. The conventional treatment of diabetes involves regular injections of genetically engineered insulin. Diabetics measure their blood sugar levels using an external meter and then calculate the dose of insulin that they should inject.

# EXAMPLE REQUIREMENTS FOR THE INSULIN PUMP SOFTWARE SYSTEM

- ✓ Req 3.2. The system shall measure the blood sugar and deliver insulin, if required, every 10 minutes.
  
- ✓ Req 3.6. The system shall run a self-test routine every minute with the conditions to be tested and the associated actions defined in Table 1.

# STRUCTURED SPECIFICATIONS

- ✓ Writing on a standard form or template:
  - Name
  - Inputs, outputs
  - The information needed for the computation
  - Action
  - Pre and post conditions (if appropriate)
  - The side effects (if any)
  
- ✓ This works well for some types of requirements e.g. requirements for embedded control system

**Insulin Pump/Control Software/SRS/3.3.2**

Function	Compute insulin dose: safe sugar level
Description	Computes the dose of insulin to be delivered when the current measured sugar level is in the safe zone between 3 and 7 units.
Inputs	Current sugar reading (r2); the previous two readings (r0 and r1).
Source	Current sugar reading from sensor. Other readings from memory.
Outputs	CompDose—the dose in insulin to be delivered
Destination	Main control loop.
Action	CompDose is zero if the sugar level is stable or falling or if the level is increasing but the rate of increase is decreasing. If the level is increasing and the rate of increase is increasing, then CompDose is computed by dividing the difference between the current sugar level and the previous level by 4 and rounding the result. If the result, is rounded to zero then CompDose is set to the minimum dose that can be delivered.
Requirements	Two previous readings so that the rate of change of sugar level can be computed
Pre-condition	The insulin reservoir contains at least the maximum allowed single dose of insulin

# TABULAR SPECIFICATION

- ✓ Particularly useful when you have to define a number of possible alternative courses of action.
- ✓ Example:

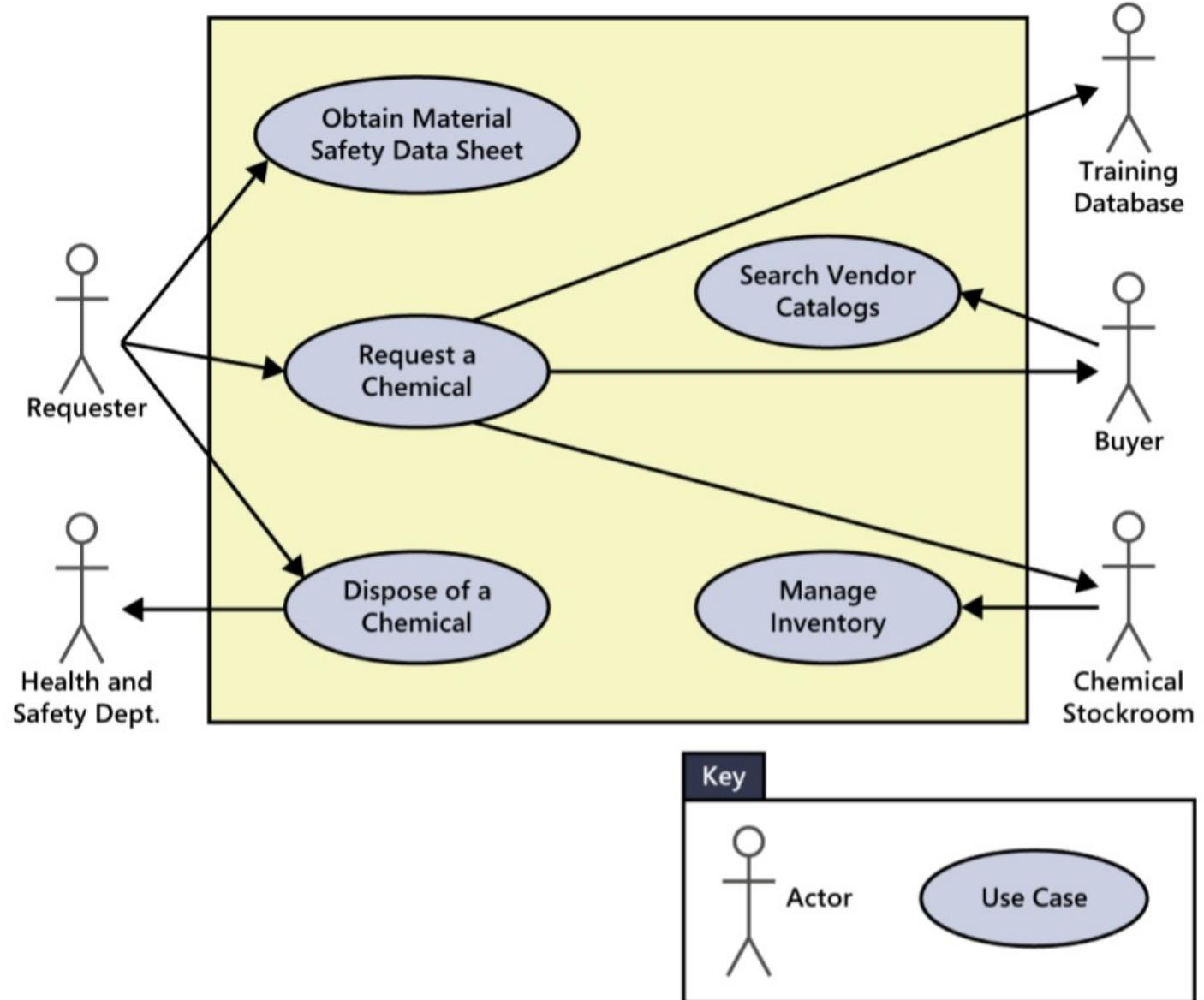
Condition	Action
Sugar level falling ( $r_2 < r_1$ )	$\text{CompDose} = 0$
Sugar level stable ( $r_2 = r_1$ )	$\text{CompDose} = 0$
Sugar level increasing and rate of increase decreasing ( $(r_2 - r_1) < (r_1 - r_0)$ )	$\text{CompDose} = 0$
Sugar level increasing and rate of increase stable or increasing ( $(r_2 - r_1) \geq (r_1 - r_0)$ )	$\text{CompDose} = \text{round}((r_2 - r_1)/4)$ <b>If rounded result = 0 then</b> $\text{CompDose} = \text{MinimumDose}$

# USE CASES

- ✓ Use-cases are a kind of scenario
  - identify the actors in an interaction and which describe the interaction itself
  - Included in the UML
- ✓ A set of use cases should describe all possible interactions with the system.
- ✓ UML sequence diagrams may be used to add detail to use-cases
  - show the sequence of event processing in the system

# USE CASE DIA

- Actors
- Use cases
- Association
- System boundary boxes



# USE CASE TEMPLATE AND EXAMPLE

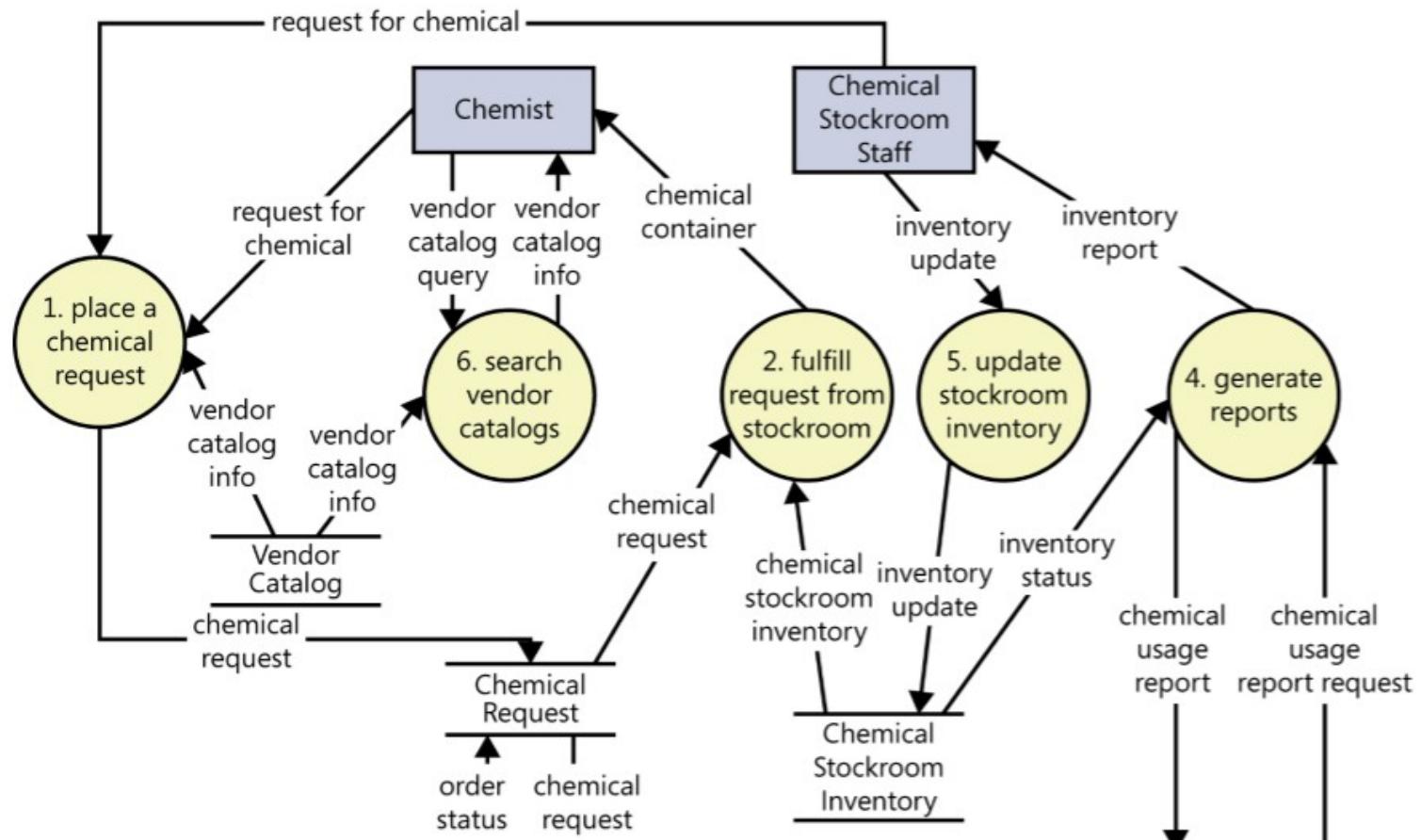
ID and Name:	UC-4 Request a Chemical		
Created By:	Lori	Date Created:	8/22/13
Primary Actor:	Requester	Secondary Actors:	Buyer, Chemical Stockroom, Training Database
Description:	The Requester specifies the desired chemical to request by entering its name or chemical ID number or by importing its structure from a chemical drawing tool. The system either offers the Requester a container of the chemical from the chemical stockroom or lets the Requester order one from a vendor.		
Trigger:	Requester indicates that he wants to request a chemical.		
Preconditions:	PRE-1. User's identity has been authenticated. PRE-2. User is authorized to request chemicals. PRE-3. Chemical inventory database is online.		
Postconditions:	POST-1. Request is stored in the CTS. POST-2. Request was sent to the Chemical Stockroom or to a Buyer.		

# USE CASE TEMPLATE AND EXAMPLE (CONT’)

ID and Name:	UC-4 Request a Chemical
Normal Flow:	<p><b>4.0 Request a Chemical from the Chemical Stockroom</b></p> <ol style="list-style-type: none"><li>1. Requester specifies the desired chemical.</li><li>2. System lists containers of the desired chemical that are in the chemical stockroom, if any.</li><li>3. System gives Requester the option to View Container History for any container.</li><li>4. Requester selects a specific container or asks to place a vendor order (see 4.1).</li><li>5. Requester enters other information to complete the request.</li><li>6. System stores the request and notifies the Chemical Stockroom.</li></ol>
Alternative Flows:	<p><b>4.1 Request a Chemical from a Vendor</b></p> <ol style="list-style-type: none"><li>1. Requester searches vendor catalogs for the chemical (see 4.1.E1).</li><li>2. System displays a list of vendors for the chemical with available container sizes, grades, and prices.</li><li>3. Requester selects a vendor, container size, grade, and number of containers.</li><li>4. Requester enters other information to complete the request.</li><li>5. System stores the request and notifies the Buyer.</li></ol>
Exceptions:	<p><b>4.1.E1 Chemical Is Not Commercially Available</b></p>

# DATA FLOW DIAGRAM

✓ How data moves through a system



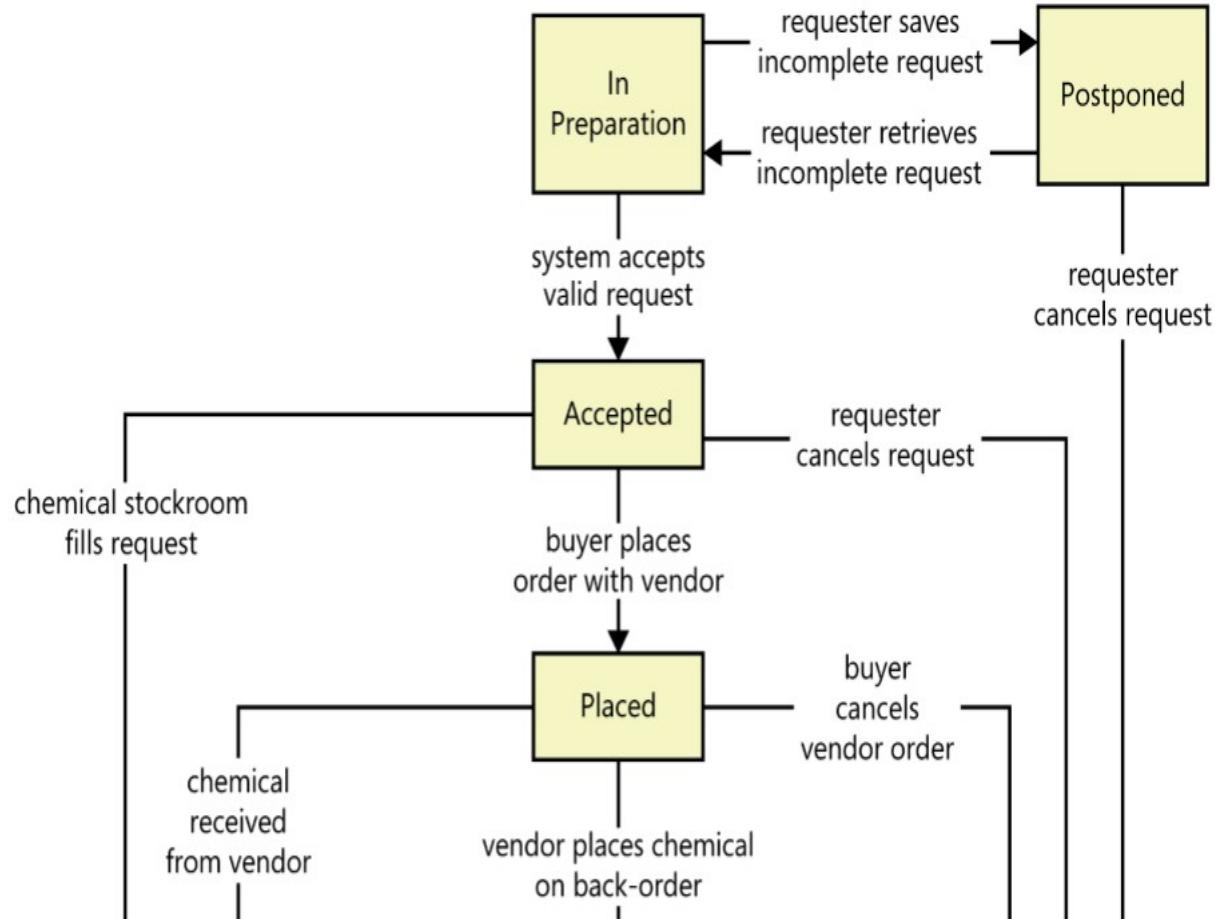
# DECISION TABLE/DECISION TREE

✓ Complex logics

Requirement Number	1	2	3	4	5
Condition	1	2	3	4	5
User is authorized	F	T	T	T	T
Chemical is available	—	F	T	T	T
Chemical is hazardous	—	—	F	T	T
Requester is trained	—	—	—	F	T
Action			X		X
Accept request			X		X
Reject request	X	X		X	

# STATE-TRANSITION DIAGRAM

- ✓ A set of complex state changes in natural language creates a high probability of overlooking a permitted state change or including a disallowed change.



# THE SOFTWARE REQUIREMENTS DOCUMENT

- ✓ The software requirements document is the official statement of what is required of the system developers.
- ✓ Should include both a definition of user requirements and a specification of the system requirements.
- ✓ It is NOT a design document. As far as possible, it should set of WHAT the system should do rather than HOW it should do it.



# REQUIREMENT VALIDATION

---

# REQUIREMENTS VALIDATION

- ✓ Requirements error can be costly !

Cost to Fix Requirements Error (in Ratios)			
Development Discipline/Phase	Composite of Studies (NASA)*	NASA Software*	Davis' Composite†
Requirements	1 (baseline)	1 (baseline)	1 (baseline)
Design	5x	8x	2.5x – 5x
Code	10x	16x	5x – 10x
Test	50.5x	21x	Unit Test: 10x – 20x Acceptance Test: 25x – 50x
Post-Deployment	n/a	29x	100x – 200x

\* NASA reviewed McGibbon (2003), Pavlina (2003), Digital (2003), Rothman (2002), Hoffman (2001), Rothman (2000), and Boehm (1981) and assessed the median of their data, presented here.

# NASA analyzed its own software projects to assess the relevance of reviewed studies.

# Leffingwell & Widrig reviewed GTE, TRW, IBM, and Davis (2003), who himself reviewed several studies. The results of Davis' work are presented here.

# REQUIREMENTS CHECKING

- ✓ Validity.
  - Does the system provide the functions which best support the customer's needs?
- ✓ Consistency.
  - Are there any requirements conflicts?
- ✓ Completeness.
  - Are all functions required by the customer included?
- ✓ Realism.
  - Can the requirements be implemented given available budget and technology
- ✓ Verifiability.
  - Can the requirements be checked?

# REQUIREMENTS VALIDATION TECHNIQUES

- ✓ Requirements reviews
  - Systematic manual analysis of the requirements.
- ✓ Prototyping
  - Using an executable model of the system to check requirements.
- ✓ Test-case generation
  - Developing tests for requirements to check testability.

# REQUIREMENTS MANAGEMENT



*"The client kept changing the requirements on a daily basis, so we decided to freeze them until the next release."*

# CHANGING REQUIREMENTS

- ✓ The business and technical environment of the system always changes after installation.
- ✓ The people who pay for a system and the users of that system are rarely the same people.
- ✓ Large systems usually have a diverse user community, with many users having different requirements and priorities that may be conflicting or contradictory.

# REQUIREMENTS MANAGEMENT

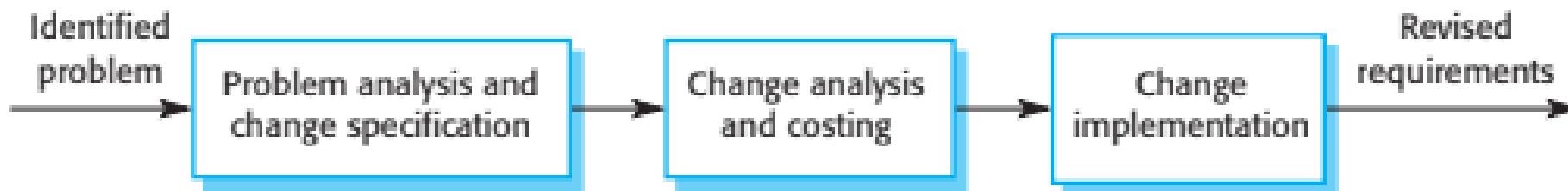
- ✓ Requirements management is the process of managing changing requirements during the requirements engineering process and system development.
- ✓ New requirements emerge as a system is being developed and after it has gone into use.
- ✓ You need to keep track of individual requirements and maintain links between dependent requirements so that you can assess the impact of requirements changes. You need to establish a formal process for making change proposals and linking these to system requirements.

# REQUIREMENTS MANAGEMENT PLANNING

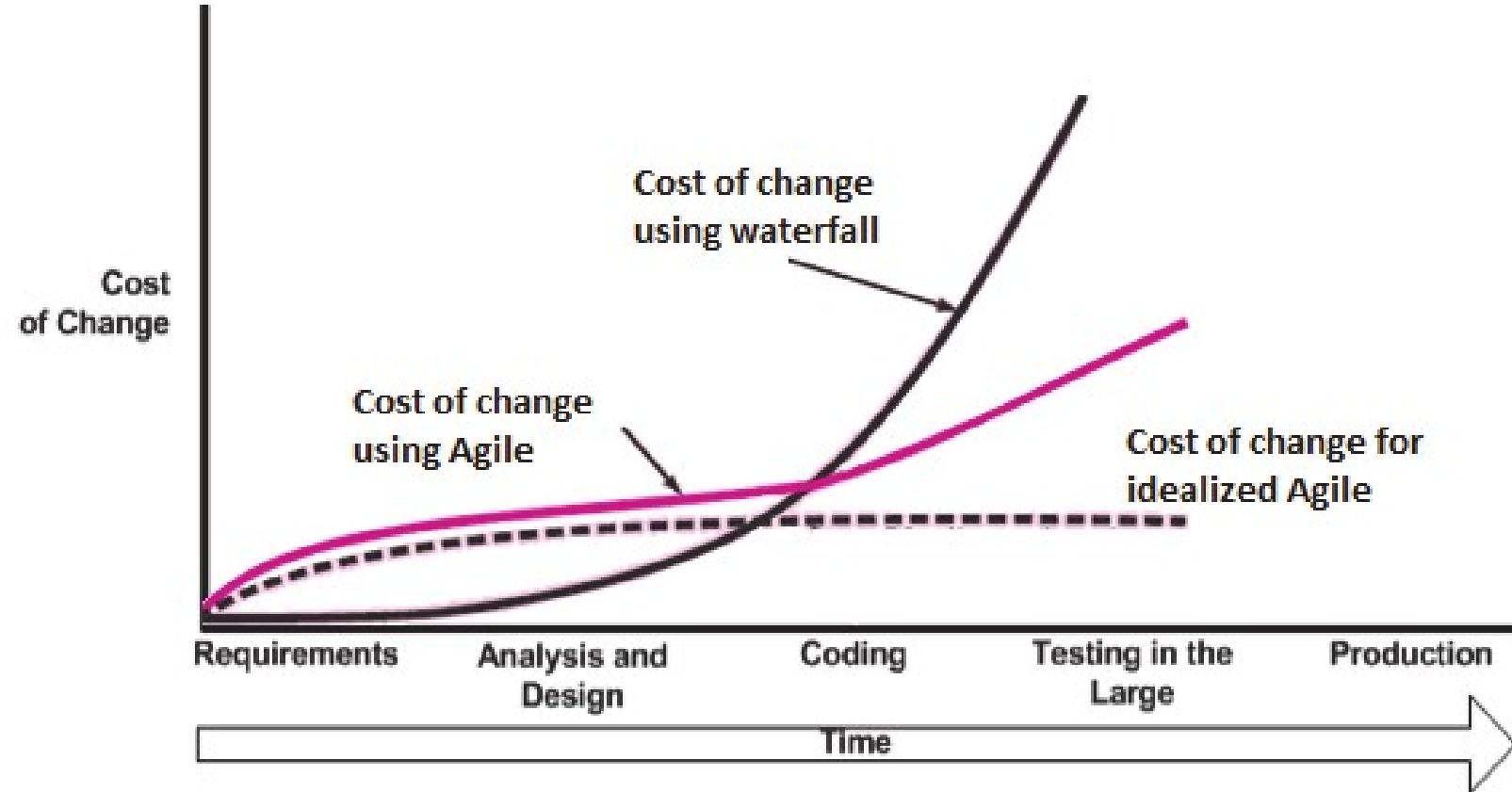
- ✓ Establishes the level of requirements management detail that is required.
  
- ✓ Requirements management decisions:
  - Requirements identification
  - A change management process
  - Traceability policies
  - Tool support

# REQUIREMENTS CHANGE MANAGEMENT

- ✓ Deciding if a requirements change should be accepted
  - Problem analysis and change specification
  - Change analysis and costing
  - Change implementation



# RESPONSE TO CHANGE: AGILE APPROACH!



# SUMMARY

- ✓ Requirements: what the system should do and constraints on its operation and implementation.
- ✓ Functional requirements = the services
- ✓ Non-functional requirements = constraints (development & use)
  - I apply to the system as a whole.
- ✓ The software requirements document (i.e. SRS) is an agreed statement of the system requirements.
- ✓ The RE process is an iterative process
  - I requirements elicitation, specification and validation,



# GOAL ORIENTED REQUIREMENT ENGINEERING



*"The client kept changing the requirements on a daily basis, so we decided to freeze them until the next release."*

# WHAT IS GORE [3]?

- ✓ GORE is approach and modeling method to
  - Identify goals and sub-goals of software systems
  - Identify who and how to achieve the goals
  - Identify domain hypothesis and conflicts
- ✓ Similar principle to use cases and user stories
- ✓ Can be applicable to cyber-physical systems and user-system interaction systems

# PROCESSES TO CREATE GORE MODELS

- ✓ Initial goal identification
- ✓ Eliciting new goals through WHY and HOW questions
- ✓ Identifying agents and responsibility assignments
- ✓ Identifying operations
- ✓ Anticipating obstacles
- ✓ Identifying domain hypothesis
- ✓ Handling conflicts

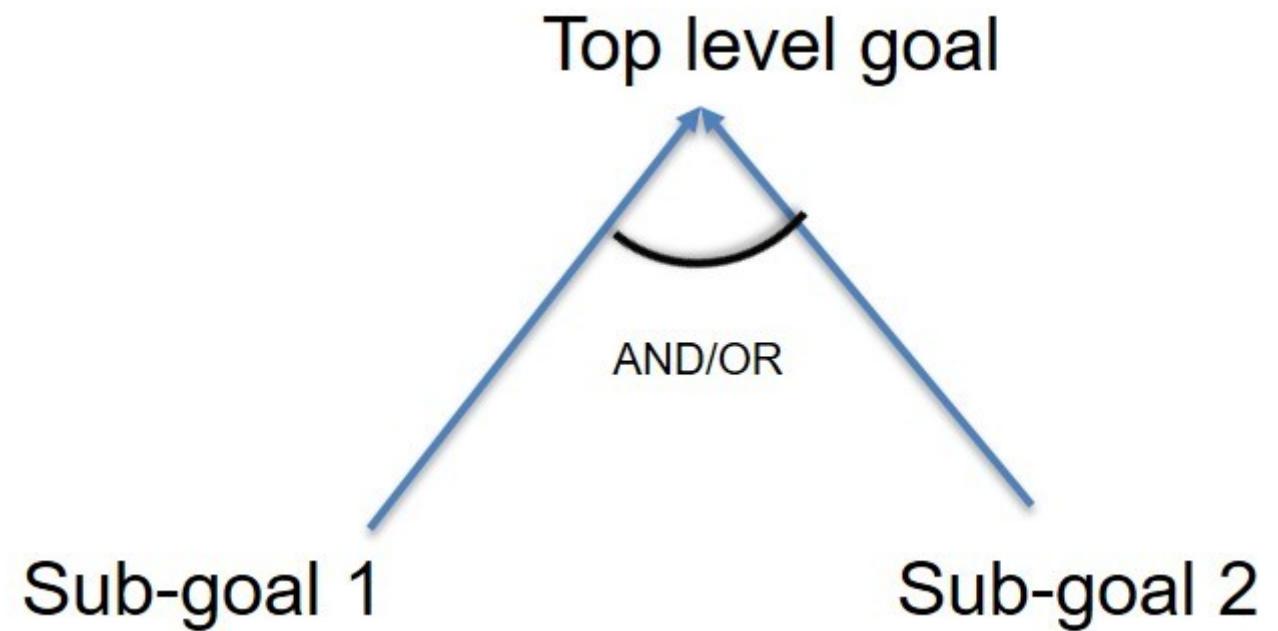


# GOAL MODEL

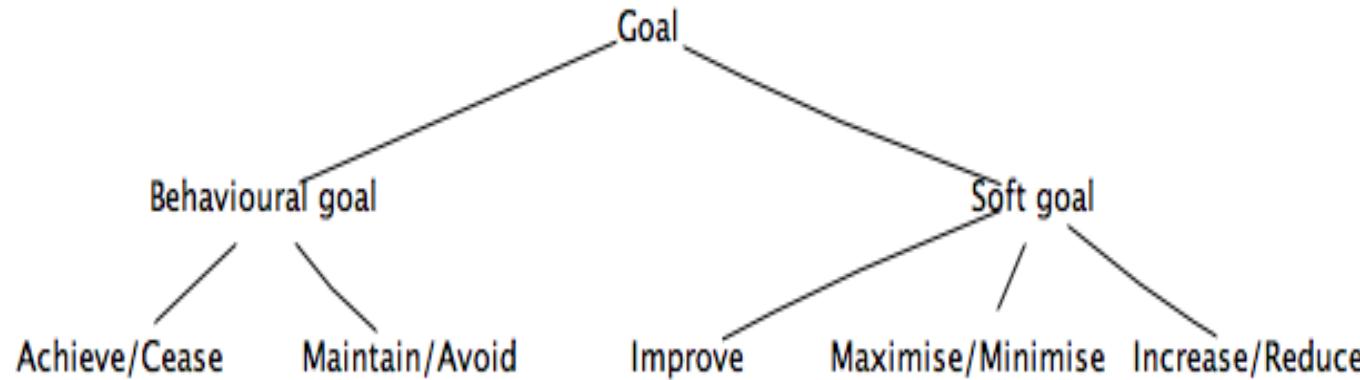
- A goal is an objective the system under consideration should achieve
- Goal is condition or situation that can be achieved or not. Goal is used to define the functional requirements of the system.
- Goals provide a precise criterion for sufficient completeness of a requirements specification

# GOAL MODEL

- Hierarchy and relationship of goals
- Goals can be of different types

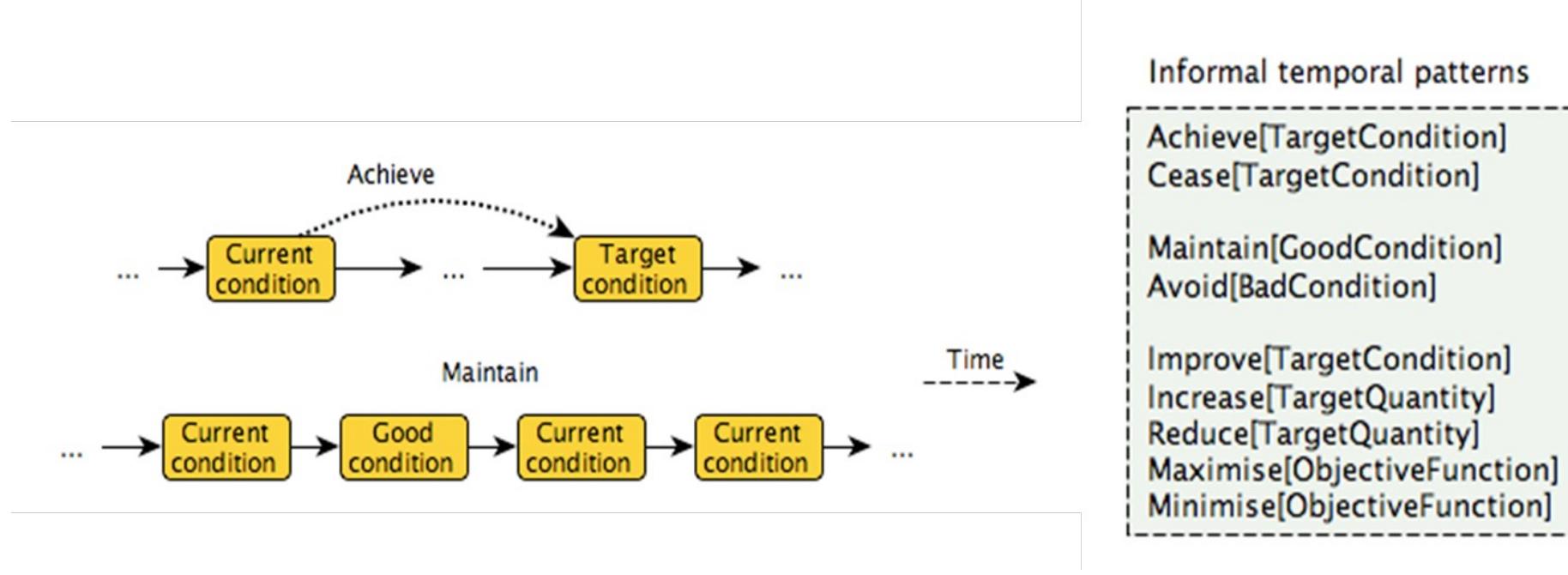


# GOAL MODEL USING BEHAVIOR AND SOFT GOAL TYPES



- Soft goal
  - Satisfaction cannot be established in a clear-cut sense
  - Useful for comparing alternative goal refinements
- Hard/behavioral goal
  - Satisfaction can be established through verification

# GOAL MODEL USING BEHAVIOR GOAL TYPES



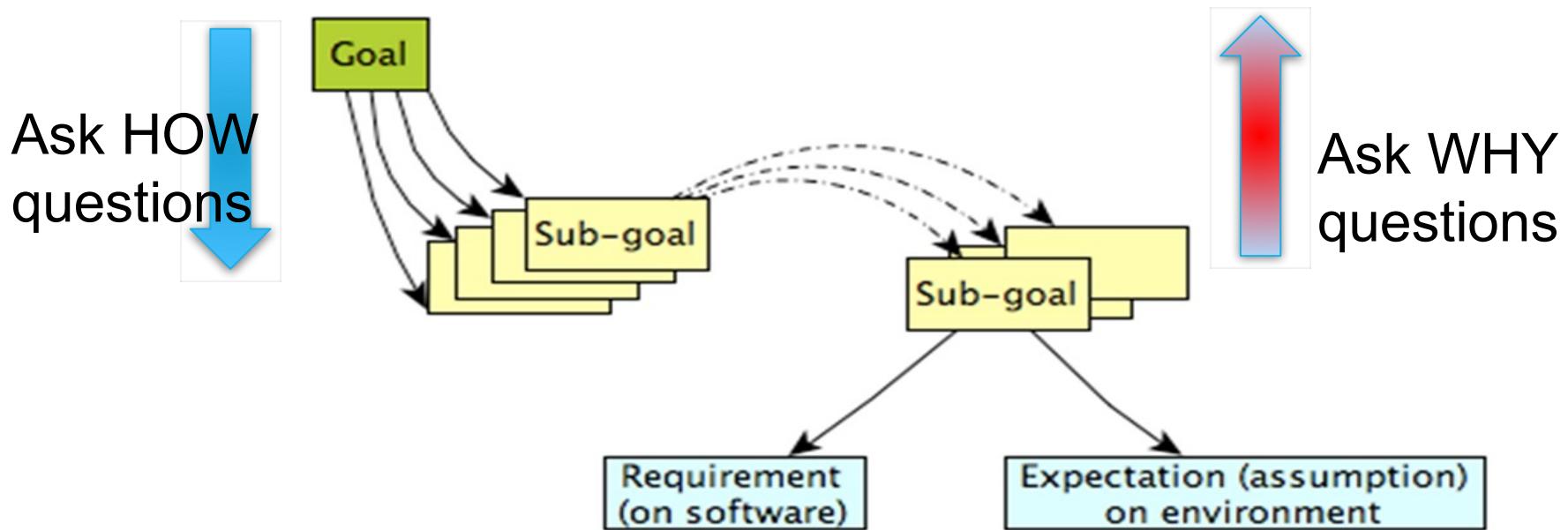
- Achieve: Some target property to be **eventually satisfied**
- Maintain: Some target property to be **permanently satisfied**
- Avoid: Some target property to be **permanently denied**
- ...

# PROCESSES TO CREATE GORE MODELS

- ✓ Initial goal identification
- ✓ Eliciting new goals through WHY and HOW questions
- ✓ Identifying agents and responsibility assignments
- ✓ Identifying operations
- Anticipating obstacles
- Identifying domain hypothesis
- Handling conflicts

# ELICITING GOALS

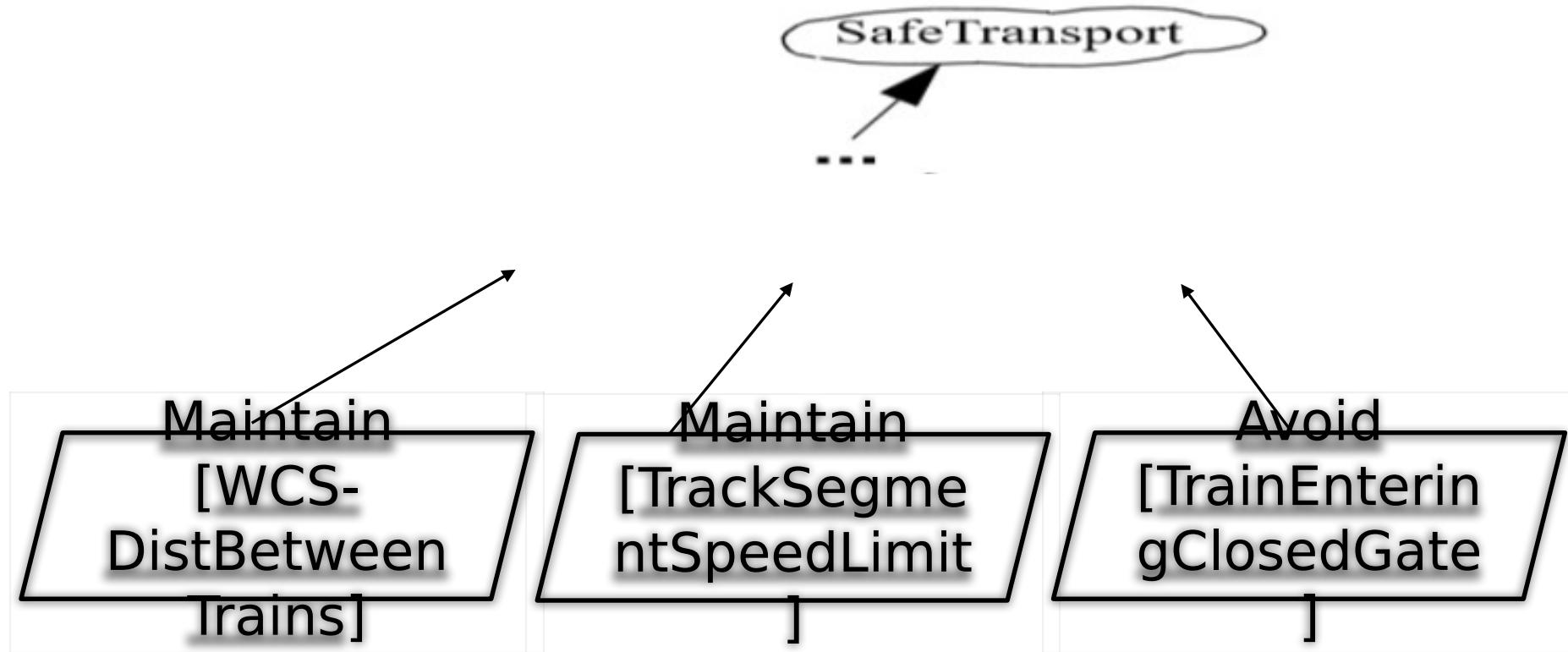
Each sub-goal is refined into finer-grained goals until we reach a software requirement associated with a single agent and expectation (assumption) on the environment.



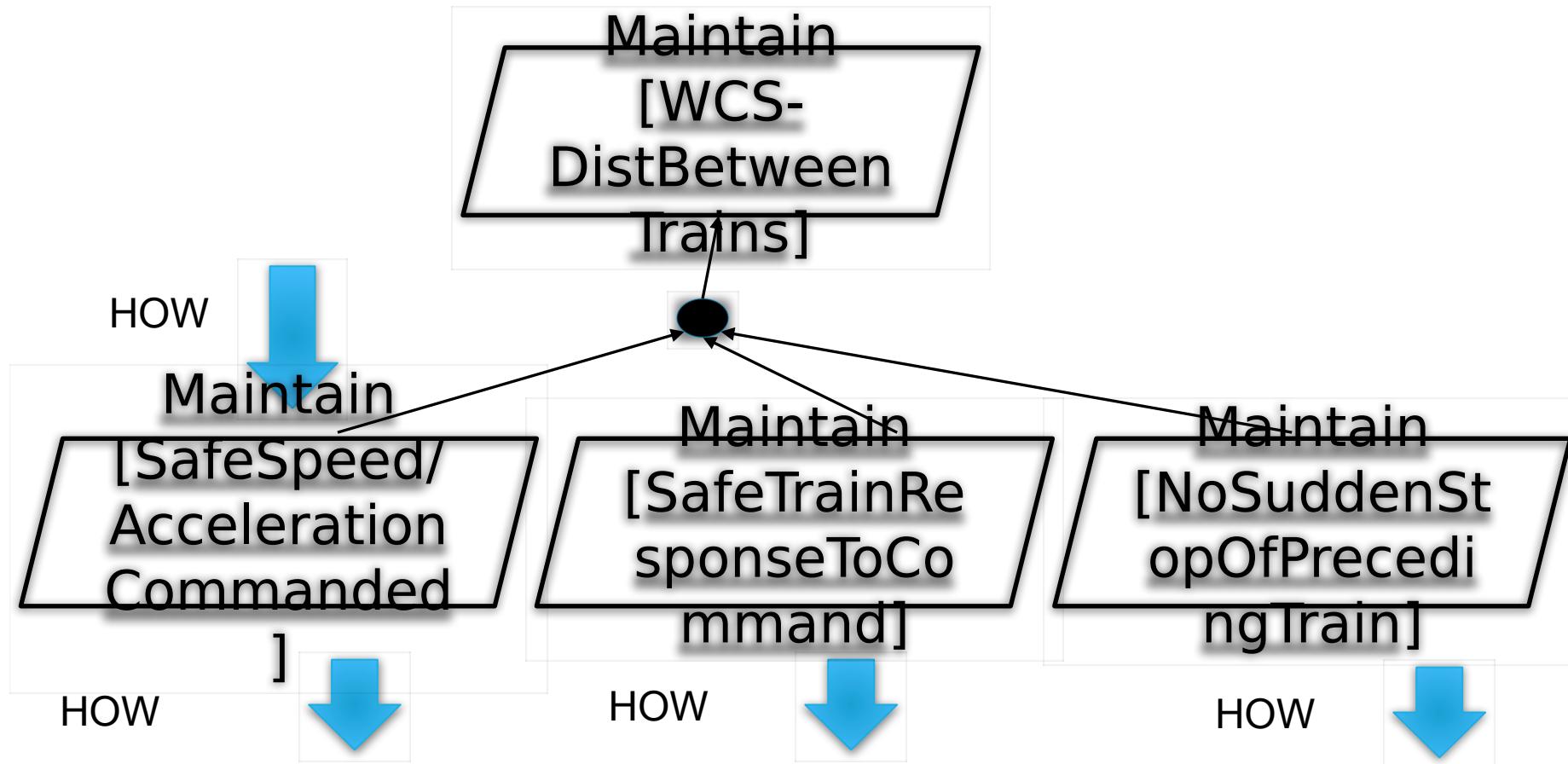
# CASE STUDY: TRAIN TRANSPORT SAFETY



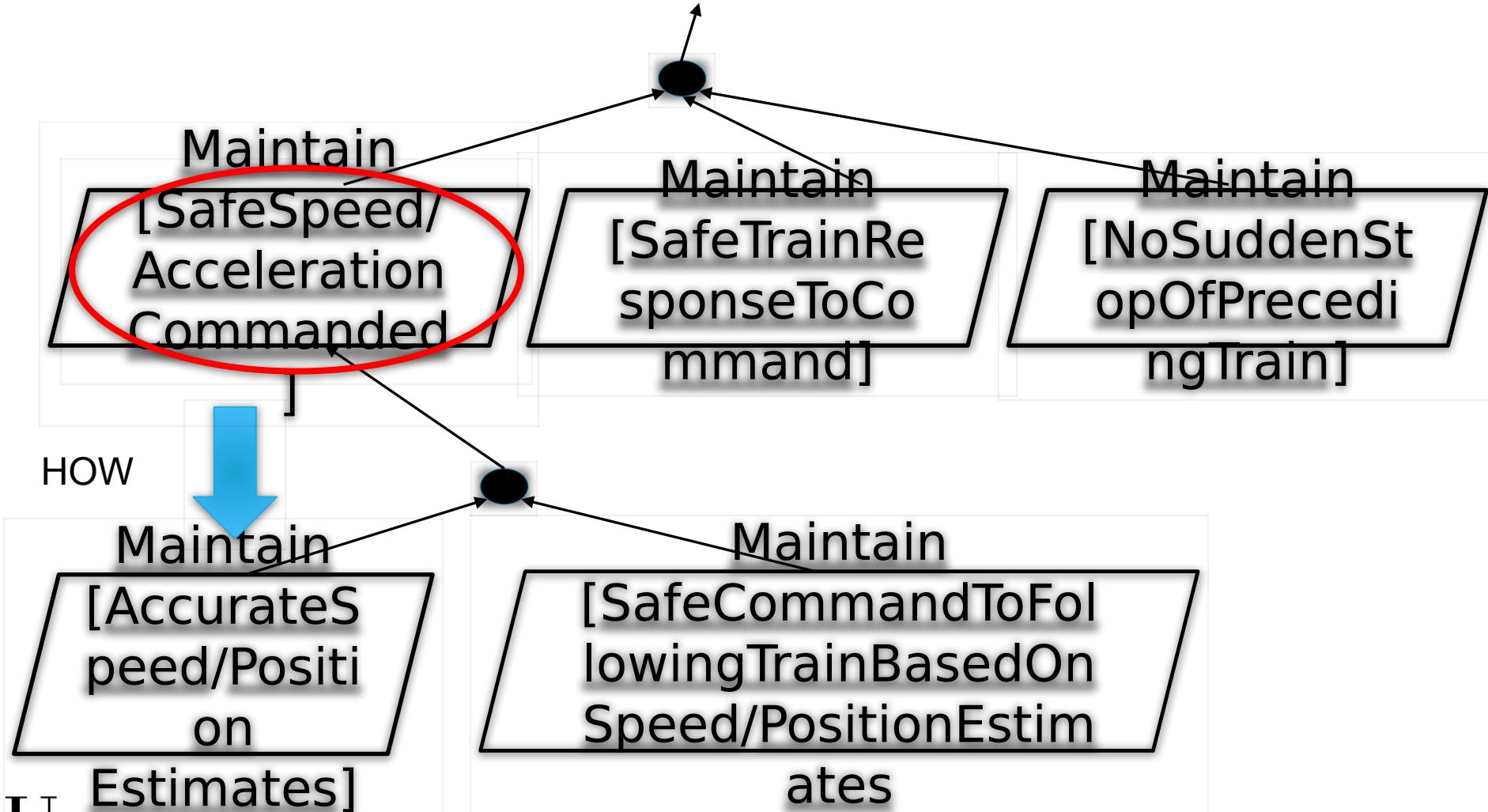
# SOFT AND BEHAVIOR GOALS



# ELICITING NEW GOALS THROUGH “HOW” QUESTIONS

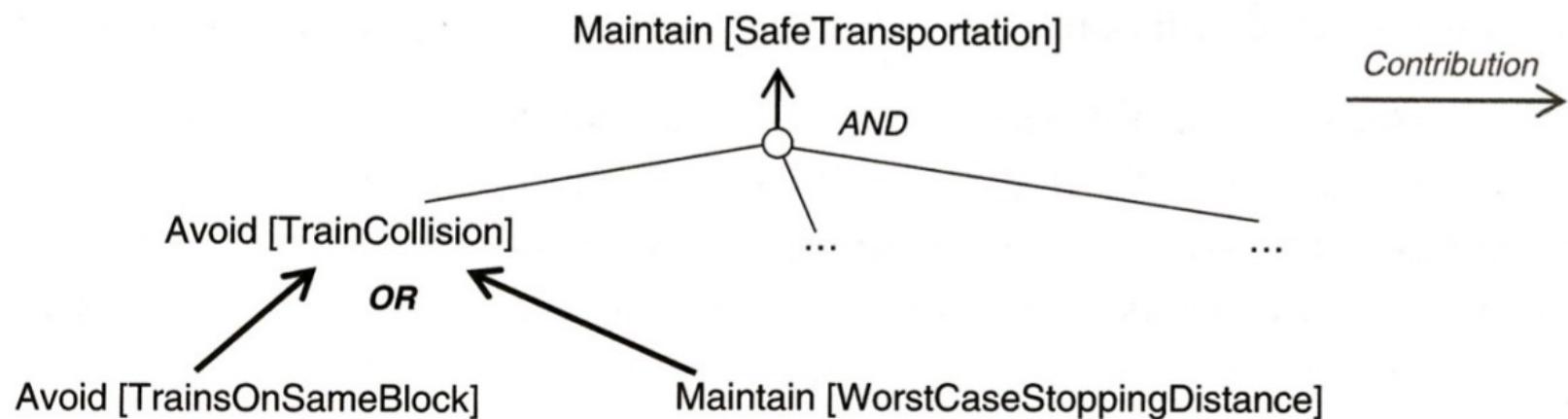


# CONTINUE ELICITING NEW GOALS THROUGH “HOW” QUESTIONS



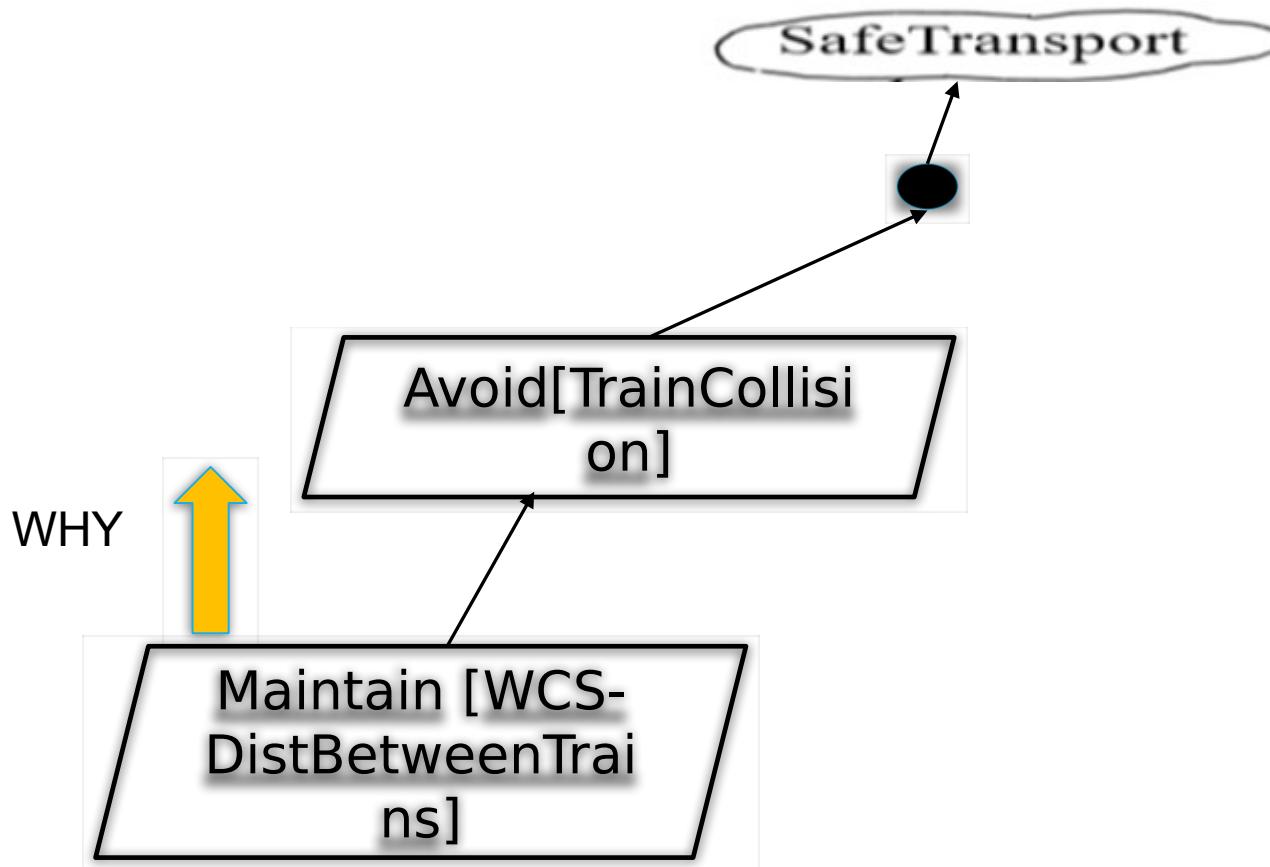
# GOALS REFINEMENT

- Mean-ends relationship ( $\rightarrow$ )
- Decomposition relationship ( $\perp\!\!\!\perp$ )
- Contribution relationship ( $\rightarrow$ )
- Dependency relationship ( $\perp\!\!\!\perp$ )

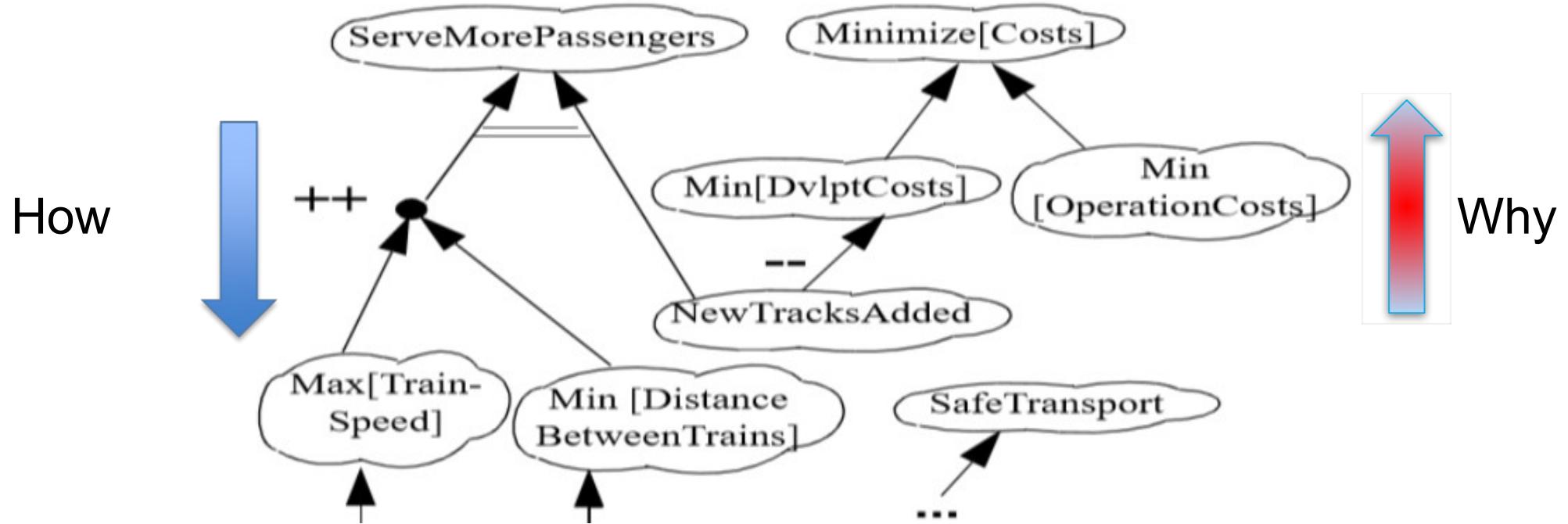


# ELICITING NEW GOALS THROUGH “WHY” QUESTIONS

- ✓ High level goals are identified by asking WHY questions



# AATC SOFT GOALS



+ : Contribute

- : Conflict

++ : Contribute strongly

-- : Conflict strongly

LINKS: Double line : OR

Black circle : AND

... : More sub-goals

# PROCESSES TO CREATE GORE MODELS

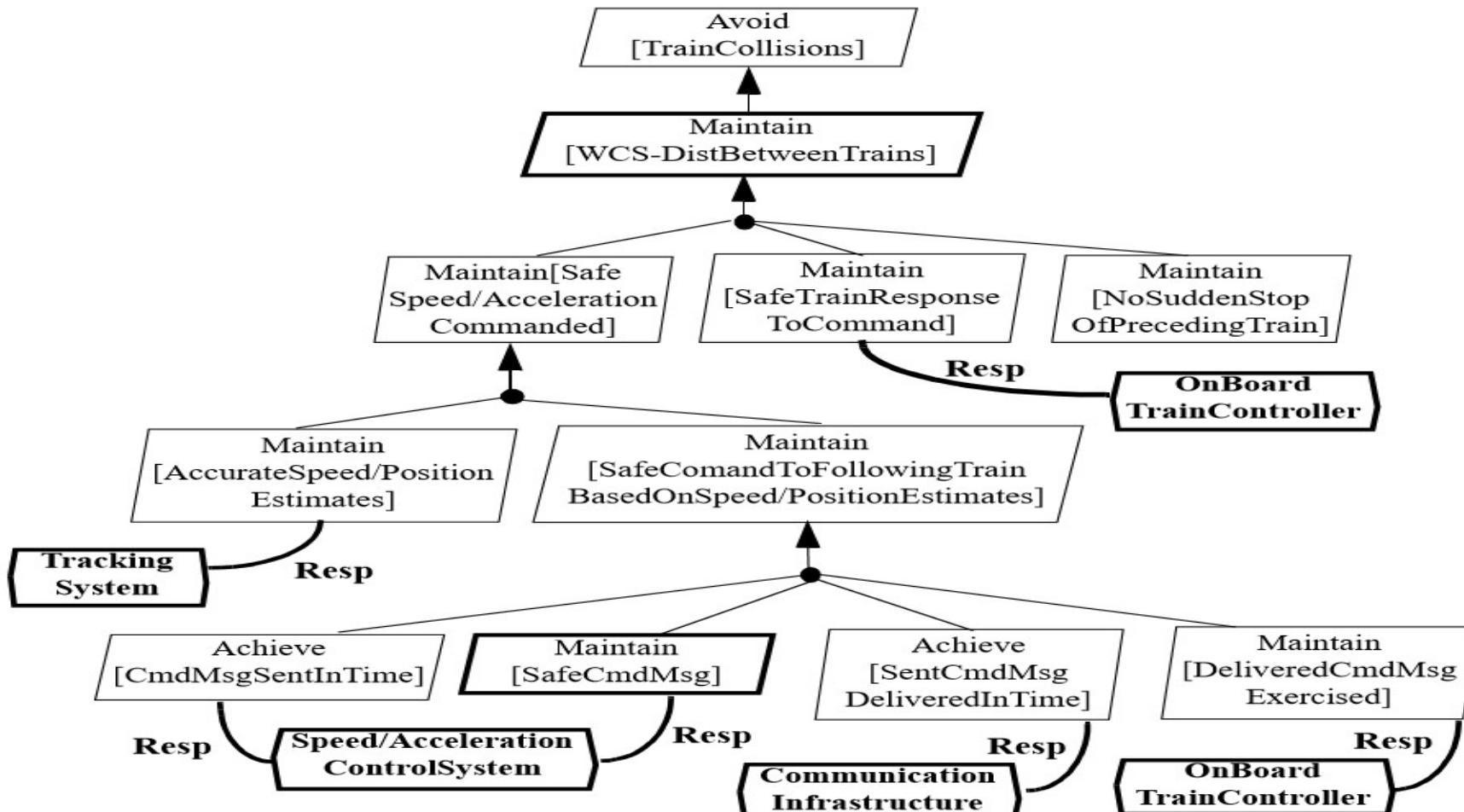
- ✓ Initial goal identification
- ✓ Eliciting new goals through WHY and HOW questions
- ✓ Identifying agents and responsibility assignments
- ✓ Identifying operations
- Anticipating obstacles
- Identifying domain hypothesis
- Handling conflicts

# OBJECT MODEL

- ✓ Agent: independent **active** object, e.g.,
  - Speed/Acceleration control system
  - On-board train controller
  - Tracking systems
  
- ✓ Agent has entities
  - E.g., tracking systems have several sensors

# AGENT RESPONSIBILITY MODEL

- A goal has to be placed under the responsibility of an **agent**.



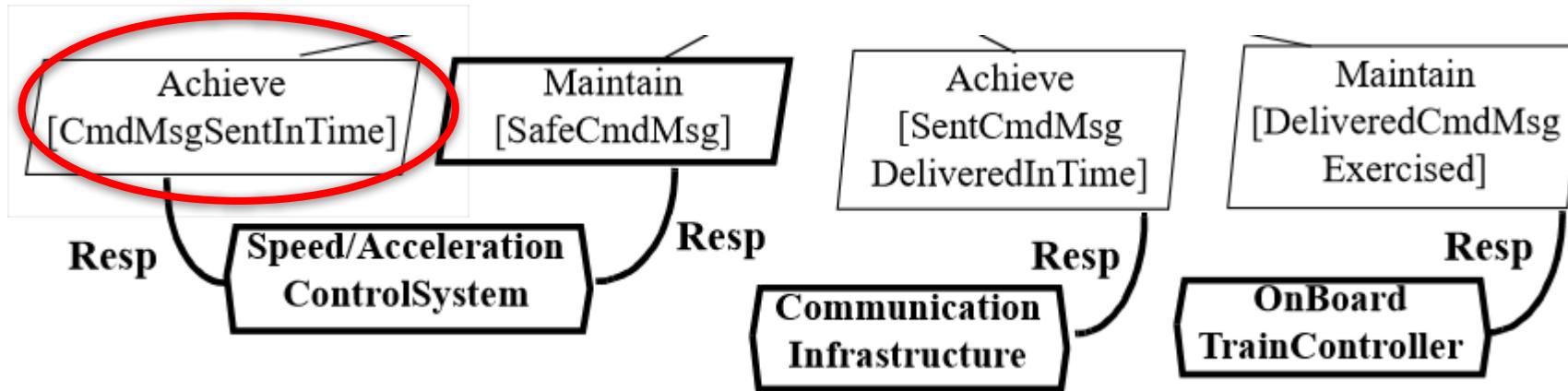
# PROCESSES TO CREATE GORE MODELS

- ✓ Initial goal identification
- ✓ Eliciting new goals through WHY and HOW questions
- ✓ Identifying agents and responsibility assignments
- ✓ Identifying operations
  - Anticipating obstacles
  - Identifying domain hypothesis
  - Handling conflicts

# OPERATION MODEL

- ✓ Operation is input-output relation over objects
- ✓ Defines state transitions
- ✓ Characterized by pre-, post-, and trigger conditions

# OPERATION MODEL – TRAIN CONTROL SYS.

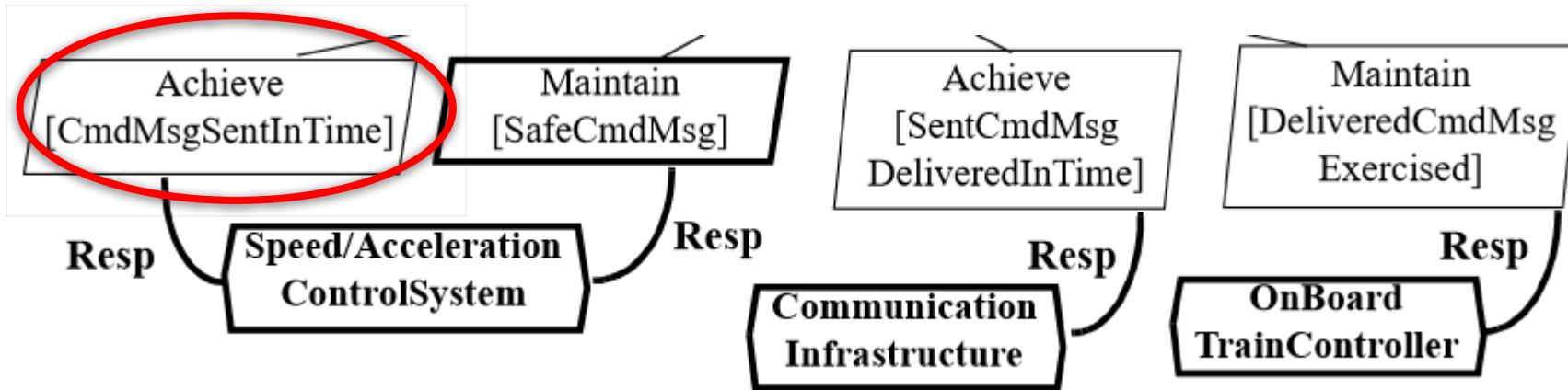


- ✓ Operation: SendCommandMessage
  - ✓ Input: Train ID
  - ✓ Output: Command to the train
  - ✓ Pre-condition: Message not sent
  - ✓ Post-condition: Message sent to correct train
- } State transition

# PROCESSES TO CREATE GORE MODELS

- ✓ Initial goal identification
- ✓ Eliciting new goals through WHY and HOW questions
- ✓ Identifying agents and responsibility assignments
- ✓ Identifying operations
- Anticipating obstacles
- Identifying domain hypothesis
- Handling conflicts

# ANTICIPATING OBSTACLES

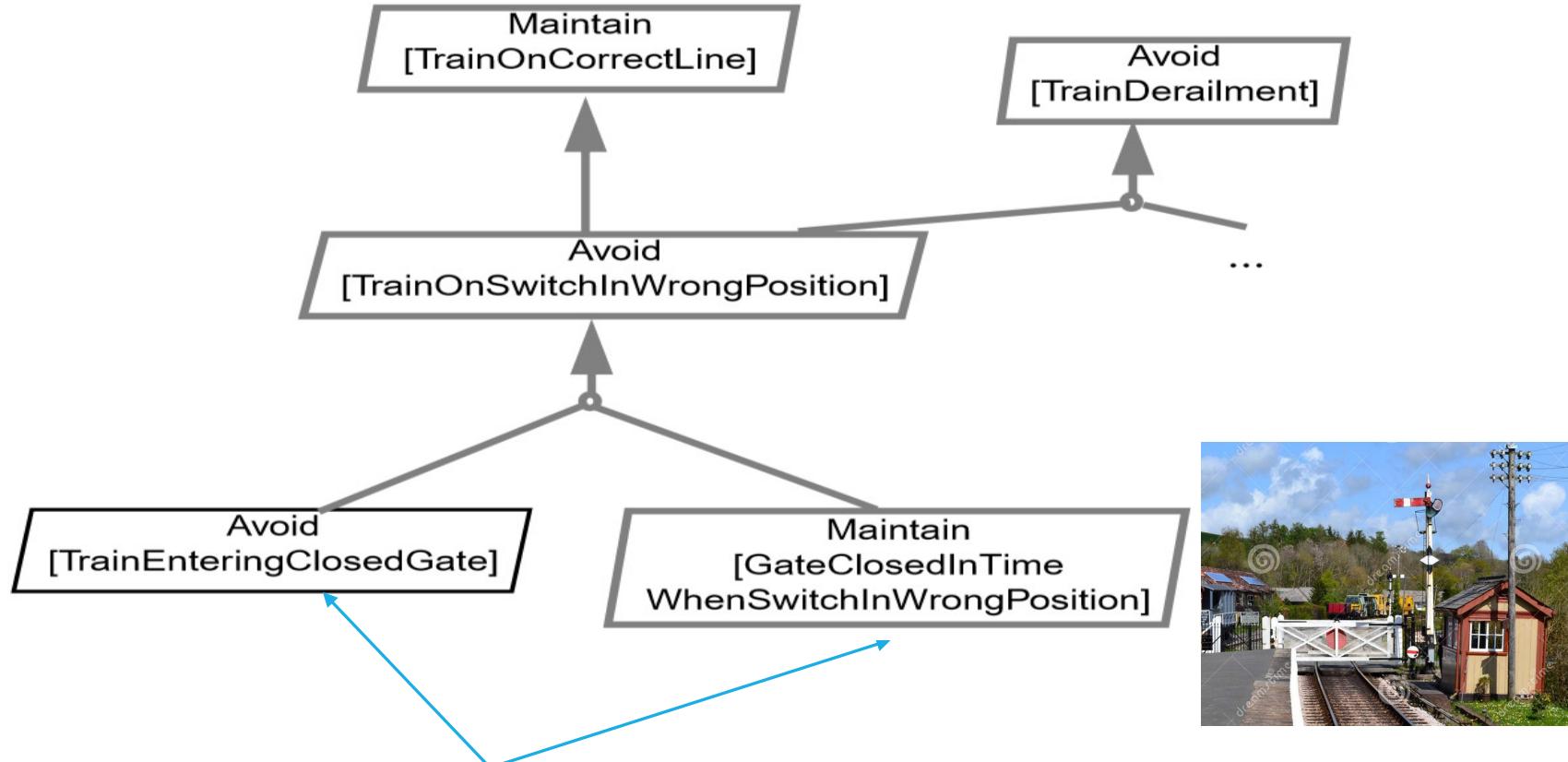


- Obstacles (Possible failures that may negatively impact goal achievement)
  - CommandMsgNotIssued
  - CommandMsgIssuedLate
  - CommandMsgSentToWrongTrain

# IDENTIFY DOMAIN HYPOTHESIS

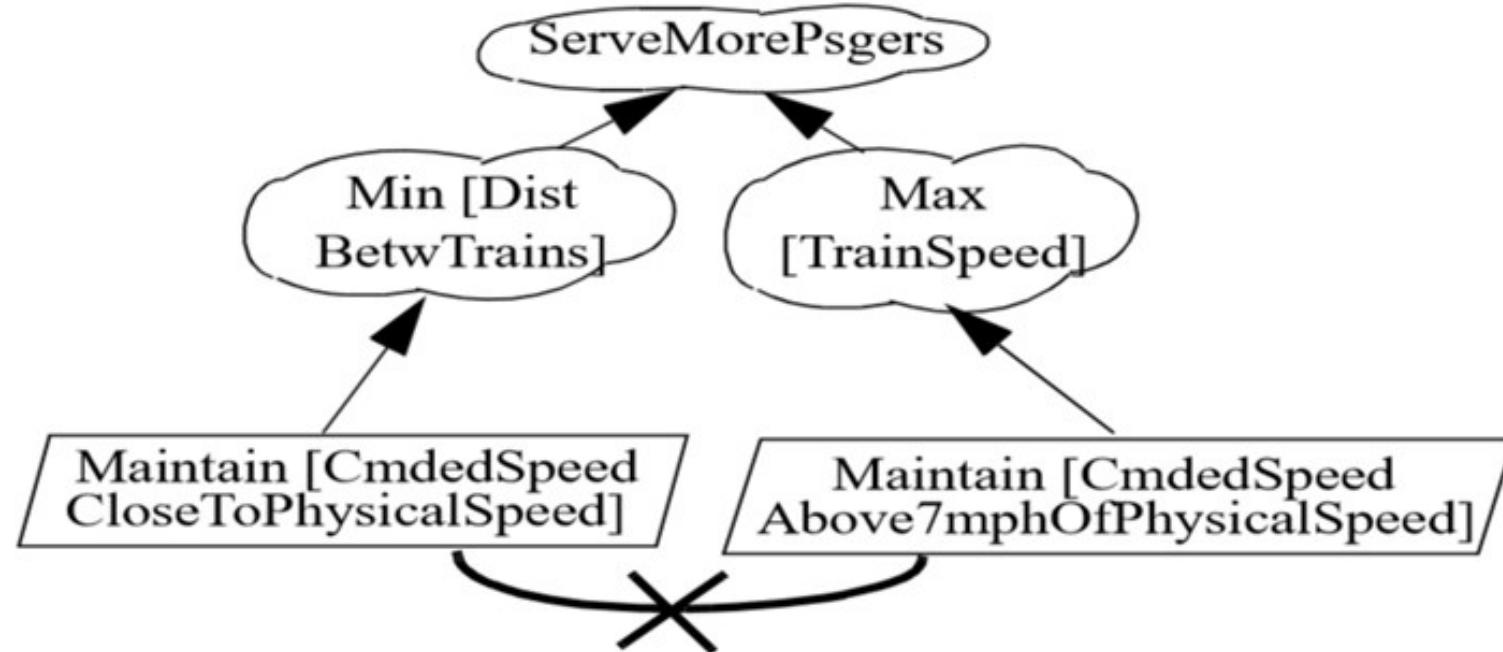
- ✓ **Domain property:** A descriptive statement about the problem world.
- ✓ **Domain invariants:** Should hold invariably, regardless of how the system behaves. This usually corresponds to some physical laws. E.g. *The light is either on or off.*
- ✓ **Domain hypothesis:** To satisfy a service request, an infrastructure/environment to perform the service must be available.

# IDENTIFYING DOMAIN HYPOTHESIS



- ✓ Domain hypothesis related to the goal: *Every track segment leading to a switch is ended with a gate.*

# HANDLING GOAL CONFLICTS



# CAN GORE HELP DERIVE BETTER REQUIREMENTS ?

- ✓ “Requirements definition must say why a system is needed, based on current or foreseen conditions, which may be internal operations or an external market. It must say what system features will serve and satisfy this context. And it must say how the system is to be constructed. [1]”

Goal models

Agent models

Responsibility  
models

Operation  
models

Domain  
hypotheses

# SUMMARY (CONT.)

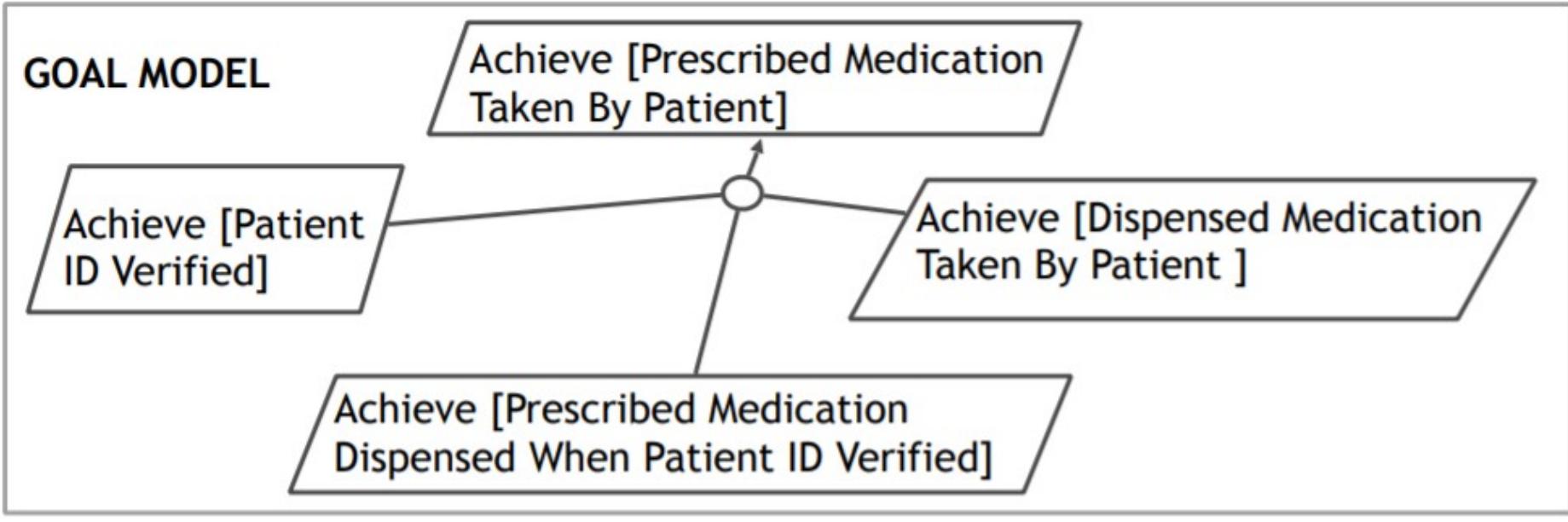
- ✓ Requirements elicitation and analysis = iterative process
  - requirements discovery, classification and organization, negotiation and requirements documentation.
- ✓ Techniques for requirements elicitation
  - interviews, scenarios, use-cases and ethnography, etc.
- ✓ Requirements validation = checking the requirements
  - for validity, consistency, completeness, realism and verifiability.
- ✓ Business, organizational and technical changes inevitably
  - => changes to the requirements for a software system.
- ✓ Requirements management = managing and controlling the requirement changes.

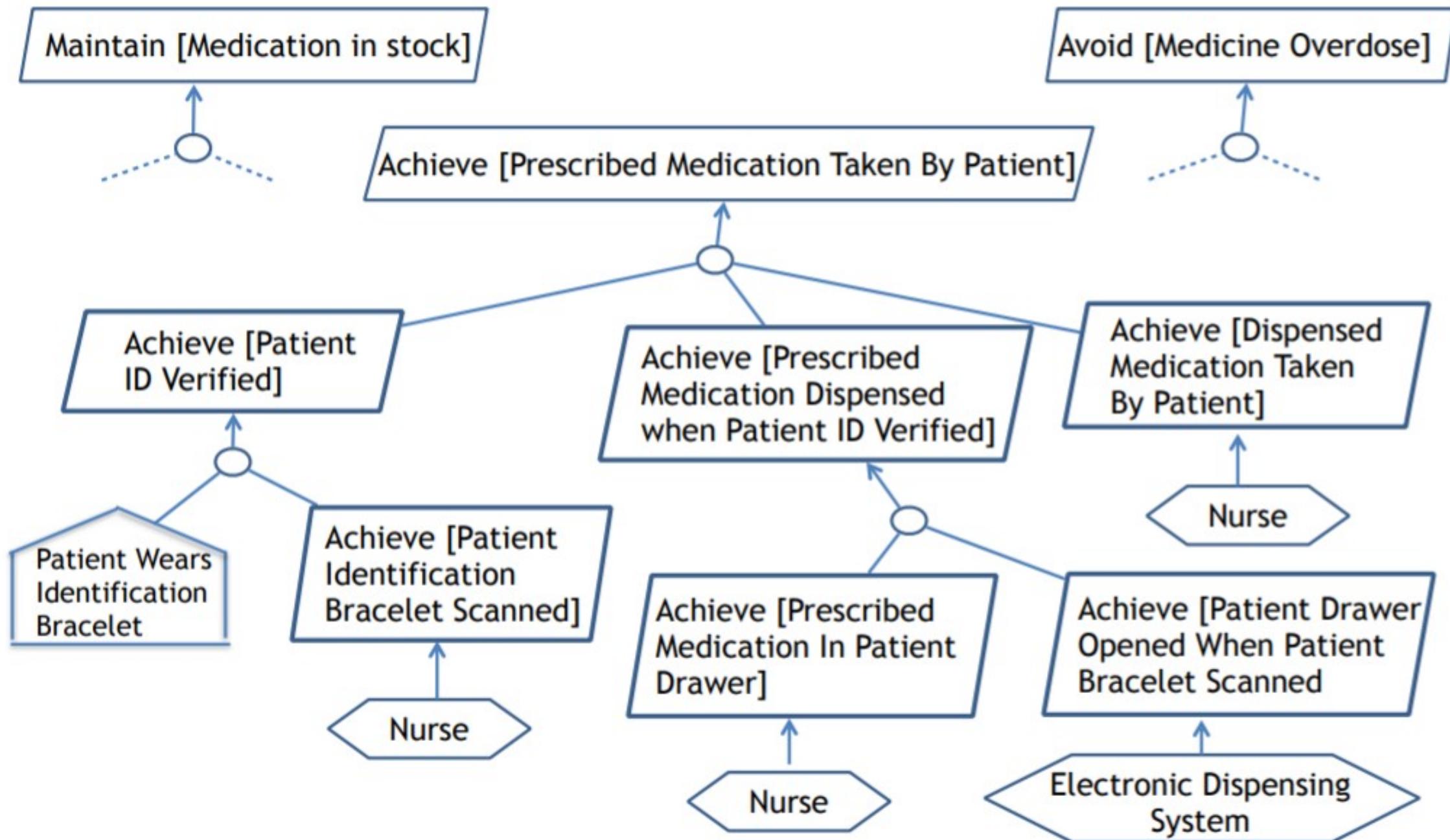
## GOAL MODEL

Achieve [Prescribed Medication Taken By Patient]

- ✓ Electronic Healthcare Record (EHR) system is a repository of patient data in digital form, stored and exchanged securely, and accessible by multiple authorised users. As a normal flow, medication will be first prescribed by authorized nurses to a patient, after that the medication is dispensed and then taken by the patient. EHR ServerRX deployed in a London hospital, required all the patients to wear a bracelet ID for identification purposes. Nurses had to scan this bracelet before dispensing any medicine in order to reduce the risk of administering the medication to the wrong patient.
- ✓ Describe at least three sub goals to the goals
- ✓ <https://app.diagrams.net/#G1pL-o8b58wxjlz1BakfxWQo-B6EWvYL8W>

# EXERCISE





# QUALITY OF REQUIREMENTS



*"The client kept changing the requirements on a daily basis, so we decided to freeze them until the next release."*

# EXAMPLE OF NIGHTMARE REQUIREMENTS

- ✓ The system shall perform at the maximum rating at all times except that in emergencies it shall be capable of providing up to 125% rating unless the emergency condition continues for more than 15 minutes in which case the rating shall be reduced to 105% but in the event that only 95% can be achieved then the system shall activate a reduced rating exception and shall maintain the rating within 10% of the stated values for a minimum of 30 minutes.

More than one requirement in a paragraph

Rambling: like a novel

Vague terms: emergencies

# ANOTHER EXAMPLE OF NIGHTMARE REQUIREMENTS

- ✓ The system shall provide general word processing facilities which shall be easy to use by untrained staff and shall run on a thin Ethernet Local Area Network wired into the overhead ducting with integrated interface cards housed in each system together with additional memory if that should be necessary.

More than one requirement in a paragraph

Rambling: like a novel

Let-out clauses: if that should be necessary

Vague words: be easy to use, additional memory

# CHARACTERISTICS OF GOOD SOFTWARE REQUIREMENTS SPECIFICATION\*

- Complete
- Unambiguous
- Consistent
- Correct
- ✓ Verifiable
- ✓ Traceable
- ✓ Ranked for importance and/or stability
- ✓ Modifiable

\* IEEE Std 830-1998

# COMPLETENESS

- ✓ Definition of the responses of the software to **all** realizable classes of **input data** in **all** realizable classes of **situations** (IEEE Std).
- ✓ All possible situations must be covered

“If X then....”, “If Y then....” Must also consider what will happen “If neither X nor Y...”

# COMPLETENESS (CONT')

## ✓ Automatic door opener

If the door is closed and a person is detected, then send signal Open\_Door.

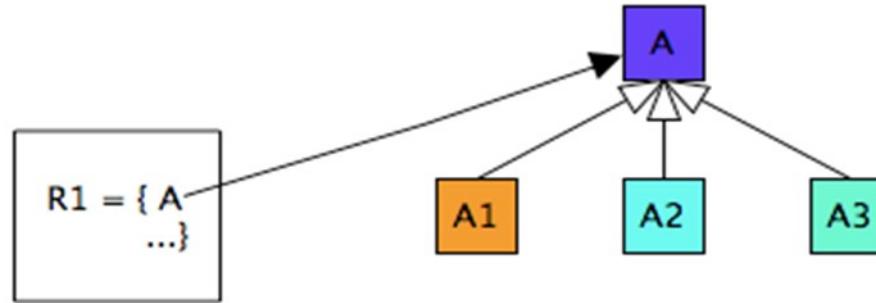
If no person is detected after 10 sec., send signal Close\_Door.

## ✓ What are missing?



# UNAMBIGUOUS

- ✓ A software requirement is unambiguous, if and only if, every requirement stated therein has only one interpretation (IEEE Std).
  - Ambiguity: Requirement with terms or statements that can be interpreted in different ways.



# UNAMBIGUOUS (CONT’)

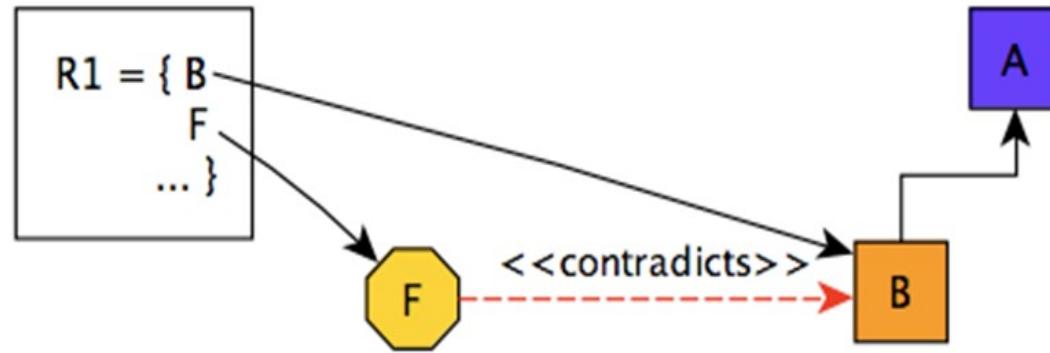
- ✓ Requirement of autonomous cruise control (ACC) system is:

*The maximum speed of a vehicle in a busy road shall be 10 mph.*

Ambiguity: What type of vehicle do you mean?  
What is busy road?

# CONSISTENT

- ✓ A software requirement is internally consistent if, and only if, no subset of individual requirements described in it conflict.



# CONSISTENT (CONT')

- Two requirements of the PROFIBUS network are in conflict
  - The PROFIBUS shall have a short reaction time of 60ms
  - PROFIBUS used in hazardous area shall have a low power dissipation of 3.6W

Conflict: **Fast** PROFIBUS has to have a **high** power

# MAIN TYPES OF CONFLICTS

- ✓ Three main types of conflicts in software requirements
  - Specific characteristics of real-world objects
  - The logical or temporal conflict between two actions
  - Different terms for describing the same real-world object
- Consistency is a challenge since we, at least in the general case, need a complete overview of all requirements that are related to the same event, function, or parameter.

# CORRECT

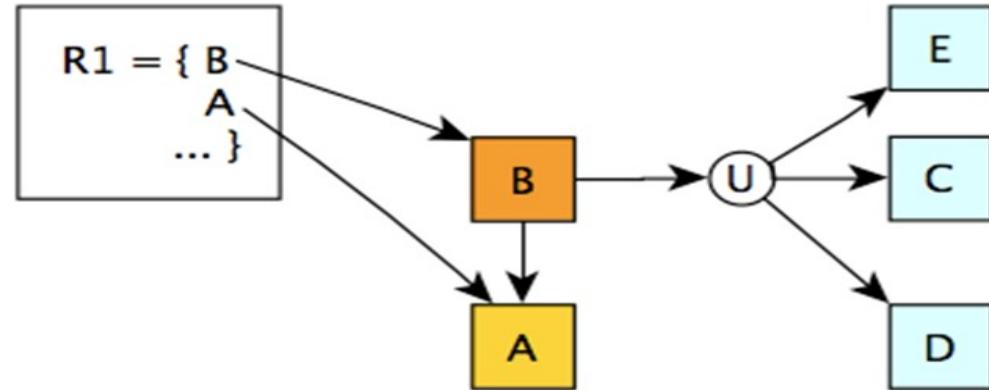
- ✓ Possible incorrect requirements
  - I Forward referencing
  - I Opacity
  - I Noise
  - I Etc.

# REQUIREMENTS IMPRECISION

- ✓ Problems arise when requirements are not precisely stated.
  - Ambiguous requirements may be interpreted in different ways by developers and users.
  
- ✓ For the term ‘search’ in requirement 1
  - User intention – search for a patient name across all appointments in all clinics;
  - Developer interpretation – search for a patient name in an individual clinic. User chooses clinic then search.

# FORWARD REFERENCING

- ✓ Requirement items that make use of problem world domain features that are not yet defined.



E, C, and D need to be mapped to a requirement item

# FORWARD REFERENCING (CONT')

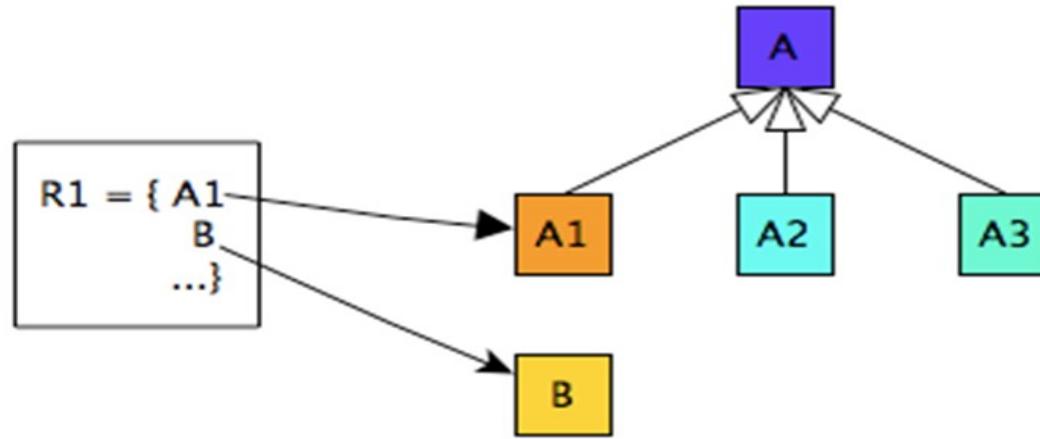
- ✓ A requirement of autonomous cruise control (ACC) system is:

ACC system shall maintain the preset speed of an ego-vehicle if there is no forward vehicle.

- Missing inference
  - Who sets the preset speed?
  - What is the value of the preset speed?

# OPACITY

- ✓ Requirement items for which rational or dependencies are invisible.



Multiple unrelated concept mapping. A is not related to B

# OPACITY (CONT')

## ✓ Requirement:

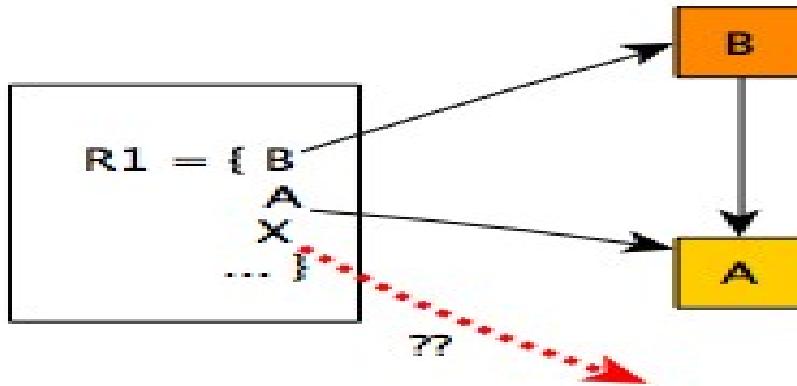
Each time the freight train (Freight trains are trains used for carrying goods) doors are closed, the passengers must all seated.

- Opacity
  - There is no visible relationship between the freight trains and passengers.

# NOISE

✓ Requirement items that yield no information on problem world features.

X refers to a concept undefined in the domain



# NOISE (CONT')

- ✓ Requirement: The train system shall guarantee safe transportation of all passengers on their residence.
- Noise
  - A residence is an unknown concept within the train domain
  - The train can only transport passengers to the train station and not to their residence

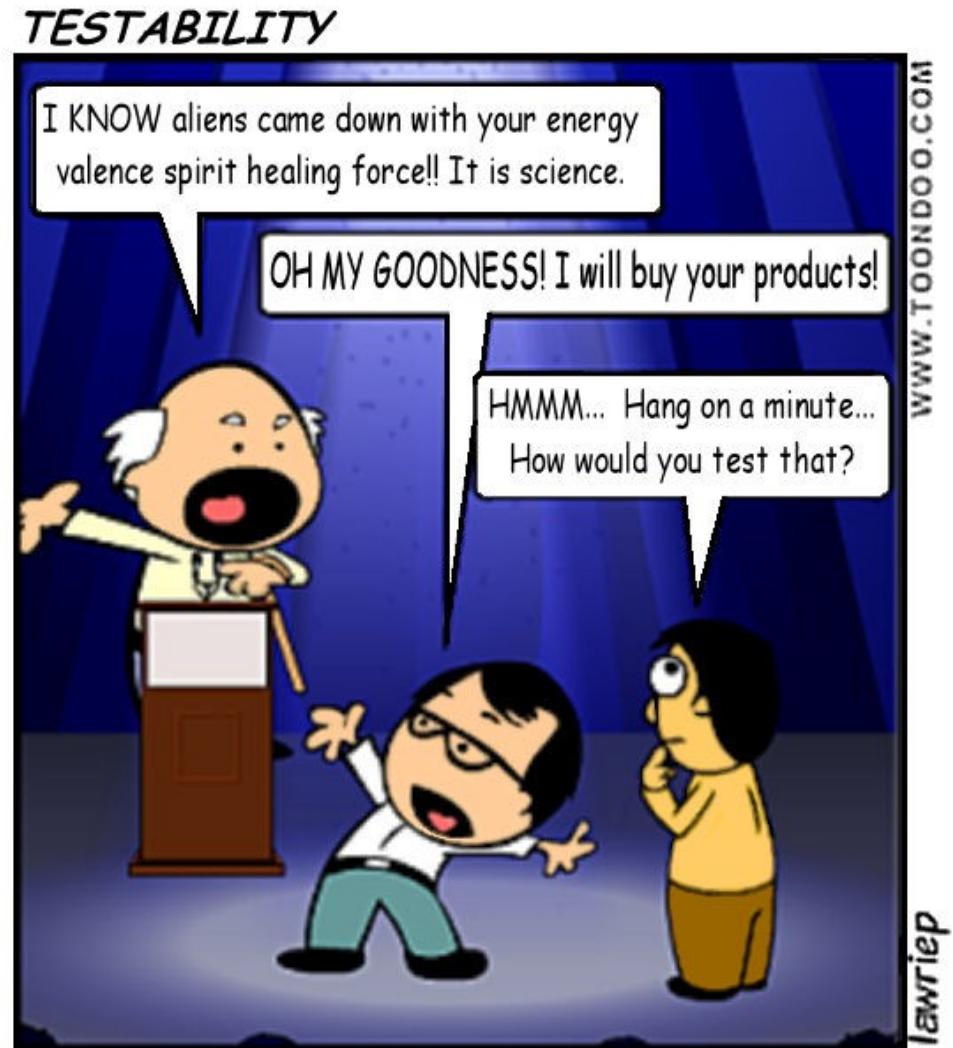
# CHARACTERISTICS OF GOOD SOFTWARE REQUIREMENTS SPECIFICATION\*

- ✓ Complete
- ✓ Unambiguous
- ✓ Consistent
- ✓ Correct
- **Verifiable**
- ✓ Traceable
- ✓ Ranked for importance and/or stability
- ✓ Modifiable

\* IEEE Std 830-1998

# TESTABLE REQUIREMENT

- ✓ If a method cannot be devised to determine whether the software meets a particular requirement, then that requirement is not testable.



# TESTABILITY CONCERNS

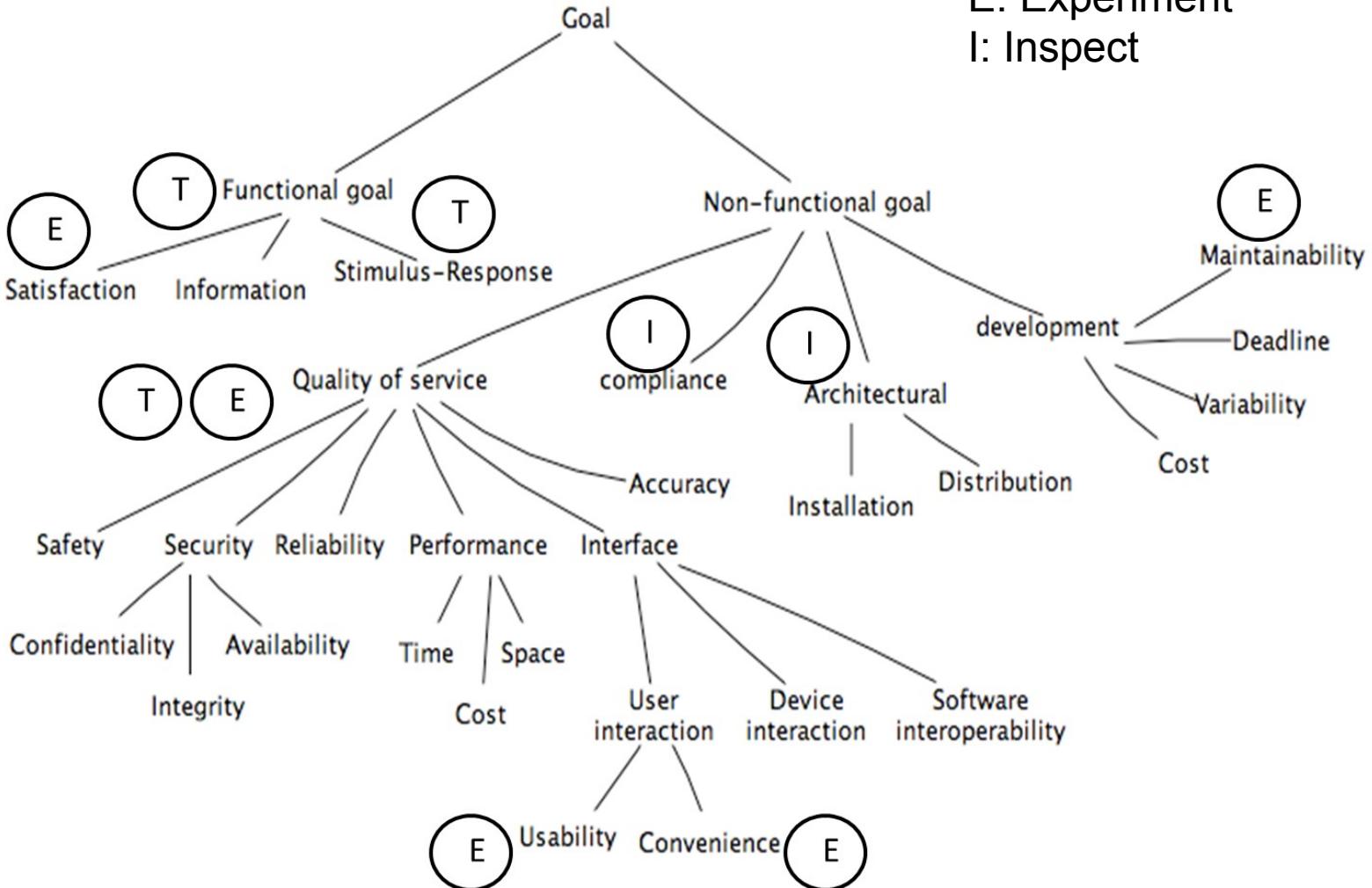
- ✓ Testability touches upon two areas of concern
  - How test-friendly is the requirement?
  - How easy is it to test the implementation?
  
- ✓ These two concerns are not independent and need to be considered together

# THREE WAYS OF CHECKING GOAL ACHIEVEMENT

- ✓ Executing a test
  - Give input
  - Observe and check the output
  - Involves the computer system and peripherals
- ✓ Run experiments
  - Input / output but also involves the users
- ✓ Inspect the code and other artifacts
  - Evaluation based on documents

# WHEN TO USE WHAT

T: Run test on code  
E: Experiment  
I: Inspect



# TO MAKE REQUIREMENTS TESTABLE

- ✓ A requirement needs to be stated in a precise way
- ✓ For some requirements, this is in place right from the start

***When the ACC system is turned on, the “Active” light on the dashboard shall be turned on.***

- ✓ For some other requirements, we need to change it to get a testable version

***The system shall be easy to use.***

# REQUIREMENT TESTABILITY CHECKLIST

- ✓ Modifying phrases
- ✓ Vague words
- ✓ Pronouns with no reference
- ✓ Passive voice
- ✓ Negative requirements
- ✓ Assumptions and comparisons
- ✓ Indefinite pronouns
- ✓ Boundary values
- ✓ ...

# MODIFYING PHRASES

- ✓ The meaning of the following words and phrases are subject to interpretation and make requirement optional:
  - as appropriate
  - if practical
  - as required
  - to the extent necessary / practical
- ✓ “Shall be considered” only requires the contractor to think about it
- ✓ “And/or” mandatory/optional?
- ✓ “Shall/should/may”, should not be used for priority

# VAGUE WORDS

- ✓ Vague words inject confusion, e.g.,
  - Manage, handle, process, necessary, appropriate
  - Usually, approximately, systematically, quickly
- ✓ Use words that express what the system must do
- ✓ **Requirement:** The system shall **process** ABC data to the extent **necessary** to store it in an **appropriate** form for future access.
- ✓ **Correction:** The system shall edit ABC data.

# PRONOUNS WITH NO REFERENCE

- ✓ **Requirement:** **It** shall be displayed.
- ✓ When this occurs, the writer is usually relying on a nearby requirement in the requirements document for the meaning of "**it**."
- ✓ As requirements are assigned for implementation, they are often reordered and regrouped, and the defining requirement is no longer nearby.

# PASSIVE VOICE

- ✓ Requirements should be written in an active voice, which clearly shows X does or provides Y
- ✓ **Requirement with passive voice:** Z shall be calculated
- ✓ **Correction by using active voice:** The cruise control system shall calculate Z.

# NEGATIVE REQUIREMENTS

- ✓ *Everything outside* the system is what the system does **not** do
- ✓ Testing would have to continue forever to prove that the system does not do something
- ✓ State what the system does

**Negative requirement:** Dangerous characters should not be used (blacklist).

**Correction:** Only characters A, B, C are allowed to be used (whitelist).



# ASSUMPTIONS AND COMPARISONS

- ✓ **Requirement:** the system shall increase throughput by 15%.
- ✓ Sounds testable, but isn't
- ✓ The **assumption** is "over current system throughput"
- ✓ By comparing to another system, the meaning of the requirement changes when the other system changes

# INDEFINITE PRONOUNS

- ✓ Indefinite pronouns are “stand-in” for **unnamed** people or things
- ✓ Their meaning is **subject to interpretation**

- All
- Another
- Any
- Anybody
- Anything
- Each
- Either
- Every
- Everybody
- Everyone
- Everything
- Few
- Many
- Most
- Much

# BOUNDARY VALUES

- ✓ Vacation requests of up to 5 days do not require approval. Vacation requests of 5 to 10 days require supervisor approval. Vacation requests of 10 days or longer require management approval.

# HOW EASY IS IT TO TEST THE IMPLEMENTATION?

- ✓ Some requirements are more difficult to test than others
- ✓ The volume of tests needed, e.g.,
  - Response time or storage capacity
- ✓ Type of event to be tested, e.g.,
  - Error handling or safety mechanisms
- ✓ The required state of the system before testing, e.g.,
  - A rare failure state or a specific transaction history

# HOW EASY IS IT TO TEST THE IMPLEMENTATION? (CONT')

- ✓ Four concerns related to implementation and testability
  - Autonomy of the system under test
  - Observability of the testing progress
  - Re-test efficiency
  - Test restartability

# AUTONOMY OF THE SYSTEM UNDER TEST

- ✓ How many other systems are needed to test this requirement?
- ✓ Requirement: “If the door is closed and a person is detected then send signal Open\_Door”
- ✓ Can be tested using a simulated actuator
  - Simulate a “person detected” signal on the sensor and check if an Open\_Door signal is sent to the actuator
- ✓ More difficult/costly to be tested if the actual sensors and actuators must be installed for running tests.

# OBSERVABILITY

- ✓ How easy is it to observe the progress of the test execution?

This is important for tests that do not produce output – e.g., the requirement is only concerned with an internal state change or update of a database.

- ✓ How easy to observe results of the test?

Important for tests where the output is dependent on an internal state or database content.

# RE-TEST EFFICIENCY

- ✓ How easy is it to perform the “test – check – change – re-test” cycle?
- ✓ This includes
  - Observe the test result and run only failed tests
  - Run only tests that are related to the change

# TEST RESTARTABILITY

- ✓ How easy is it to
  - Stop the test temporarily
  - Study current state and output
  - Start the test again from the point where it was stopped

# INVOLVE TESTERS IN REQUIREMENT PHASE

- ✓ That a requirement is testable does not necessarily mean that it is **easy** to test
- ✓ In order to have testable requirements it is important that

*The testers are involved right from the start of the project. It is difficult to add testability later.*

- ✓ The tests are integrated parts of the requirement

# NON-FUNCTIONAL REQUIREMENTS



*"The client kept changing the requirements on a daily basis, so we decided to freeze them until the next release."*

# WHAT ARE NON-FUNCTIONAL REQUIREMENTS (NFR)

- ✓ Many definitions can be found of NFR
- ✓ A NFR is basically an **attribute of** or a **constraint on** a system <sup>[1]</sup>

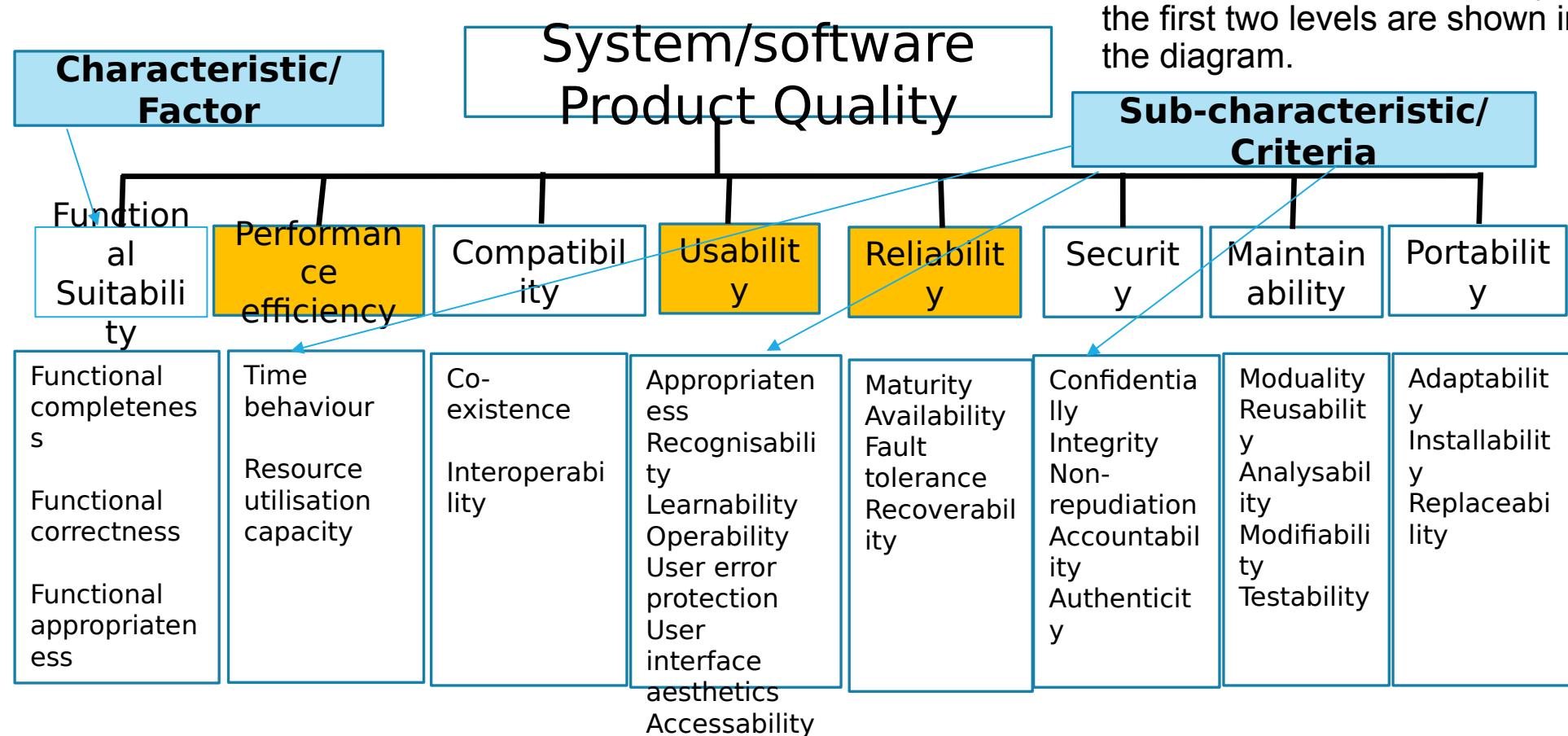
- Attributes

- **-ilities:** understandability, reliability, portability, flexibility, availability, maintainability, scalability, ...
    - **-ties:** security, simplicity, clarity, ubiquity, integrity, safety, modularity, ...
    - **-ness:** user-friendliness, robustness, timeliness, responsiveness, ...

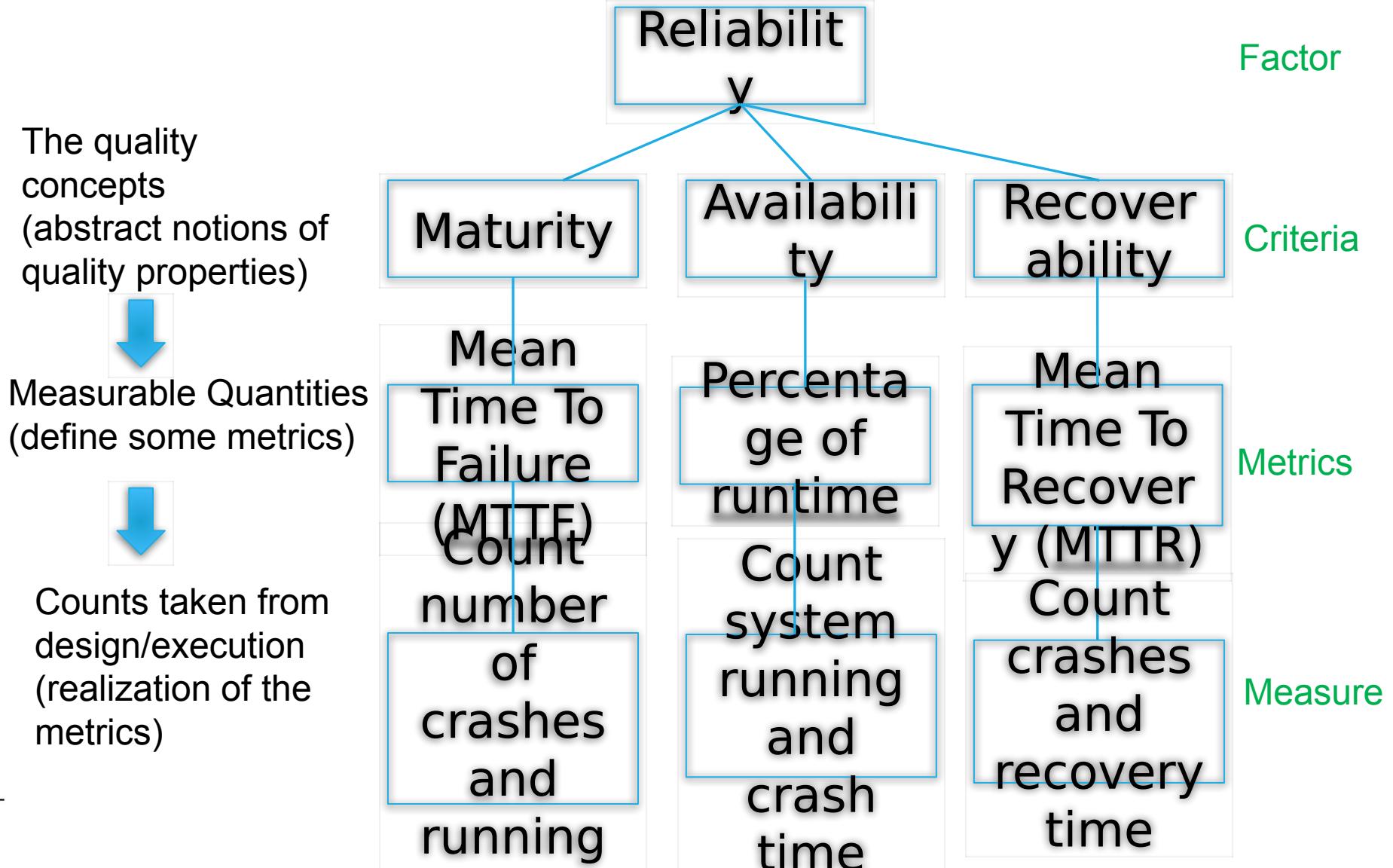
- Constraint

- **Physical**
    - **Environmental**
    - **Legal**

# ISO/IEC 25010 PRODUCT QUALITY MODEL



# MAKE RELIABILITY MEASURABLE



# RELIABILITY REQUIREMENTS SPECIFICATION

## □ **Maturity:**

MTTF (Mean Time To Failure, the expected time to failure for a non-repairable system) > 500 hrs

## ✓ **Availability:**

The system shall meet or exceed 99.99% uptime

## ✓ **Recoverability:**

In case of an error, the time needed to get the system up and running again (MTTR) shall not exceed one hour

# RELIABILITY TESTS

- ✓ MTTF calculation
  - Assume you tested 3 identical systems starting from time 0 until all of them failed
  - The first system failed at 10 hours, the second failed at 12 hours and the third failed at 13 hours
  - The MTTF is the average of the three failure times, which is  $(10+12+13)/3 = 11.6667$  hours
- How to verify MTTF > 500 hrs?
  - Use 10 PCs. Test for two weeks (40 hr per week) => TTT (Total Test Time) = 800
  - Failed more than once? Satisfied/Not satisfied

# TOTAL TEST TIME IN LAB VS. USAGE TIME

- ✓ We need to distinguish
  - Total test time (TTT) in the lab
  - Usage time (UT) of the system in real life
  
- ✓ The usage time is related to
  - How often is the system used at the users' site?
  - Number of users each time the system is used

# TOTAL TEST TIME IN LAB VS. USAGE TIME (CONT')

A simple example:

- ✓ We have TTT = 800 hrs in the lab.
  - Use 10 PCs. Test for two weeks (40 hrs per week)
- ✓ The system will be used for only 20 hrs per week - e.g., for accounting purposes
- ✓ The 800hrs in the lab. TTT is equivalent to 1600 hrs of usage time
- ✓ Failed twice? Satisfy/Not satisfy MTTF > 500 hrs?

# EXERCISE : AVAILABILITY REQUIREMENTS

- ✓ If the system shall meet or exceed the following availability, what is the maximum downtime per year to satisfy such availability requirements?

maximum downtime = (total time per year) x (1-availability)

Availability requirement	Maximum downtime per year?
99%	– 3.65 days/year
99.9%	– 8.76 hours/year
99.999%	– 5 minutes/year
99.9999%	– 31 seconds/ year

# PERFORMANCE

Response speed and amount of resources used

- Software products
- Hardware facilities
- Materials (e.g., print paper, diskettes)

## Criteria

- **Time behavior:** Response and processing times and throughput rates
- **Resource utilization:** Amount and types of resources used
- **Capability:** Minimum/maximum limits

# PERFORMANCE REQUIREMENT EXAMPLES

- ✓ Authorization of an ATM withdrawal request shall take no more than **2.0 seconds**. Response time
- ✓ Webpages shall fully download in an average of 3 seconds or less over **30 megabits/second Internet connection**. Resource utilization
- ✓ The on-line banking systems should support **minimum 10.000 credit card transactions** proceed per second. Capability

# USABILITY

Help users achieve specified goals with

- Effectiveness
- Efficiency
- Satisfaction

## Criteria

- **User interface aesthetics**
- **Easy to operate**
- **Easy to learn**

Usability criteria are subjective

Tests should have a strong component of subjectivism

# SETTING REQUIREMENTS – THE MBO METHOD

- ✓ The method used in the example is based on T. Gilb's ideas of MbO – Management by Objectives [5]
- ✓ We start with the requirement – e.g., the system shall be **easy to learn**
- ✓ We then follow up with “what do you mean by...” until we reach something that is **observable** and thus **testable**

# REQUIREMENTS – FIRST STEP

- ✓ The first step is to apply the MbO for each criterion
- ✓ Examples
  - I Learn its application  
What do we mean by “**learn application.**”
  - I Is the software product liked by the user  
What do you mean by “**beautiful / pleasing / like**”?

# REQUIREMENTS – SECOND STEP

- ✓ Learn application: After a two weeks course and after two weeks use of the system, a normal user can solve a set of standardized problems
- ✓ Pleasing/like: Score high on a likeability scale - e.g., 90 % score 7 or higher on a scale from 1 to 10 - after a two weeks course and two weeks of real use

We need to develop a course, a set of standardized problems, and a likeability questionnaire.

# USER INVOLVEMENT IN SETTING REQUIREMENTS

When we use MbO or other related techniques for setting requirements, we will in most cases have a situation where:

- ✓ The user will have to participate in the tests
- ✓ There will be a strong link between requirement and test
- ✓ In many cases, ***passing the test by the user will be the requirement***

# NFR REQUIREMENTS – TO REMEMBER

- ✓ Some NFR requirements are conflict with each other
- ✓ Trade-off of NFR requirements will impact system design and implementation

# CHARACTERISTICS OF GOOD SOFTWARE REQUIREMENTS SPECIFICATION\*

- ✓ Complete
- ✓ Unambiguous
- ✓ Consistent
- ✓ Correct
- ✓ Verifiable
- Ranked for importance and/or stability
- Traceable
- Modifiable

\* IEEE Std 830-1998

# RANKED FOR IMPORTANCE AND/OR STABILITY

- ✓ If, and only if, each requirement in it has an identifier to indicate either the importance or stability of that particular requirement.
- ✓ Rank and distinguish classes of requirements
  - Essential
  - Conditional
  - Optional

# REQUIREMENT TRACEABILITY



*"The client kept changing the requirements on a daily basis, so we decided to freeze them until the next release."*

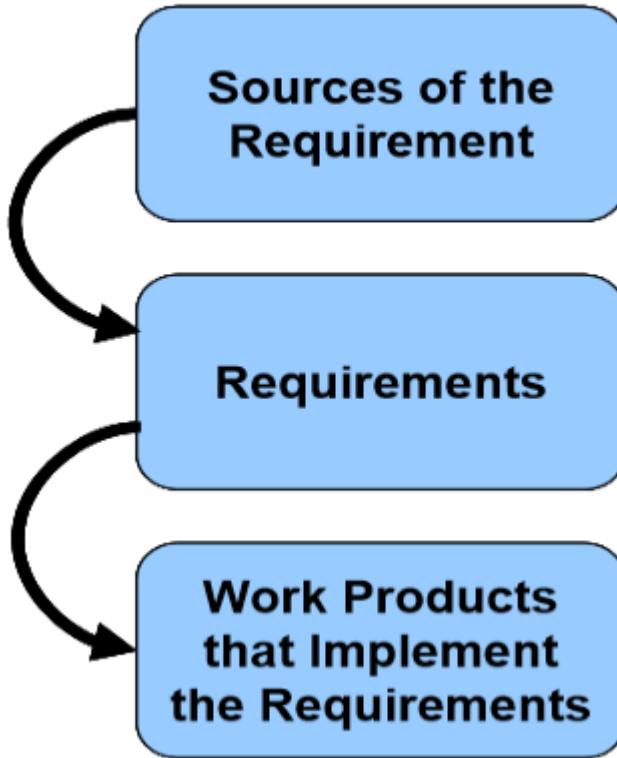
# REQUIREMENTS TRACEABILITY

“Requirements traceability refers to the ability to describe and follow the life of a requirement, **in both a forwards and backwards direction**, i.e., from its origins, through its development and specification, to its subsequent deployment and use, and **through periods of on-going refinement and iteration** in any of these phases.”

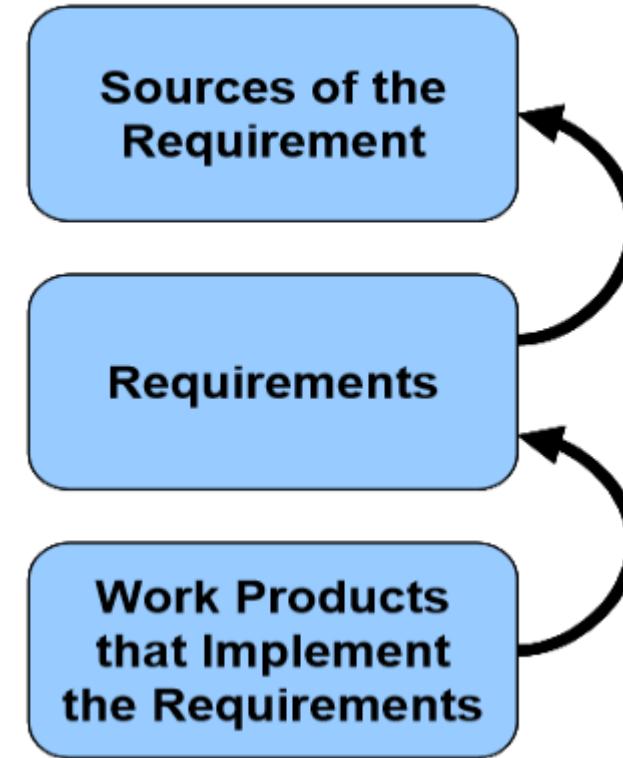
Gotel and Finkelstein

# BIDIRECTIONAL TRACEABILITY

## Forward Traceability



## Backward Traceability



# PURPOSES OF REQUIREMENT TRACEABILITY

- ✓ Forward traceability
  - Ensure all requirements are implemented
  - Change impact analysis
  
- ✓ Backward traceability
  - Avoid “gold plating”
  - Change impact analysis
  - Defect impact analysis
  - Root cause analysis of defects

# HOW TO IMPLEMENT TRACEABILITY?

## ✓ Traceability matrix

**Project Name**

Unique No.	Requirement	Source of Requirement	Software Reqs. Spec / Functional Req. Doc.	Design Spec.	Program Module	Test Case(s)	Successful Test Verification	Modification of Requirement	Remarks

# EXAMPLE OF TRACEABILITY MATRIX

Requirement Source	Product Requirements	HLD Section #	LLD Section #	Code Unit	UTS Case #	STS Case #	User Manual
Business Rule #1	R00120 Credit Card Types	4.1 Parse Mag Strip	4.1.1 Read Card Type	Read_Card_Type.c Read_Card_Type.h	UT 4.1.032 UT 4.1.033 UT 4.1.038 UT 4.1.043	ST 120.020 ST 120.021 ST 120.022	Section 12
			4.1.2 Verify Card Type	Ver_Card_Type.c Ver_Card_Type.h Ver_Card_Types.dat	UT 4.2.012 UT 4.2.013 UT 4.2.016 UT 4.2.031 UT 4.2.045	ST 120.035 ST 120.036 ST 120.037 ST 120.037	Section 12
Use Case #132 step 6	R00230 Read Gas Flow	7.2.2 Gas Flow Meter Interface	7.2.2 Read Gas Flow Indicator	Read_Gas_Flow.c	UT 7.2.043 UT 7.2.044	ST 230.002 ST 230.003	Section 21.1.2
	R00231 Calculate Gas Price	7.3 Calculate Gas price	7.3 Calculate Gas price	Cal_Gas_Price.c	UT 7.3.005 UT 7.3.006 UT 7.3.007	ST 231.001 ST 231.002 ST 231.003	Section 21.1.3

# HOW TO IMPLEMENT TRACEABILITY? (CONT')

- ✓ Trace tagging

**SRS**

**R00104** ← The system shall cancel the transaction at the earliest time prior to the actual dispensing requests cancellation.

**SDS**

SDS Identifier	SRS Tag	Component Name
7.01.032	R00104	Cancel_Transaction

**UTS**

Test Case #	SDS Identifier	Test Description
23476	7.01.032	Cancel_Before_Dispatch
23477	7.01.032	Cancel_After_Dispatch
23478	7.01.032	Cancel_After_Dispatch

# CHARACTERISTICS OF GOOD SOFTWARE REQUIREMENTS SPECIFICATION\*

- ✓ Complete
- ✓ Unambiguous
- ✓ Consistent
- ✓ Correct
- ✓ Verifiable
- ✓ Ranked for importance and/or stability
- ✓ Traceable
- Modifiable

\* IEEE Std 830-1998

# MODIFIABLE

- ✓ If, and only if, a requirement structure and style are such that any changes to the requirements can be made easily, completely, and consistently while retaining the structure and style.
- ✓ Redundancy can easily lead to low modifiability
- ✓ Boilerplate can help improve modifiability
  - E.g., “will” -> “shall”

# **Course summary**

Jingyue Li (Bill)

# Exam

## Vurderingsordning: Mappeevaluering

Termin	Statuskode	Vurdering	Vektning	Hjelpermidler	Dato	Tid	Eksamens-system
Vår	ORD	Arbeider	50/100				INSPERA
Vår	ORD	Hjemme-eksamen (1)	50/100		Utlevering 03.06.2022 09:00		INSPERA
					Innlevering 03.06.2022 13:00		
Sommer	UTS	Arbeider	50/100				INSPERA
Sommer	UTS	Hjemme-eksamen	50/100		A: 89–100 points B: 77–88 points C: 65–76 points D: 53–64 points E: 41–52 points F: 0–40 points		INSPERA

**Letter grade A-F**

# Evaluation and grading

- Exercises: 50%
- Exam: 50%
- You have to pass both!
  - You have to hand in **all** the exercises **and** get a passing grade to be allowed to take the exam
  - If you fail the exam, you will fail the course

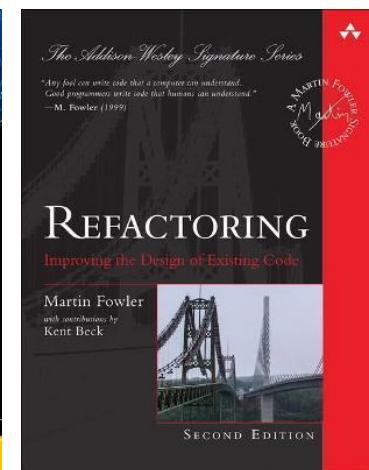
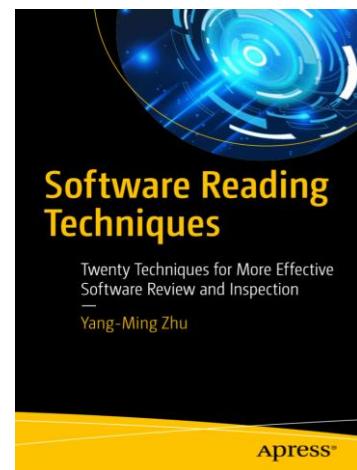
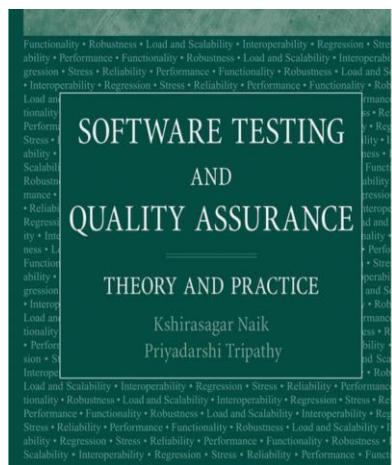
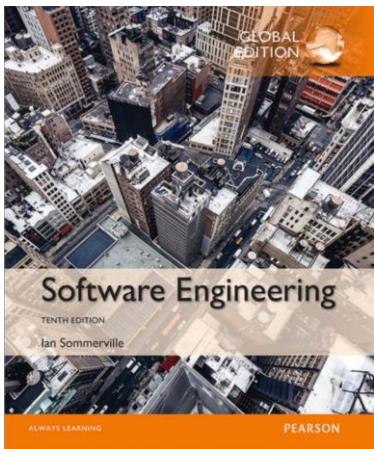
Exercises	Exam	Final grade
Fail	Fail	Fail
Fail	Pass	Fail
Pass	Fail	Fail
Pass	Pass	Combine exercise and exam grades

# Course modules

- Requirement module
- Testing module
- Code review and refactoring
- Chosen topics
  - DevOps
  - Technical debt
  - Cost and effort estimation
  - Defect estimation
  - Software ecosystem
  - Software startup
- Exercises 1 - 3

# Curriculum

- Journal and conference articles (uploaded to blackboard)
- Some chapters from books (accessible in NTNU library)



(1<sup>st</sup>/2<sup>nd</sup> edition)

# Requirement module

Description	Students should be able to	To read
Requirement elicitation	<ul style="list-style-type: none"><li>• <b>Apply</b> goal oriented RE approach</li></ul>	<ul style="list-style-type: none"><li>• Ian Sommerville (2016), Software Engineering (10th ed.), ISBN 978- 0133943030, chapter 4</li><li>• Paper 1: Goal-oriented RE A guided tour</li></ul>
Requirement quality	<ul style="list-style-type: none"><li>• <b>Identify</b> quality issues of requirements and fix them</li><li>• <b>Define</b> different function and non-functional requirements</li></ul>	<ul style="list-style-type: none"><li>• Paper 1: IEEE Recommended Practice for Software Requirements Specifications Std 830-1998</li><li>• Paper 2: Glinz, M. (2007). On Non-Functional Requirements. 15th IEEE International Requirements Engineering Conference (RE 2007), 21–26.</li><li>• Paper 3: Ontology-Driven Guidance for Requirements Elicitation</li><li>• Paper 4: Bidirectional requirement traceability</li></ul>

# Testing module - 1

Description	Students should be able to	To read
Control flow and data flow testing	<ul style="list-style-type: none"><li>• Explain code, decision, path, and output coverages</li><li>• Create test cases by using different data flow testing strategies</li><li>• Explain how to use the test coverage information for different purposes</li></ul>	Software testing and quality assurance book, chapter 4 and 5
Domain and function testing	<ul style="list-style-type: none"><li>• Apply domain testing approach to generate test cases of single variable and multiple variables in combination</li><li>• Explain risk-based testing</li></ul>	Software testing and quality assurance book, chapter 6 and 9

# Testing module - 2

Description	Students should be able to	To read
Integration and system testing	<ul style="list-style-type: none"><li>• Explain different approaches for creating integration test cases and their pros and cons</li><li>• Create different system test cases</li><li>• Explain different categories of acceptance test cases</li><li>• Explain different test prioritization approaches</li></ul>	Software testing and quality assurance book, chapter 7, 8, and 14
Regression testing	<ul style="list-style-type: none"><li>• Apply graph-walk safe regression test selection approach</li><li>• Explain firewall regression test selection approach</li><li>• Explain different regression test minimization and prioritization strategies</li></ul>	Paper 1: Regression testing minimization, selection, and prioritization: a survey

# Testing module - 3

Description	Students should be able to	To read
Test in practice	<ul style="list-style-type: none"><li>• Explain concepts and strategies related to test planning and execution.</li><li>• Create a test plan.</li></ul>	Software testing and quality assurance book, chapter 12 and 13

# Code review and refactoring

Description	Students should be able to	To read
Code review and code refactoring	<ul style="list-style-type: none"><li>• Explain why code inspection and testing complement each other</li><li>• Explain unsystematic vs. systematic reading techniques</li><li>• Apply check-list based, defect-based, and perspective-based reading techniques</li><li>• Explain the purpose and steps of code refactoring</li><li>• Apply code refactoring methods to identify bad code smells in code (Python) and refactor them</li></ul>	Software reading techniques, chapter 3 and 4  Refactoring book: Improving the Design of Existing Code, chapter 3 (Bad smells in code)  Paper 1: Modern code review a case study at google  Paper 2: Expectations, Outcomes, and Challenges of Modern Code Review

# Chosen topics - 1

Description	Students should be able to	To read
DevOps	<ul style="list-style-type: none"><li>• Explain the motivation for using DevOps</li><li>• Explain the basic concepts of DevOps</li></ul>	<ul style="list-style-type: none"><li>• Slides only</li></ul>
Technical debt	<ul style="list-style-type: none"><li>• Explain the motivation and approaches to measure technical debt</li></ul>	<ul style="list-style-type: none"><li>• Slides only</li></ul>
Cost&Effort and defect estimation	<ul style="list-style-type: none"><li>• Explain importance and challenges in estimating cost and software defects</li><li>• Describe different estimation techniques, i.e., expert judgement, estimation by analogy, Lines of code, and COCOMO</li><li>• Describe machine learning techniques for software defect predictions</li></ul>	<ul style="list-style-type: none"><li>• Slides only</li></ul>

# Chosen topics - 2

Description	Students should be able to	To read
Software ecosystem	<ul style="list-style-type: none"><li>• Explain definitions, key elements and examples of software ecosystems</li><li>• Describe characteristics of software ecosystems: complexity, productivity, robustness, healthiness and competition</li></ul>	<ul style="list-style-type: none"><li>• Slides only</li></ul>
Software startup	<ul style="list-style-type: none"><li>• Explain definitions, key elements and examples of software startups</li><li>• Describe concept of Technical Debt in Software startups</li><li>• Explain logic of effectuation and causation</li><li>• Describe concept and usages of Minimum Viable Product in Software startups</li></ul>	<ul style="list-style-type: none"><li>• Slides only</li></ul>

# Structure of the exam

- Two case studies (10 points each, 20 points in total)
- Five open-ended questions (3 points each, 15 points in total)
- 15 close-ended (Multiple-choice) questions (1 point each, 15 points in total)

# A case study example



- Company A is going to pay Company B for developing Autonomous Truck Platooning. Truck platooning (as shown in the above picture) is the linking of two or more trucks in convoy, using connectivity technology and automated driving support systems. These vehicles automatically maintain a set, close distance between each other when they are connected for certain parts of a journey, for instance, on motorways.
- The requirements Company B got from Company A are as follows.

*We are going to develop Autonomous Truck Platooning. We want some numbers of trucks can drive autonomously, and other trucks are driven by humans. The autonomous trucks should drive at a default speed. We hope the truck platooning can drive along the route we set in GPS before the journey and follow the human drivers' command to change the route. We hope the autonomous trucks should have three chairs for the drivers. We want the truck to drive safely. So the trucks should have sensors to avoid collisions with obstacles because there are always other vehicles or obstacles in the motorway. When there are obstacles on the road, the trucks should stop. We also want the trucks to drive efficiently, which means that the trucks should minimize the distance between them and maximize the speed of the whole Platooning. The trucks should use sensors to detect the distances between trucks and the speed of the trucks.*

# A case study example (cont')

- Task: Identify quality issues in these requirements according to the requirements quality metrics: *ambiguity*, *inconsistency*, *forward referencing*, and *opacity*. If one requirement has several quality issues, list all of them. Then, try to fix the requirements quality issues of each requirement and write down the improved requirements.
- Example answers
- Ambiguity (1 point): So the trucks should have sensors to avoid collisions with obstacles because there are always other vehicles or obstacles in the motorway (**Not explain obstacles**).  
Your fix: Only need to make a short list of obstacles. It does not need to be a complete list.
- Inconsistency (1 point): E.g., We hope the truck platooning can drive along the route we set in GPS before the journey and follow the human drivers' command to change the route (**Conflict on who can make the control decision**).  
Your fix: Only need to eliminate the inconsistency.

# An open-ended question example

**Question:** Explain what static backward slicing is and how to create backward slicing using data flow information.

**Answers:**

- The purpose of code slicing is to choose only a subset of code that is relevant to a particular variable at a certain point to minimize the code for more cost-efficient testing and debugging.
- Static backward slicing: A slice with respect to a variable  $v$  at a certain point  $p$  in the program is the set of statements that contributes to the value of the variable  $v$  at  $p$ .
- To establish static backward slicing, all Defs and All P-uses of a variable  $v$  before a certain point  $p$  are chosen.

# Summary

- Use the slides as the table of content to read books and papers
- The content of the course, especially the chosen topics, varies each year. Not all questions from previous years are relevant to this year
- The case studies and open-ended questions in previous years' exams can be used for practice
- We will create a discussion channel in Blackboard for Q&A exam related questions

# Where does code go when it is finished?

- A story about DevOps

# About Me

- Platform Engineer at Vipps
  - Develop login-systems
- Very interested in software security
- Twitter: noratomas3
- Linkedin: Nora Tomas



• Wi-Fi Vipps 13:12 100%



Emma Dahl  
Takk 🙌 I dag

Emilia Janson  
⊗ Du avslø Tirsdag

Oscar Hansen  
↓ Du ba om 300 kr Tirsdag

Filin Kriktiansen  
Mandag

Hjem Betalinger Oppgjør Profil

# About Me

- Studied Computer Science at Gløs
- Failed “a few” subjects in my time at NTNU
  - So don’t worry, you will be fine! :)



# After this lecture I hope you will

- Know what a frontend and backend is
- Get a sense of what an API is
- Understand what deployment means and why it is important
- Know what DevOps means
- See (in the demo) what a DevOps engineer does
  - [github.com/NoraTomas/DevOpsDemo](https://github.com/NoraTomas/DevOpsDemo)

# “Hello World”

The screenshot shows the PyCharm IDE interface. The title bar indicates the project is 'BeginnersBook' and the file is 'HelloWorld.py'. The left sidebar shows the project structure with a 'venv' folder and the 'HelloWorld.py' file selected. The main code editor window displays the following Python code:

```
# This prints Hello World on the output screen
print('Hello World')
```

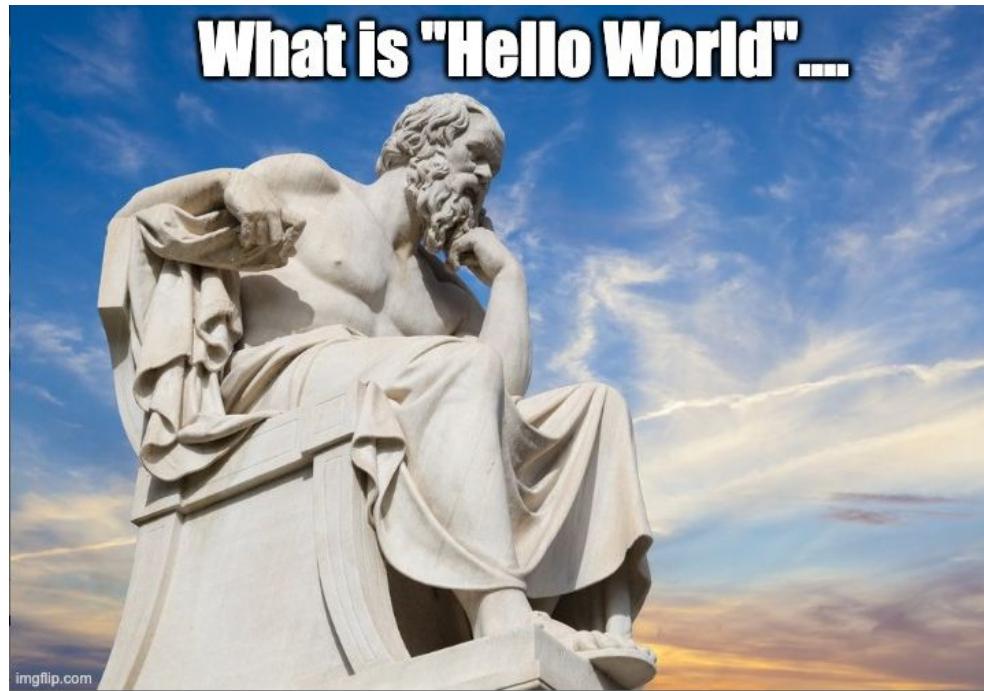
The run tool window at the bottom shows the command used to run the script: '/Users/chaitanyasingh/PycharmProjects/BeginnersBook/venv/bin/python /Users/chaitanyasingh/PycharmProjects/BeginnersBook>Hello World'. The output pane shows the printed text 'Hello World' and the message 'Process finished with exit code 0'. The status bar at the bottom right shows the time as 1:47, encoding as UTF-8, and other system information.

# “Hello World”

- “Coding is not difficult” -> <https://www.youtube.com/watch?v=hb7Q33ysCwl>

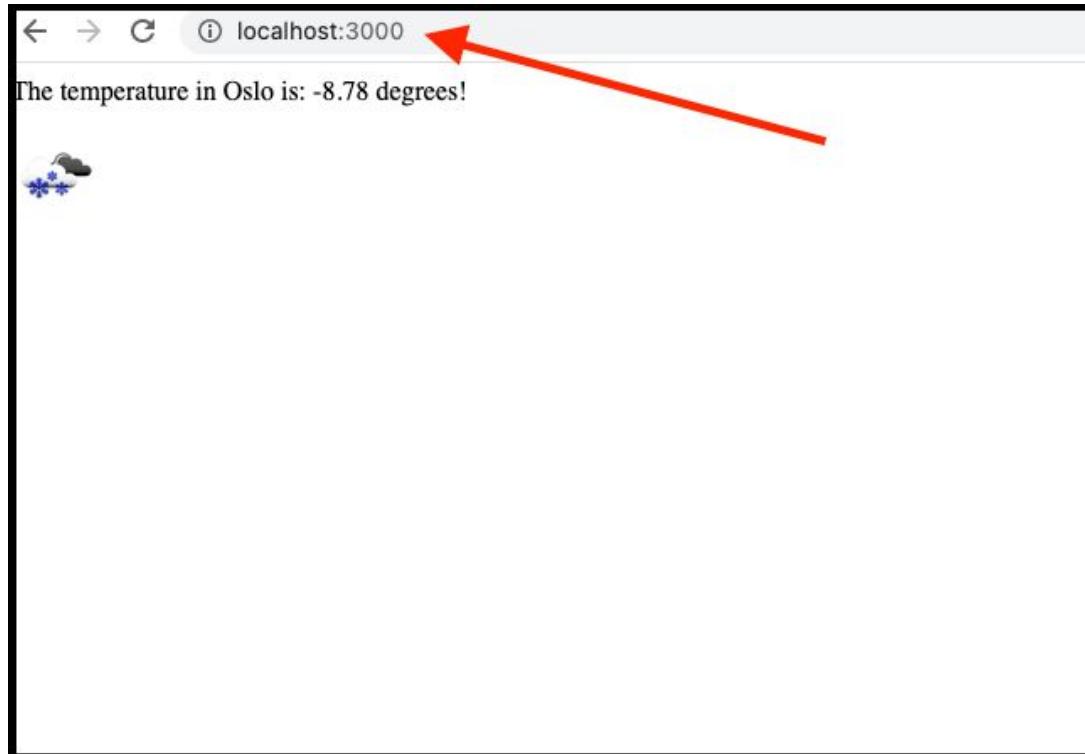


# “Hello World”



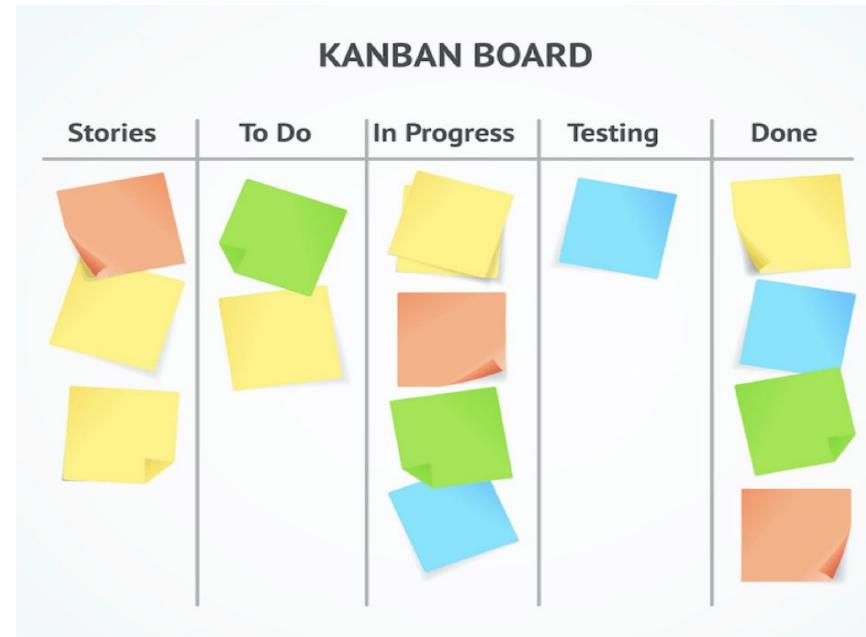
Time for a quick Kahoot!

# Customer-Driven project - Localhost

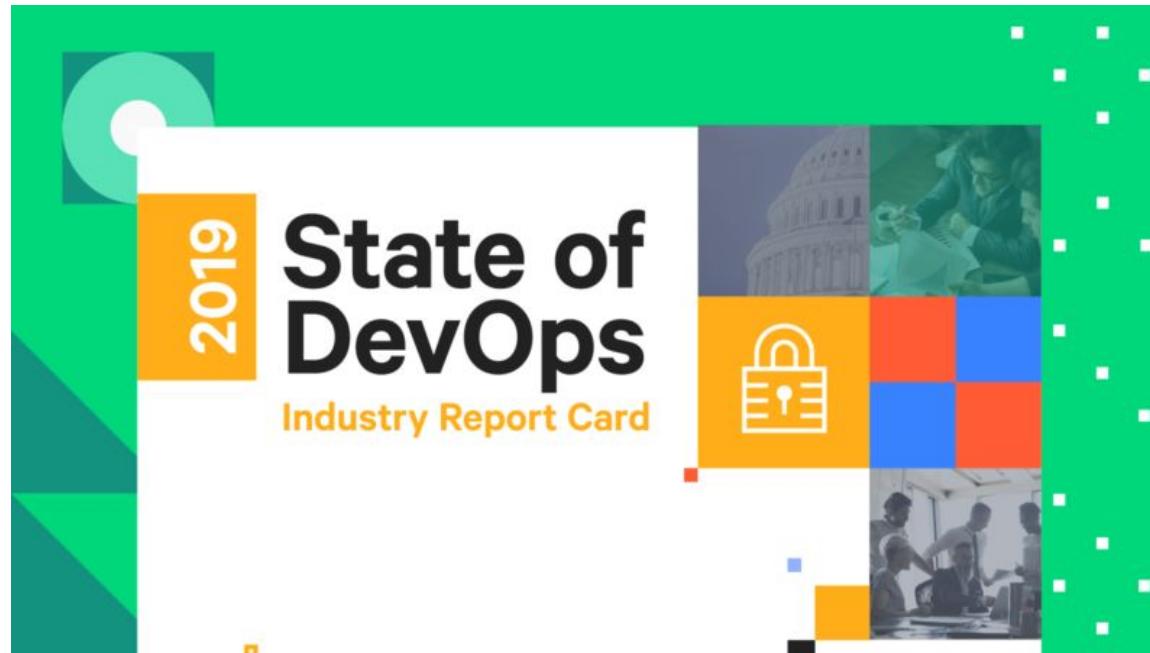


# Introducing Deployment

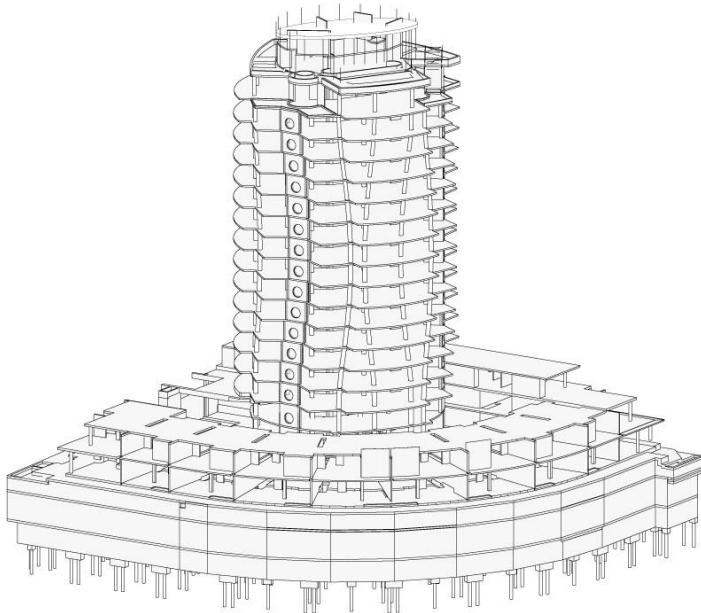
- “Software deployment is all of the activities that make a software system available for use”



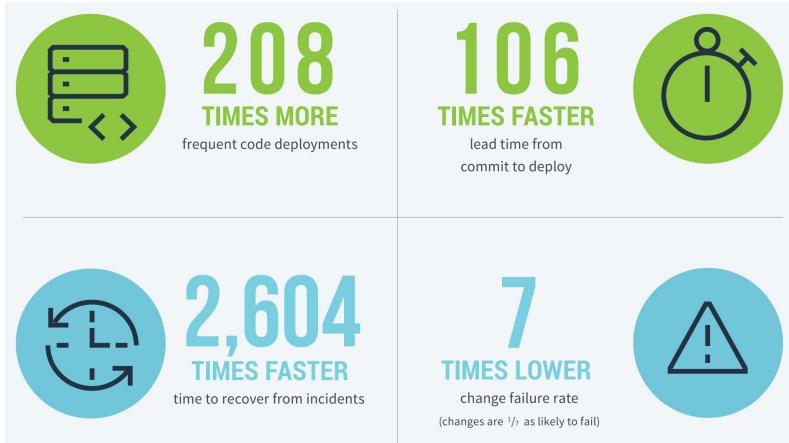
# Failed IT-projects - Forgotten Deployment



# Forgetting deployment = forgetting to build the building

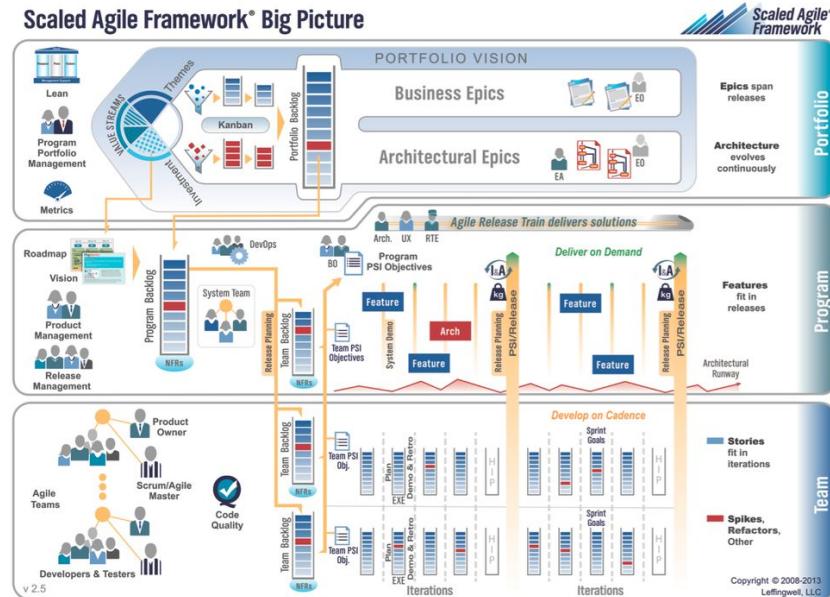


- Source: State of DevOps 2019 report
- Download here: <https://cloud.google.com/devops/state-of-devops/>



# Agile Alignment Trap

- Companies get into the agile alignment trap



# DevOps Engineer

- A lot of job postings for “DevOps Engineer”
- What does it really mean?

[www.finn.no](http://www.finn.no) › Jobb › Ledige stillinger ▾ [Translate this page](#)

## Ledige stillinger med søkeordet "devops" | FINN Jobb - FINN.no

Du finner 291 ledige stillinger med søkeordet "devops" på FINN Jobb. Søk blant alle typer  
jobber i hele Norge!

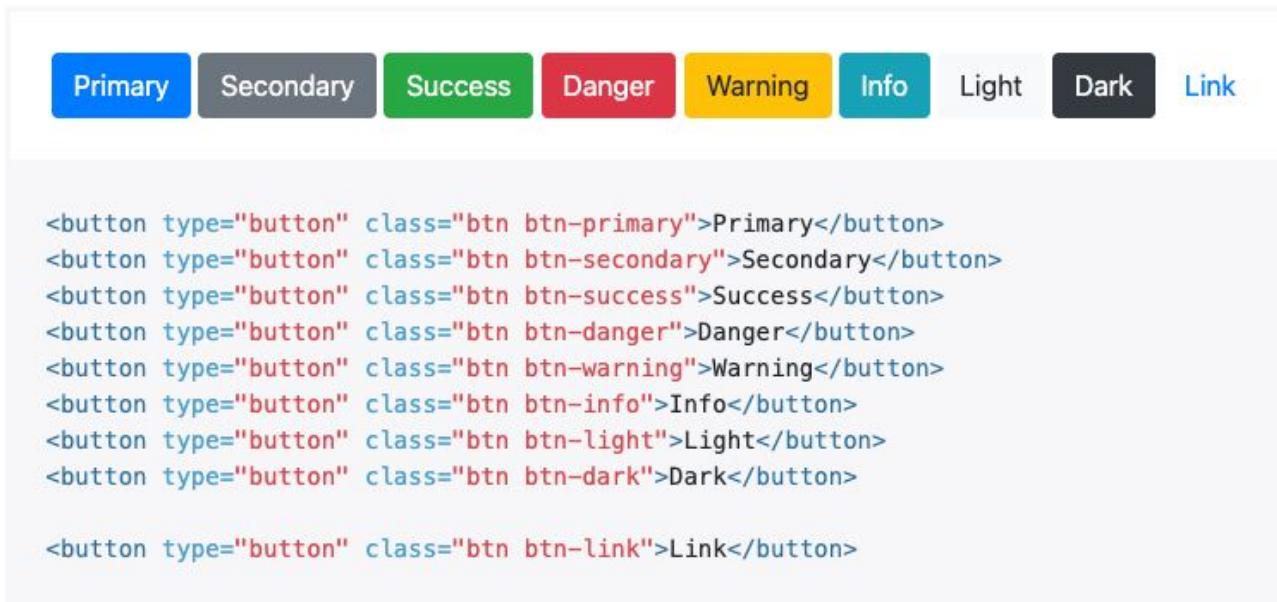
# What is DevOps



To understand what DevOps engineer does...  
Let us create and deploy an API!

# Example: What is an API?

- API is a term used very loosely



The image shows a row of eight colored buttons with labels above them: Primary (blue), Secondary (dark grey), Success (green), Danger (red), Warning (yellow), Info (teal), Light (light grey), Dark (black), and Link (light blue). Below the buttons is a block of HTML code:

```
<button type="button" class="btn btn-primary">Primary</button>
<button type="button" class="btn btn-secondary">Secondary</button>
<button type="button" class="btn btn-success">Success</button>
<button type="button" class="btn btn-danger">Danger</button>
<button type="button" class="btn btn-warning">Warning</button>
<button type="button" class="btn btn-info">Info</button>
<button type="button" class="btn btn-light">Light</button>
<button type="button" class="btn btn-dark">Dark</button>

<button type="button" class="btn btn-link">Link</button>
```

# Example: What is an API?

- Most commonly, when people say API, they mean: HTTP endpoints (URLs) that you can use to retrieve information
- Example: <https://openweathermap.org/current>

API call

---

```
api.openweathermap.org/data/2.5/weather?q={city name}&appid={API key}
```



# Let us make our own Weather API

- We use Java and Spring Boot
- What we will do
  - Show what a backend API looks like
  - Build a JAR file locally
  - Deploy the JAR file



Spring Boot

# Opening our API to the world

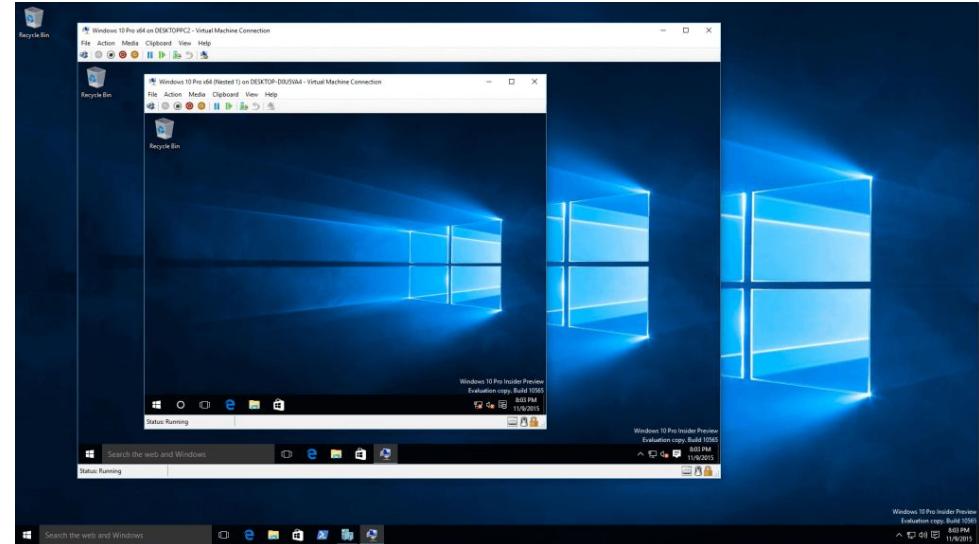
- Could do it from our own computer!
  - Need
    - Public IP
    - Open firewall in router
    - Make an “inbound connection” to port 8080
- Drawbacks from using our own computer
  - Your computer might become slow (with many connections)
  - A lot of other software taking space
  - Exposing your public IP to the world
  - You have to buy a new computer if you want more power
- A server is just a computer



↑  
SERVER :)

# Let us deploy to EC2!

- EC2 is a virtual machine in AWS
- We can use that virtual machine - just like any other computer
- We just don't own the hardware



# We made a backend!

- Why is it a backend?
  - The Java-application is not supposed to run on users computers
  - End-users will not “download” this Jar-file and run it on their computers
  - We know what machine the Jar-file will run on, because we created it!
  - ***Front-end development:*** Making sure things work on different clients

# Our Python application - was a frontend!



The screenshot shows the PyCharm IDE interface. The title bar reads "BeginnersBook [~/PycharmProjects/BeginnersBook] - .../HelloWorld.py [Beg...]" with standard window controls. The left sidebar shows a project structure with a folder named "BeginnersBook" containing a "venv" folder and a file named "HelloWorld.py". The main editor window displays the following code:

```
1 # This prints Hello World on the output screen
2 print('Hello World')
```

The run tool bar at the bottom has a green play button icon and the text "Run HelloWorld". The run output window below shows the command line output:

```
/Users/chaitanyasingh/PycharmProjects/BeginnersBook/venv/bin/python /Us
Hello World
Process finished with exit code 0
```

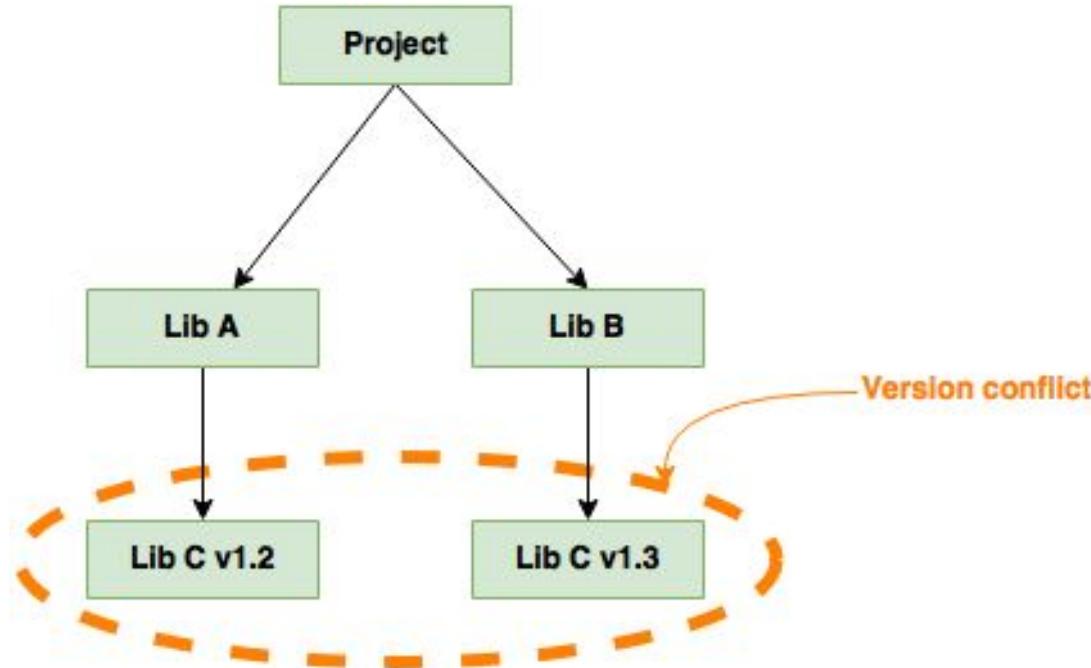
At the bottom right, the status bar shows the time as "1:47", encoding as "n/a", and character encoding as "UTF-8". There are also icons for a user profile and a search function.

# Manual Deployment

- In the “old days” people did deployment like we do now
  - Developers would give operations a “manual” on what was required
- Many things can go wrong!
  - Operations installs the wrong version of Java on machines
  - Dependencies might not work well together
  - Operations can’t install what you need that easily (database software)
- Introducing Operations Engineer



# Classic Failure - Dependencies

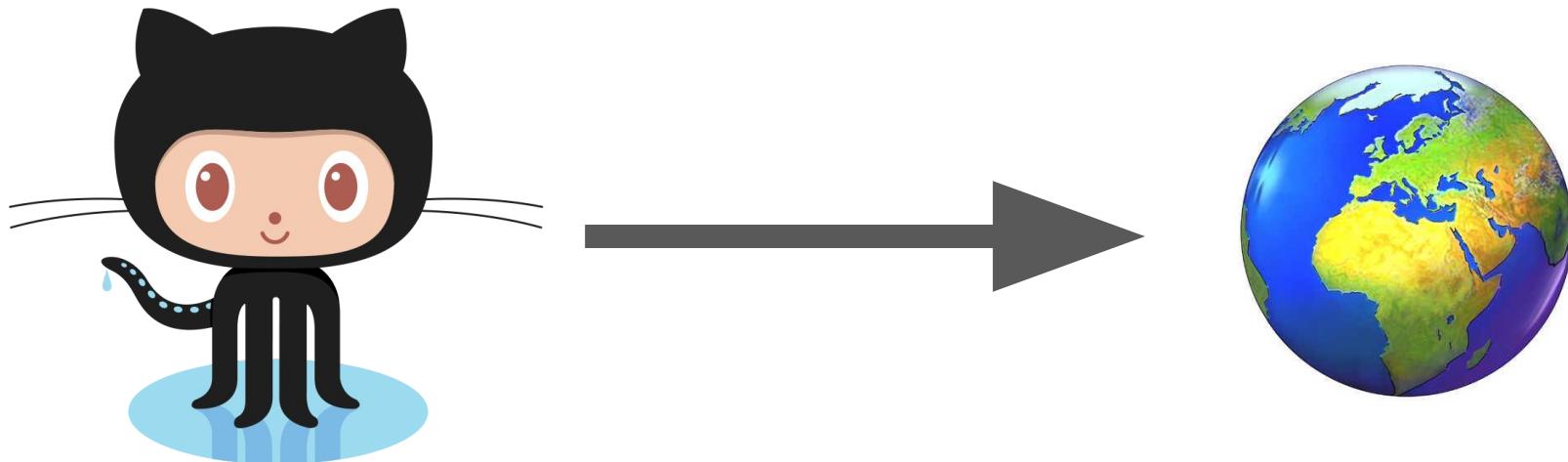


# Before we build pipeline - quick recap!

- A frontend runs on the users computer
  - This is why frontend is more than just design, or HTML, or CSS
  - A console application (such as “Hello World” in Python) is also a type of frontend
- A backend runs on a computer that does not belong to the user. Usually it runs on a virtual machine, such as EC2 in AWS
- Backend and frontend communicate via the HTTP protocol
  - Also often called “An API”, but the term API is used very loosely and can mean almost anything
- A frontend and backend need to be deployed so that end users can interact with them
- The deployment process is very important and should not be left for last
- In our case, we deployed our backend by uploading the JAR-file to an EC2. We did this upload manually.
- Now, we want to automate the upload. This is where the pipeline comes in!

# Pipelines - Automating Deployment

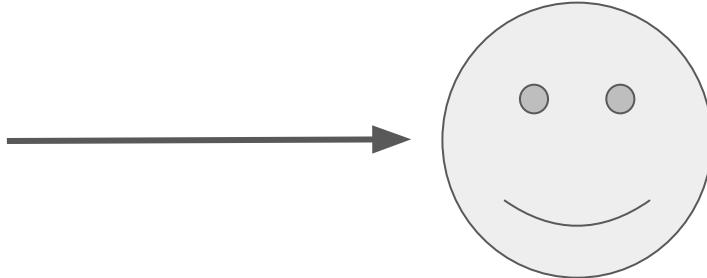
- A pipeline is a mechanism to move your “code” from a GitHub-repo into production
- Behind the scenes, a pipeline is just a computer that runs commands



# Testing

- Testing is important because of automation
- Tests can be added to the pipeline
- Creating tests is like creating your first user

```
Run Test | Debug Test
8  class TestIncrementDecrement(object):
    Run Test | Debug Test
9    def test_increment():
10      assert increment(3) == 4
11
12    Run Test | Debug Test
13    def test_decrement():
14      assert decrement(3) == 2
```



Thank you! :)

# **Control flow and data flow testing**

Jingyue Li (Bill)

# Outline

- Control flow testing and coverage
  - Data flow testing and coverage
  - Mutation testing and coverage

# Industry standards define test needed

- IEEE 1012 – general software verification and validation
- IEC 61508 – general safety critical software
- ISO 26262 – automotive software
- Other domain specific standards, e.g.,
  - Healthcare
  - Oil & Gas
  - Railway

# ISO 26262 – automotive software

The ASIL (Automotive System Integrity Level) – A, B, C, or D – is the outcome of the combination of three factors

- S – Severity. How dangerous is the event
- E – Probability. How likely is the event
- C – Controllability. How easy is it to control the event if it occurs

# Finding the ASIL level

Severity	Probability	C1	C2	C3
S1	E1	QM	QM	QM
	E2	QM	QM	QM
	E3	QM	QM	A
	E4	QM	A	B
S2	E1	QM	QM	QM
	E2	QM	QM	A
	E3	QM	A	B
	E4	A	B	C
S3	E1	QM	QM	A
	E2	QM	A	B
	E3	A	B	C
	E4	B	C	D

# ASIL chart\*

ASIL chart

Functional category	Hazard	ASIL-A	ASIL-B	ASIL-C	ASIL-D
Driving	Sudden start				
	Abrupt acceleration				
	Loss of driving power				
Braking	Maximum 4 wheel braking				
	Loss of braking function				
Steering	Self steering				
	Steering lock				
	Loss of assistance				

\* <https://www.jnovel.co.jp/en/service/compiler/iso26262.html>

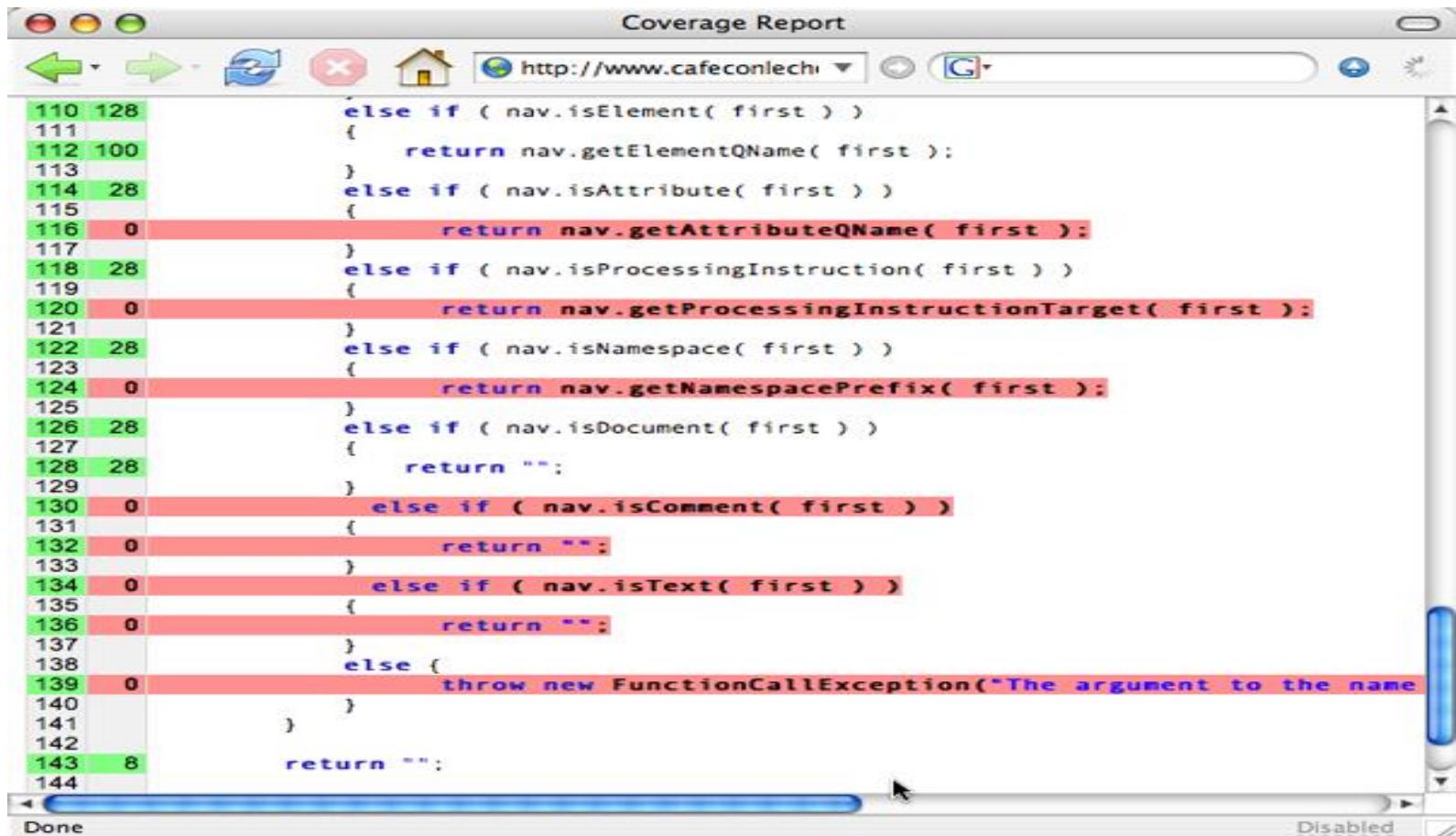
# Methods and measurements for software unit testing

Methods and measures	Sub-methods/Measures	A	B	C	D
Functional tests	...	...	...	...	...
Structural coverage	Statement coverage Decision coverage ...	++ +	++ +	+	+
Resource usage measurement	...	+	+	+	++
Back-to-back test between simulation	...	+	+	+	++

# Control flow coverage

- Statement coverage
- Decision coverage
- Path coverage

# Statement coverage report



The screenshot shows a 'Coverage Report' window with a Java code editor. The code is a switch statement that returns different values based on the type of the input 'first'. The coverage report indicates which lines have been executed (green) and which have not (red). The code is as follows:

```
110 128        else if ( nav.isElement( first ) )
111          {
112 100            return nav.getElementQName( first );
113          }
114 28        else if ( nav.isAttribute( first ) )
115          {
116 0            return nav.getAttributeQName( first );
117          }
118 28        else if ( nav.isProcessingInstruction( first ) )
119          {
120 0            return nav.getProcessingInstructionTarget( first );
121          }
122 28        else if ( nav.isNamespace( first ) )
123          {
124 0            return nav.getNamespacePrefix( first );
125          }
126 28        else if ( nav.isDocument( first ) )
127          {
128 28            return "";
129          }
130 0        else if ( nav.isComment( first ) )
131          {
132 0            return "";
133          }
134 0        else if ( nav.isText( first ) )
135          {
136 0            return "";
137          }
138        else {
139 0            throw new FunctionCallException("The argument to the name
140          )
141          }
142          }
143 8        return "";
144
```

The coverage report shows the following coverage percentages:  
Line 110: 128 (green)  
Line 111: 100 (green)  
Line 112: 28 (green)  
Line 113: 0 (red)  
Line 114: 28 (green)  
Line 115: 0 (red)  
Line 116: 0 (red)  
Line 117: 0 (red)  
Line 118: 28 (green)  
Line 119: 0 (red)  
Line 120: 0 (red)  
Line 121: 0 (red)  
Line 122: 28 (green)  
Line 123: 0 (red)  
Line 124: 0 (red)  
Line 125: 0 (red)  
Line 126: 28 (green)  
Line 127: 0 (red)  
Line 128: 28 (green)  
Line 129: 0 (red)  
Line 130: 0 (red)  
Line 131: 0 (red)  
Line 132: 0 (red)  
Line 133: 0 (red)  
Line 134: 0 (red)  
Line 135: 0 (red)  
Line 136: 0 (red)  
Line 137: 0 (red)  
Line 138: 0 (red)  
Line 139: 0 (red)  
Line 140: 0 (red)  
Line 141: 0 (red)  
Line 142: 0 (red)  
Line 143: 8 (green)  
Line 144: 0 (red)

You can get test coverage report with [GitLab CI](#)

# Coverage report

Coverage Report

http://www.cafeconleche.org/

Packages

- All
- [org.jaxen](#)
- [org.jaxen.dom](#)
- [org.jaxen.dom.html](#)
- [org.jaxen.function](#)
- [org.jaxen.function.ext](#)
- [org.jaxen.function.xslt](#)
- [org.jaxen.javabean](#)
- [org.jaxen.idom](#)
- [org.jaxen.pattern](#)
- [org.jaxen.saxpath](#)
- [org.jaxen.saxpath.base](#)
- [org.jaxen.saxpath.helpers](#)
- [org.jaxen.util](#)
- [org.jaxen.xom](#)
- [Dom4jXPath \(100%\)](#)
- [DocumentNavigator \(36%\)](#)
- [NamespaceNode \(2%\)](#)
- [DocumentNavigator \(0%\)](#)
- [BaseXPath \(77%\)](#)
- [Context \(93%\)](#)
- [ContextSupport \(91%\)](#)
- [DefaultNavigator \(36%\)](#)
- [HTMLXPath \(0%\)](#)

All Packages

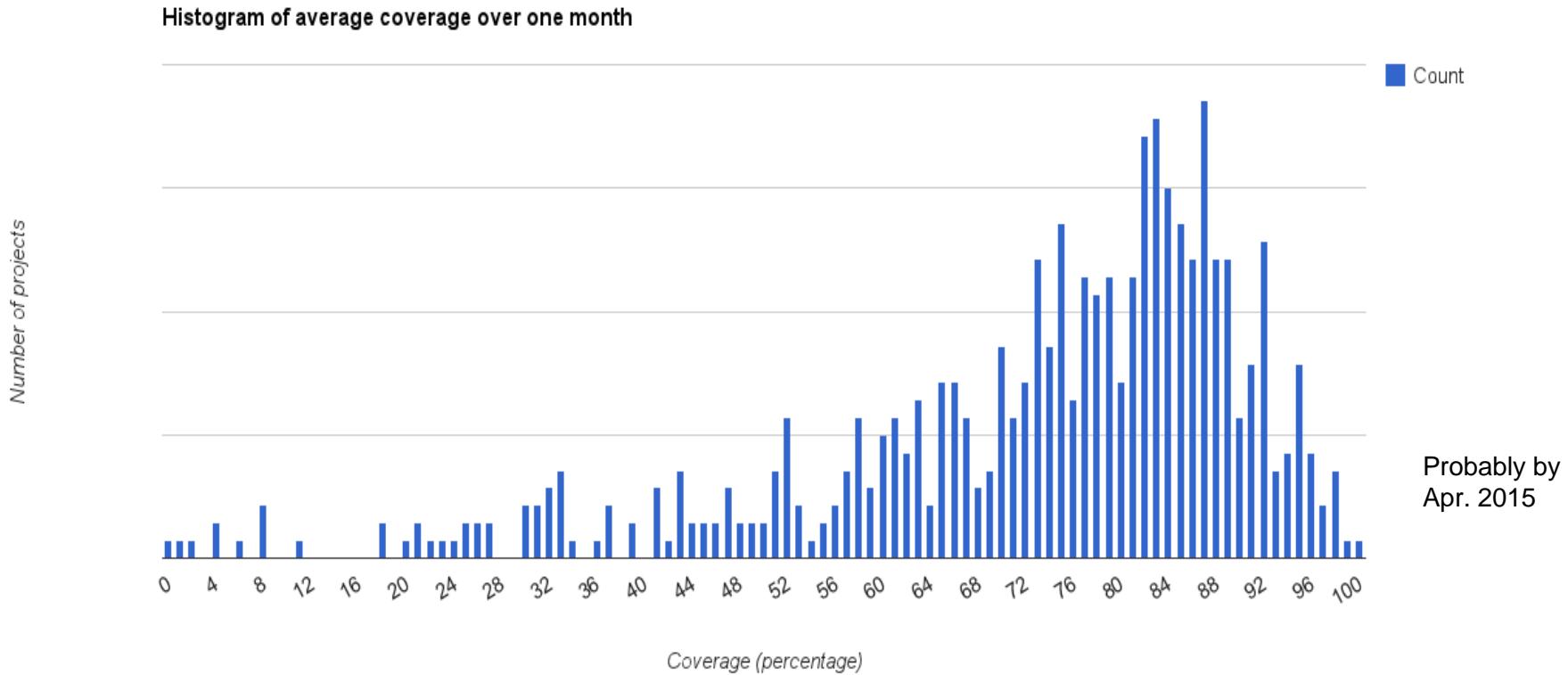
Package	# Classes	Line Coverage	Branch Coverage	Complexity
All Packages	205	69%	80%	2.811
<a href="#">org.jaxen</a>	24	77%	73%	1.38
<a href="#">org.jaxen.dom</a>	3	55%	60%	1.907
<a href="#">org.jaxen.dom.html</a>	2	0%	0%	1.364
<a href="#">org.jaxen.function</a>	2	78%	85%	2.395
<a href="#">org.jaxen.function.ext</a>	73	73%	84%	1.566
<a href="#">org.jaxen.function.xslt</a>	14	98%	100%	1.029
<a href="#">org.jaxen.javabean</a>	27	64%	76%	5.373
<a href="#">org.jaxen.idom</a>	6	63%	72%	4.235
<a href="#">org.jaxen.pattern</a>	1	86%	100%	2.5
<a href="#">org.jaxen.saxpath</a>	4	44%	72%	1.87
<a href="#">org.jaxen.saxpath.base</a>	3	62%	63%	2.897
<a href="#">org.jaxen.saxpath.helpers</a>	13	49%	52%	2.135
<a href="#">org.jaxen.util</a>	8	51%	81%	1.887
<a href="#">org.jaxen.xom</a>	6	95%	100%	10.723
<a href="#">Dom4jXPath (100%)</a>	2	28%	83%	1.34
<a href="#">DocumentNavigator (36%)</a>	15	41%	50%	2.432
<a href="#">NamespaceNode (2%)</a>	2	71%	66%	1.783

Reports generated by [Cobertura](#).

Disabled

# Test coverage at Google

- Per-project statement coverage
- 1.5 years worth of data across hundreds of projects



The median is 78%, the 75th percentile is 85% and 90th is percentile 90%.

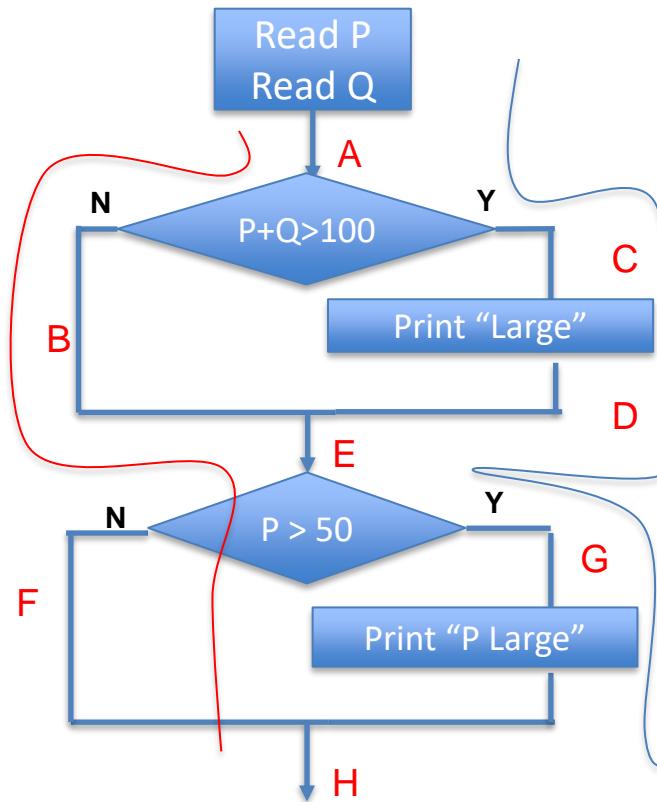
# Decision coverage

- Each one of the possible branch (true/false or switch-case) from each **decision** point is executed at least once

\* Read P  
Read Q

If  $P+Q > 100$  THEN  
    Print "Large"  
ENDIF

IF  $P > 50$   
    Print "P Large"  
ENDIF



Statement coverage:  
A, C, D, E, G, H

Decision coverage:  
A, C, D, E, G, H  
+  
A, B, E, F, H

# Branch coverage

- Count branches covered
- Slightly different from decision coverage

If 3 out of 4 branches of a switch statement are executed

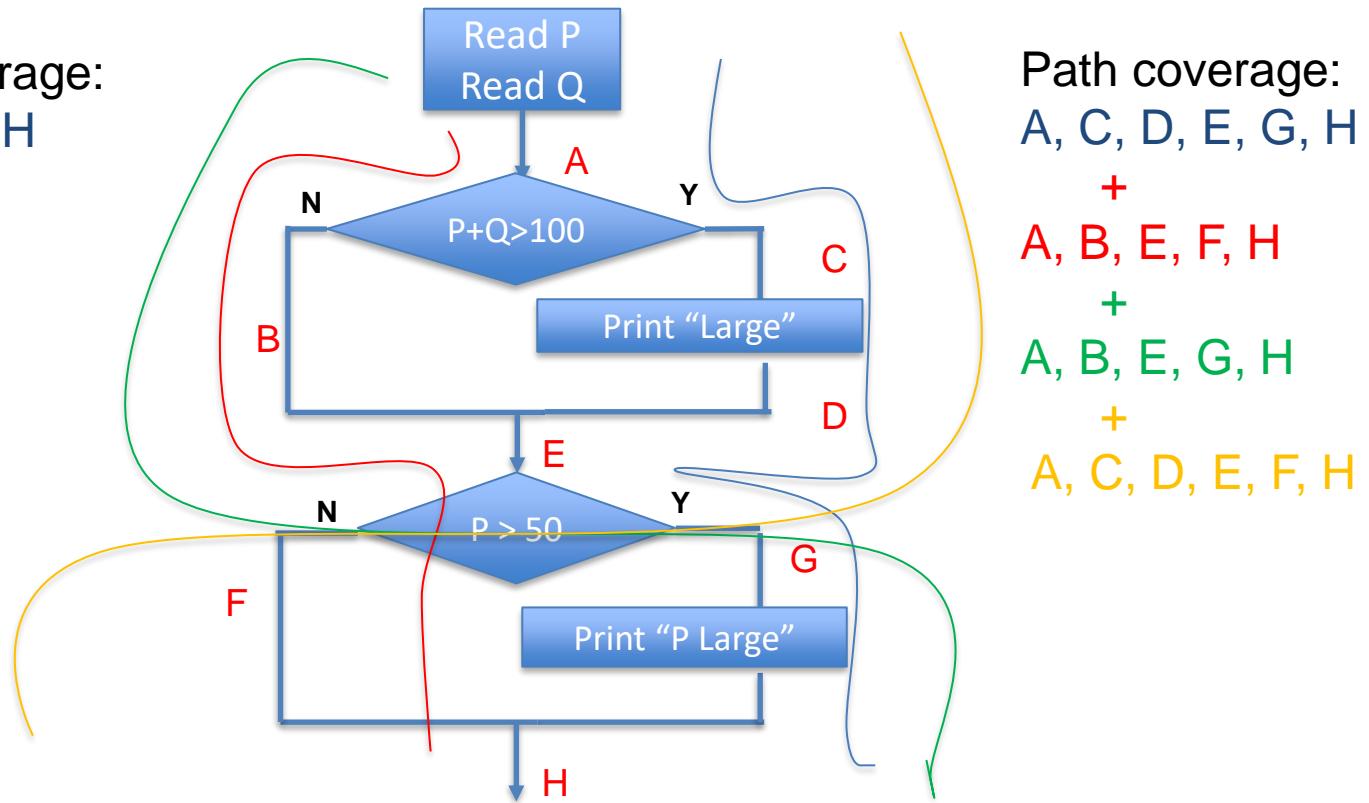
- Branch coverage 75%
- Decision coverage 0% (a decision is considered covered only if all its branches are covered)

# Path coverage

- All possible **paths** of the software should be tested

Decision coverage:  
A, C, D, E, G, H

+  
A, B, E, F, H



# White-box testing methods

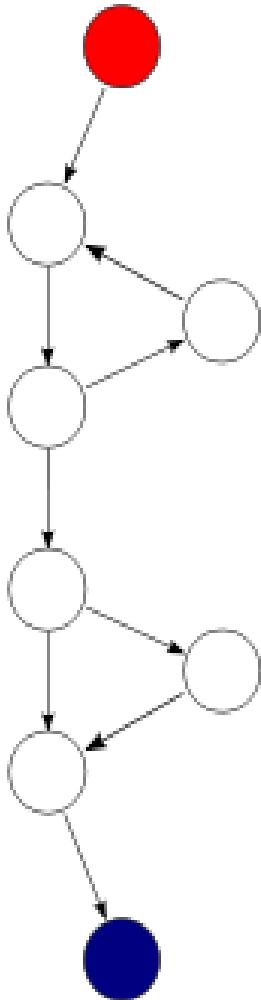
- Usually need to generate control/data flow graph first using some tools, e.g., Java parser (Front End)\*
- Then, use different test inputs to achieve different coverages
- Path coverage is more difficult to achieve
  - How many test cases will be sufficient?
    - McCabe's cyclomatic complexity can help
  - How to systematically create test cases?
    - Decision table can help

\* <http://www.semanticdesigns.com/Products/FrontEnds/JavaFrontEnd.html>

# McCabe's cyclomatic complexity

- From control flow graph of the program
- $v(G) = E - N + 2P$
- $v(G)$  = cyclomatic complexity
- $E$  = the number of edges of the graph
- $N$  = the number of nodes of the graph
- $P$  = the number of connected components

# Graph example

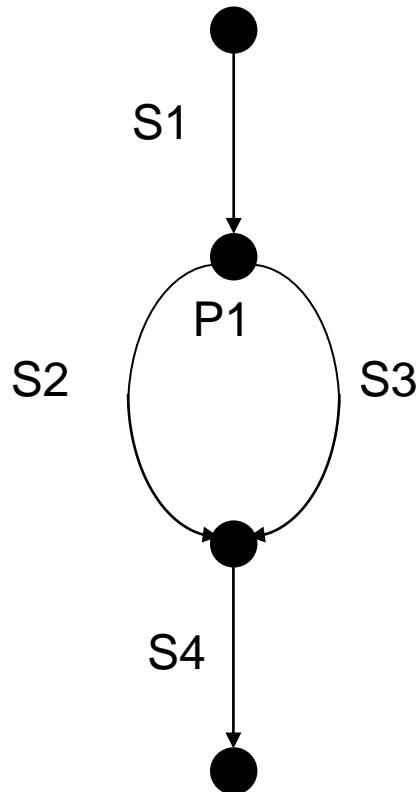


- $v(G) = E - N + 2P$
- E: edge, N:node, P: connected component
- $E = 9$
- $N = 8$
- $P = 1$
- $V(G) = 9 - 8 + 2 \times 1 = 3$

# Using $v(G)$ to count number of paths

- The minimum number of paths through the code is  $v(G)$
- As long as the code graph is a **DAG – Directed Acyclic Graph** – the maximum number of paths is  $2^{**|\{\text{predicates}\}|}$  (when all predicates have only two possible values)
- Thus, we have that  
$$V(G) \leq \text{number of paths} \leq 2^{**|\{\text{predicates}\}|}$$

# Simple case - 1

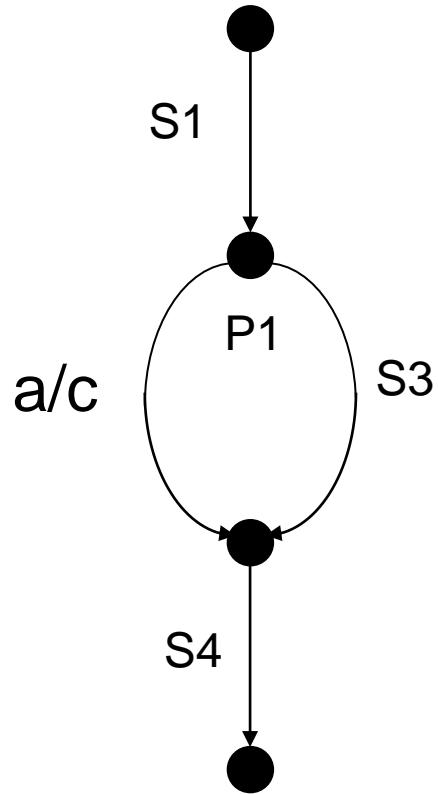


$S_1;$   
IF  $P_1$  THEN  $S_2$  ELSE  $S_3$   
 $S_4;$

$$E = 4, N = 4, P = 1, E-N+2P = 2$$

- $V(G) = 2$ ,
- One predicate ( $P_1$ )
- Two test cases can cover all paths

# Simple case – 2

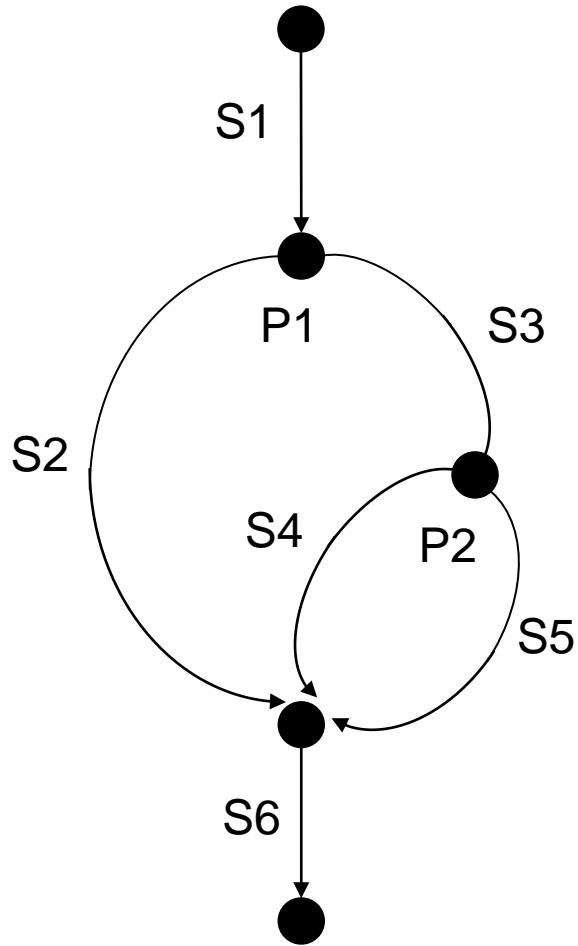


S1;  
IF P1 THEN X := a/c ELSE S3;

S4;

- $V(G) = 2$ ,
- One predicate ( $P1$ )
- Two test cases will cover all paths but not all cases. What about the case  $c = 0$ ?

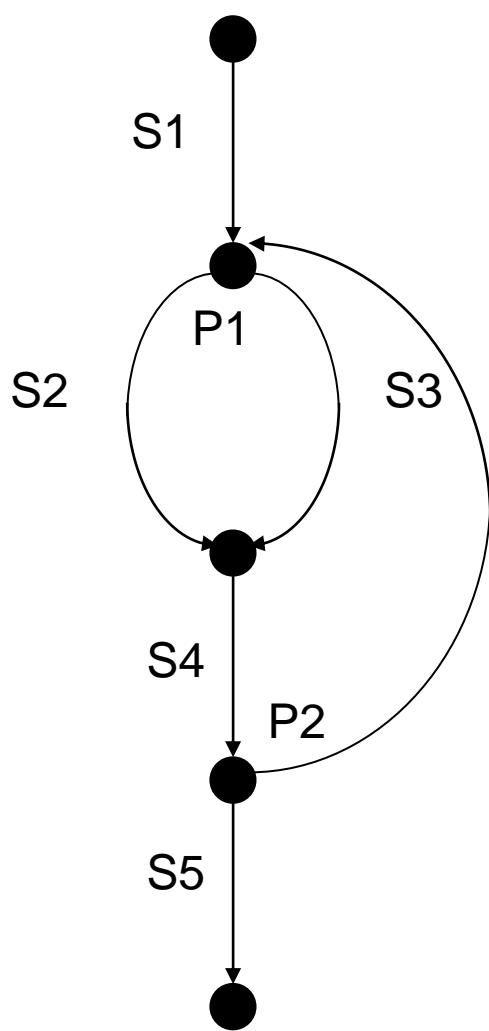
# Nested decisions



S1;  
IF P1 THEN S2 ELSE  
S3;  
IF P2 THEN S4 ELSE S5  
FI  
S6;

$v(G) = 3$ , while Max = 4.  
Three test case will cover all paths.

# Problem – the loop



S1;  
DO  
  IF P1 THEN S2 ELSE S3;  
  S4  
OD UNTIL P2  
S5;

Not a **Directed Acyclic Graph**  
 $v(G) = 3$  and Max  
is 4 but there is an “infinite”  
number of paths

# Problem – the loop (cont’)

- Loops are the great problem in white box testing
- It is common practice to test the system going through each loop
  - 0 times – loop code never executed
  - 1 time – loop code executed once
  - 5 times – loop code executed several times
  - 20 times – loop code executed “many” times

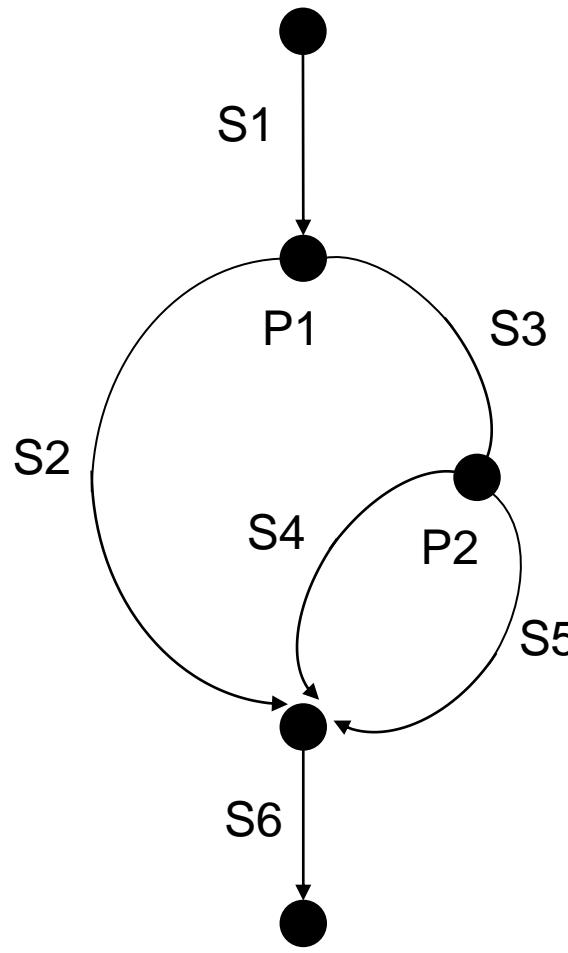
# Use decision table to generate test cases for path coverage

- Make a table of all predicates
  - Insert all combinations of True / False/Switch – 1 / 0/many – for each predicate
  - Construct a test for each combination
- 
- A general technique used to achieve full path coverage
  - However, in many cases, it may lead to over-testing

# Using a decision table

P1	P2	P3	Test description or reference
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

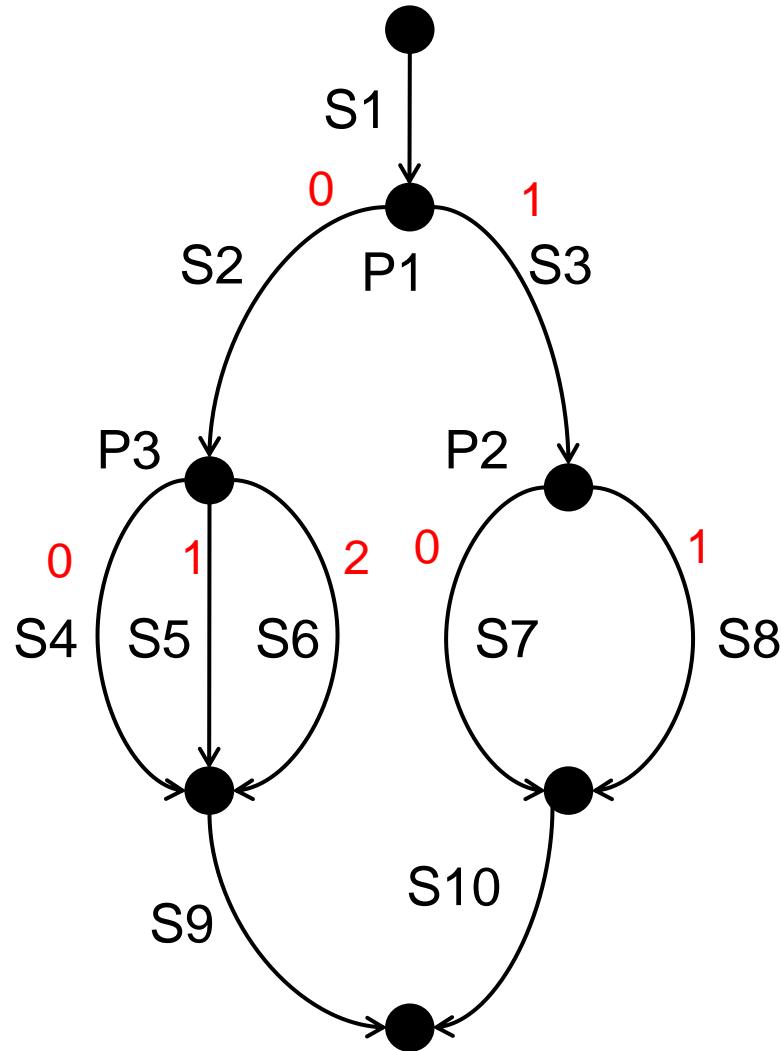
# Decision table example – binary



P1	P2	Test description or reference
0	0	S1, S3, S5, S6
0	1	S1, S3, S4, S6
1	0	S1, S2, S6
1	1	S1, S2, S6

**Go to [www.menti.com](https://www.menti.com) and use the code 5729 3680**

# Exercise



# Outline

- Control flow testing and coverage
- Data flow testing and coverage
- Mutation testing and coverage

# Data flow testing

- Data flow testing uses the **control flow graph** to explore the **unreasonable** things that can happen to data (data flow anomalies)
- Data flow anomalies are detected based on the associations between definition, initialization, and use of variables, e.g.,
  - Variables are used without being initialized

# Variable lifecycle

- Variables have a life cycle
  - Defined and initialized
    - A value is bound to the variable
    - E.g.,  $x = \dots$
  - Used
    - The value of the variable is referred
    - Predicate use (**p-use**), E.g., if ( $x > 10$ )
    - Computational use (**c-use**). E.g.,  $y = x + 1;$
  - Killed (destroyed)

# Variable lifecycle (cont')

```
{          // begin outer block
    int x = 1; // x is defined as an integer within this outer block
    ...
{
    int y = 2; // y is defined within this inner block
    ...
}
...
}
```

// x is defined as an integer within this outer block

// x can be accessed here

// begin inner block

// y is defined within this inner block

// both x and y can be accessed here

// y is automatically destroyed at the end of this block

// x can still be accessed, but y is gone

// x is automatically destroyed

# Data flow anomalies and normal use

- **dd:** define and then define again – suspicious
- **dk:** define and then kill – potential bug
- **ku:** kill and then used – serious defect
- **kk:** kill and then kill again – potential bug
- **du:** define and then use – OK
- **kd:** kill and then redefine – OK
- **ud:** use and then redefine – OK
- **uk:** use and then kill – OK
- **uu:** use and then use – OK
- ...

# Why data-flow testing?

- To uncover possible bugs in data usage during the execution of the code
- Test cases are created to trace every definition to each of its use, and every use is traced to each of its definition
- Various coverage strategies are employed for the creating of the test cases

# Data-flow test strategies

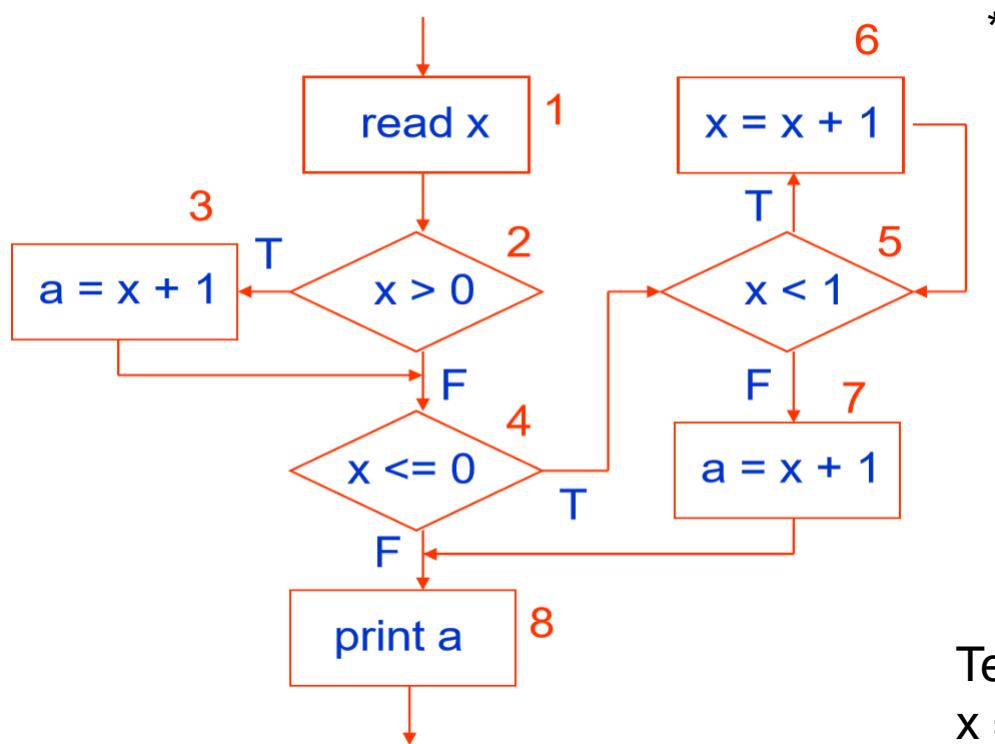
- All definitions (AD): **every definition** of every variable should be covered by **at least one use** (c-use or p-use) of that variable
- All computational uses (ACU): for every variable, there is a path from **every definition** to **every c-use** of that definition
- All predicate-uses (APU): for every variable, there is a path from **every definition** to **every p-use** of that definition
- All c-uses/some p-uses (ACU+P): for every variable and **every definition** of that variable, at least one path from the definition to **every c-use** should be included. If there are definitions of the variable with **no c-use** following it, then **add p-use** test cases to cover every definition

# Data-flow test strategies (cont')

- All p-use/some c-use (APU+C): for every variable and **every definition** of that variable, at least one path from the definition to **every p-use** should be included. If there are definitions of the variable with **no p-use** following it, then **add c-use** test cases to cover every definition
- All uses (AU): for every variable, there is a path from **every definition** to **every use** of that definition, i.e., APU + ACU
- All du paths (ADUP): test cases **cover every simple sub-path** from each variable definition to **every p-use** and **c-use** of that variable
- Note that the “**kill**” usage is not included in any of the test strategies

# All definitions

- All definitions (AD): **every definition** of every variable should be covered by at least **one use** (c-use/p-use) of that variable.



AD

- (1, 2, x)
- (3, 8, a)
- (7, 8, a)
- (6, 5, x)

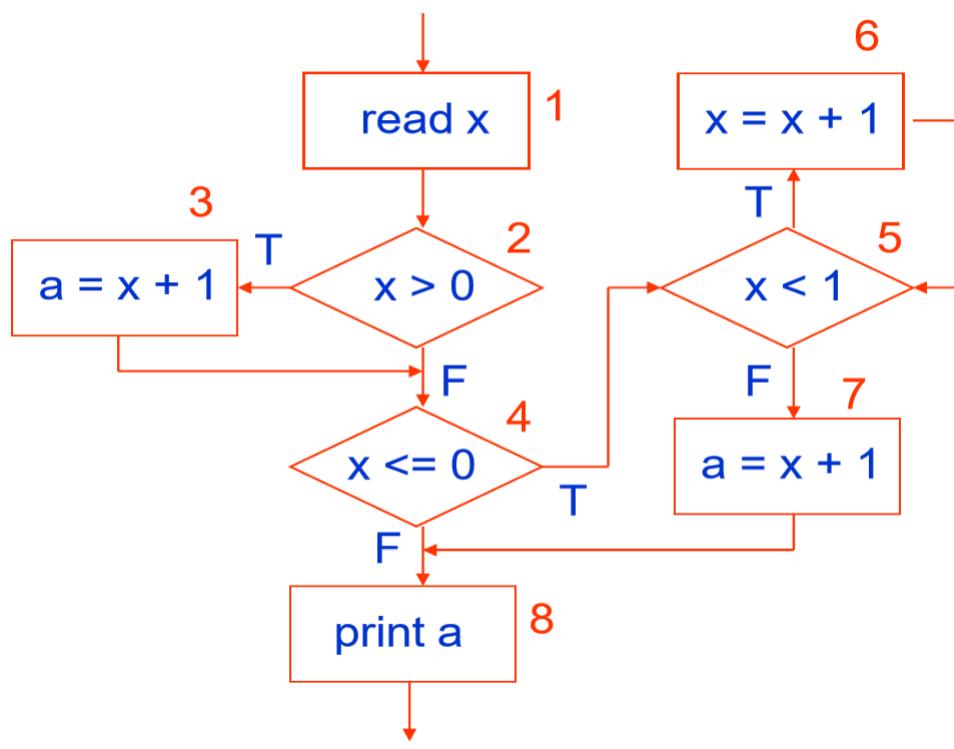
Test cases for AD coverage

$x = 0, a = \text{any}$  (1, 2, 4, 5, 6, 5, 7, 8)  
 $x = 1, a = \text{any}$  (1, 2, 3, 4, 8)

\*<https://www.cs.ccu.edu.tw/~naiwei/cs5812/st5.pdf>

# All computational uses

- All computational uses (ACU): for **every** variable, there is a path from **every** definition to **every c-use** of that definition

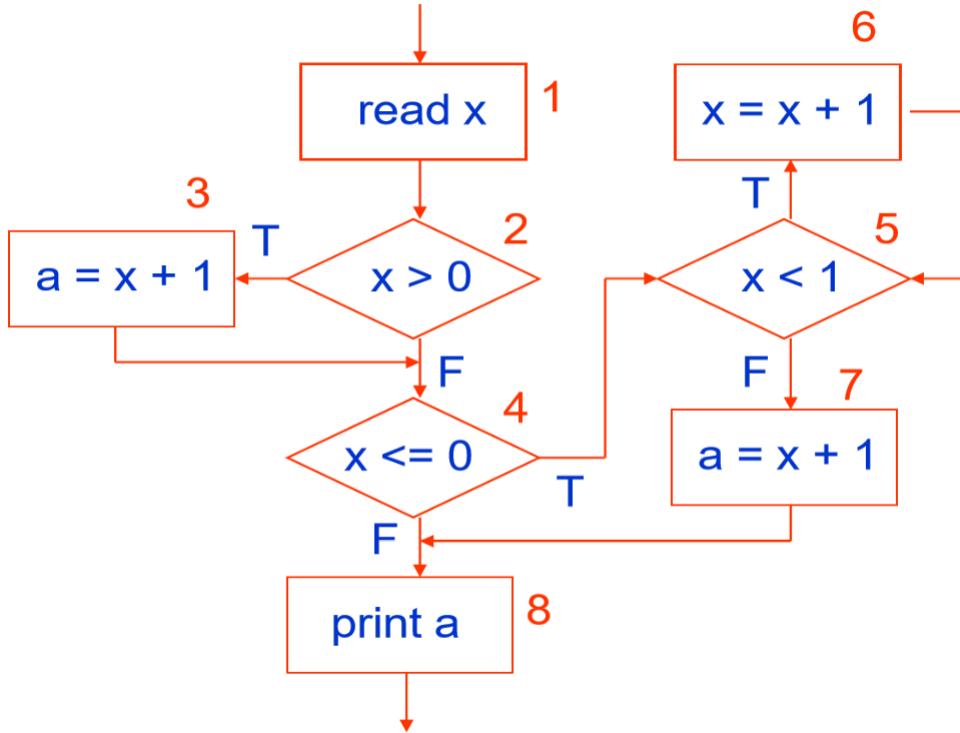


ACU

- $(1, 3, x)$
- $(1, 6, x)$
- $(1, 7, x)$
- $(6, 6, x)$
- $(6, 7, x)$
- $(3, 8, a)$
- $(7, 8, a)$

# All predicate-uses

- All predicate-uses (APU): for **every** variable, there is a path from **every definition** to **every p-use** of that definition

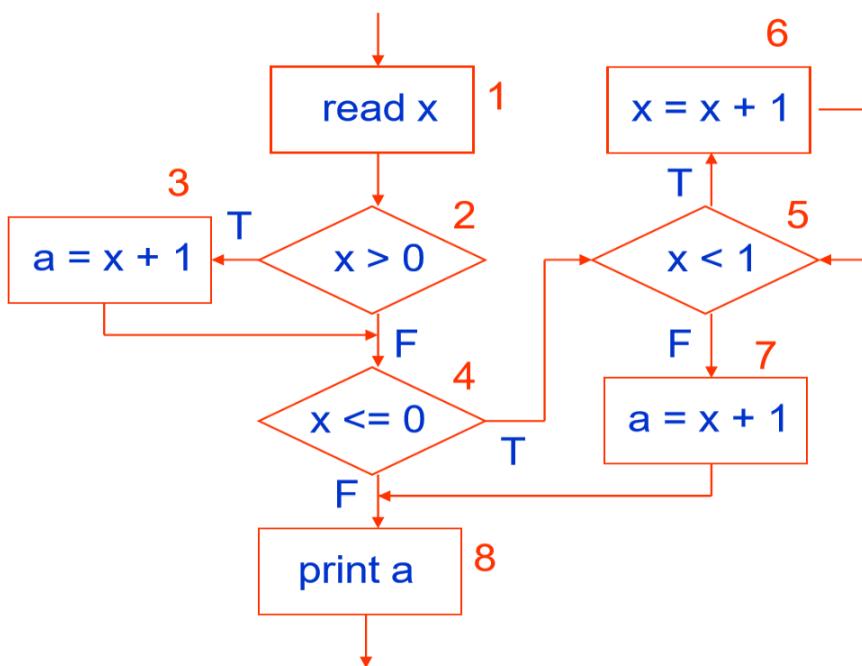


APU

- (1, 2,  $x$ )
- (1, 4,  $x$ )
- (1, 5,  $x$ )
- (6, 5,  $x$ )

# All c-uses/some p-uses

- All c-uses/some p-uses (ACU+P): for **every** variable and **every definition** of that variable, at least one path from the definition to **every c-use** should be included. If there are definitions of the variable with **no c-use** following it, then **add p-use test cases** to cover every definition



ACU  
• (1, 3, x)

• (1, 6, x)

• (1, 7, x)

• (6, 6, x)

• (6, 7, x)

• (3, 8, a)

• (7, 8, a),

ACU + P  
• (1, 3, x)

• (1, 6, x)

• (1, 7, x)

• (6, 6, x)

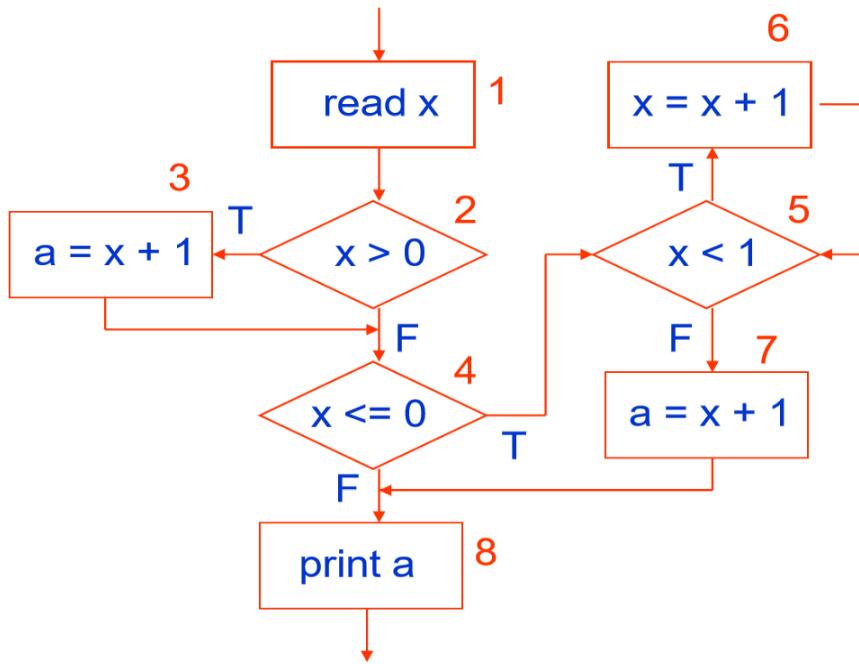
• (6, 7, x)

• (3, 8, a)

• (7, 8, a)

# All p-use/some c-use

- All p-use/some c-use (APU+C): for every variable and **every definition** of that variable, at least one path from the definition to **every p-use** should be included. If there are definitions of the variable with **no p-use** following it, then **add c-use** test cases to cover every definition



APU

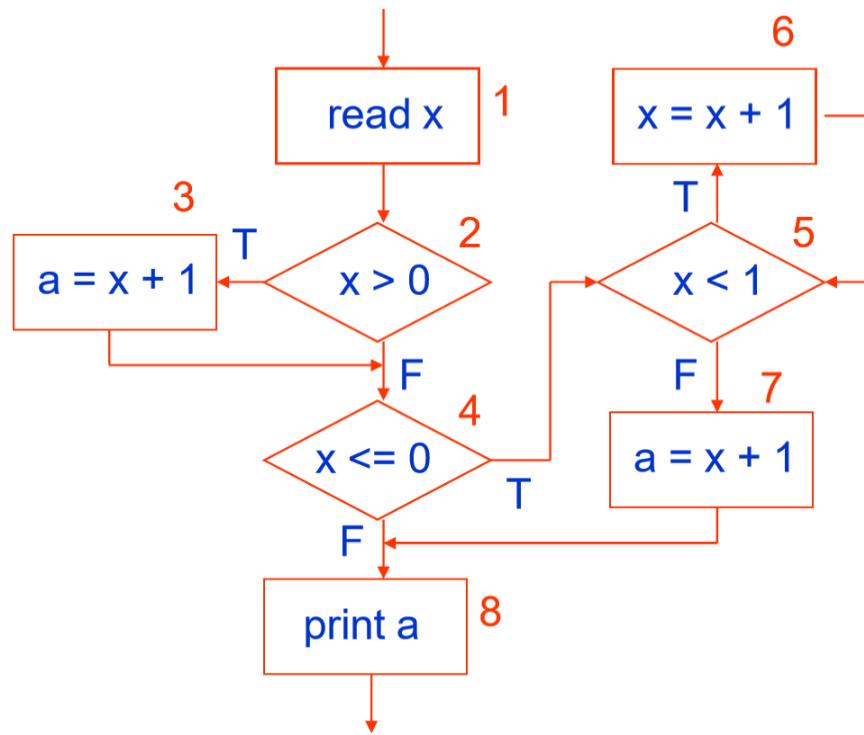
- (1, 2, x)
- (1, 4, x)
- (1, 5, x)
- (6, 5, x)

APU + C

- (1, 2, x)
- (1, 4, x)
- (1, 5, x)
- (6, 5, x)
- (3, 8, a)
- (7, 8, a)

# All uses

- All uses (AU): for **every** variable, there is a path from **every definition** to **every use** of that definition, i.e., APU + ACU

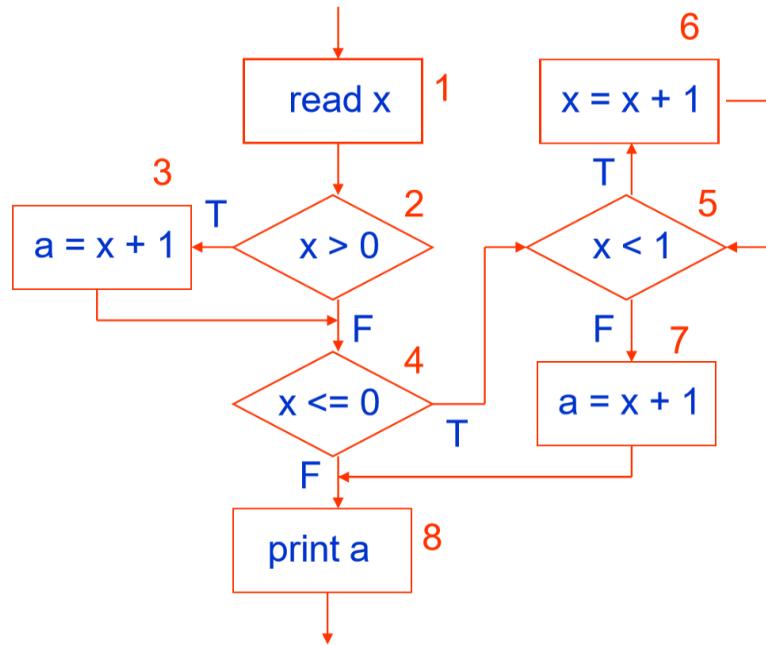


AU

- (1, 2, x)
  - (1, 4, x)
  - (1, 5, x)
  - (6, 5, x)
- APU
- 
- (1, 3, x)
  - (1, 6, x)
  - (1, 7, x)
  - (6, 6, x)
  - (6, 7, x)
- ACU
- 
- (3, 8, a)
  - (7, 8, a)

# All du paths

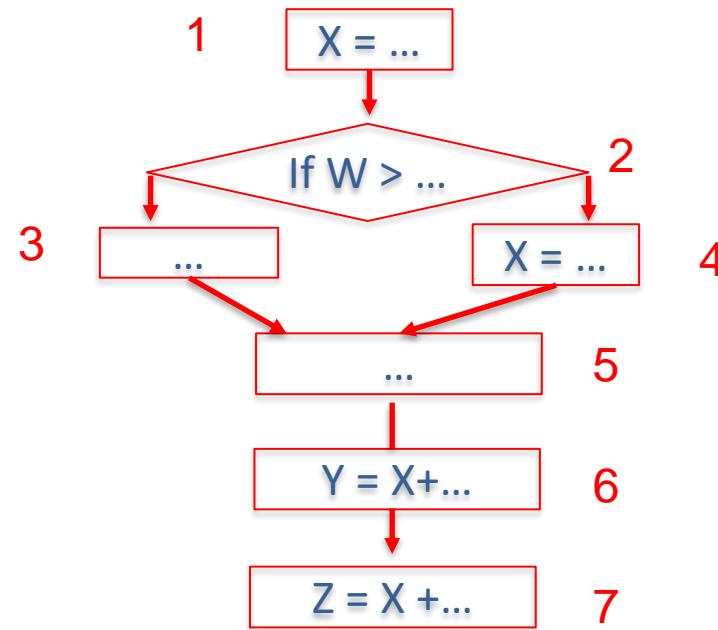
- All du paths (ADUP): More demanding than All uses (AU). If there are **multiple paths** between a given definition and a use, **they must all be included**
- However, ADUP includes loop just once



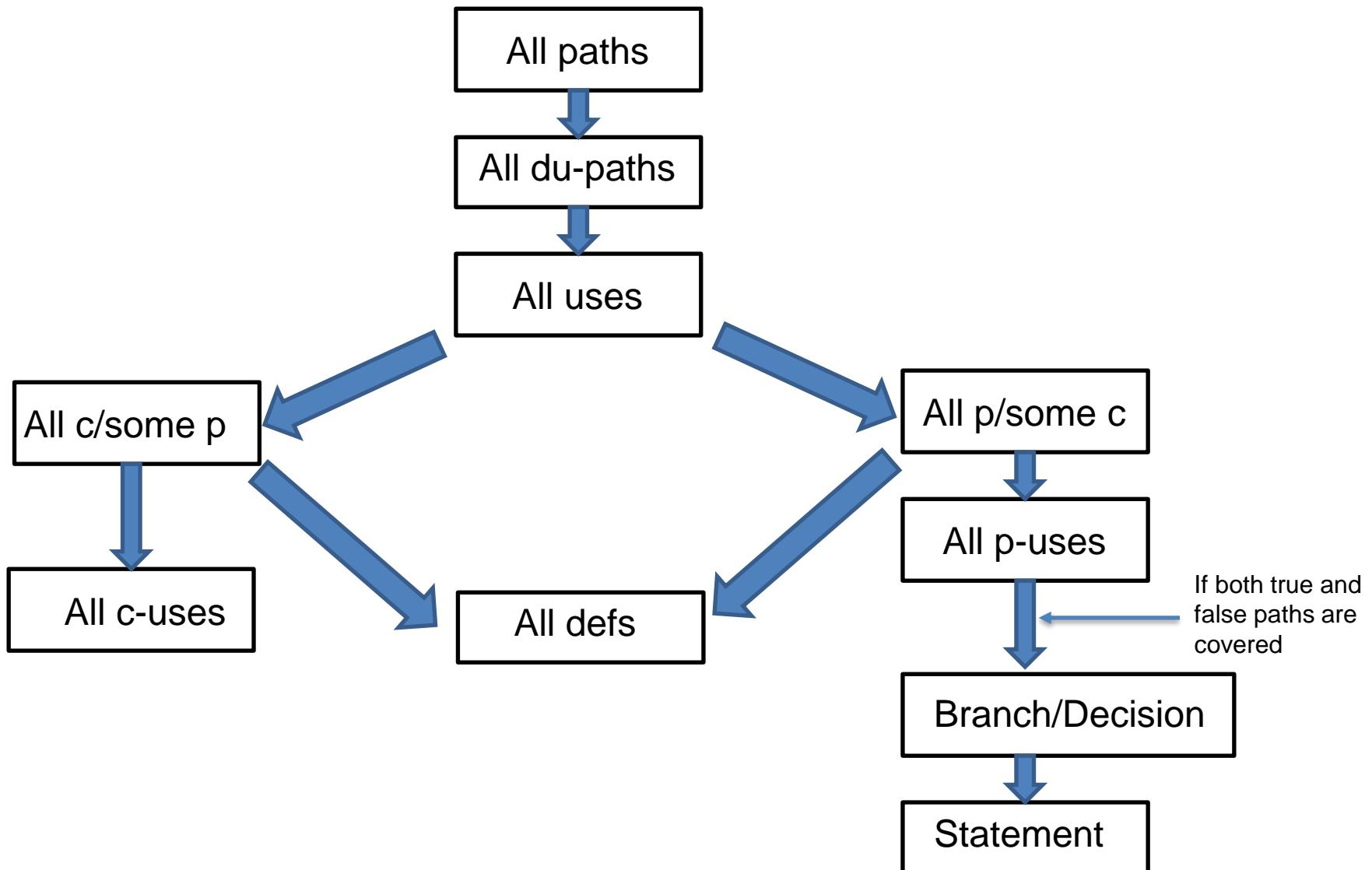
- In AU, we include path (3, 8, a), meaning either (3, 4, 5, 7, 8) **OR** (3, 4, 8) **OR** (3, 4, 5, 6, 5, 7, 8) is ok
- In ADUP, since there are multiple paths from 3 to 8, we need to include (3, 4, 5, 7, 8) **AND** (3, 4, 8) **AND** (3, 4, 5, 6, 5, 7, 8)

**Go to [www.menti.com](https://www.menti.com) and use the code 5729 3680**

# Exercise



# Relationship between strategies

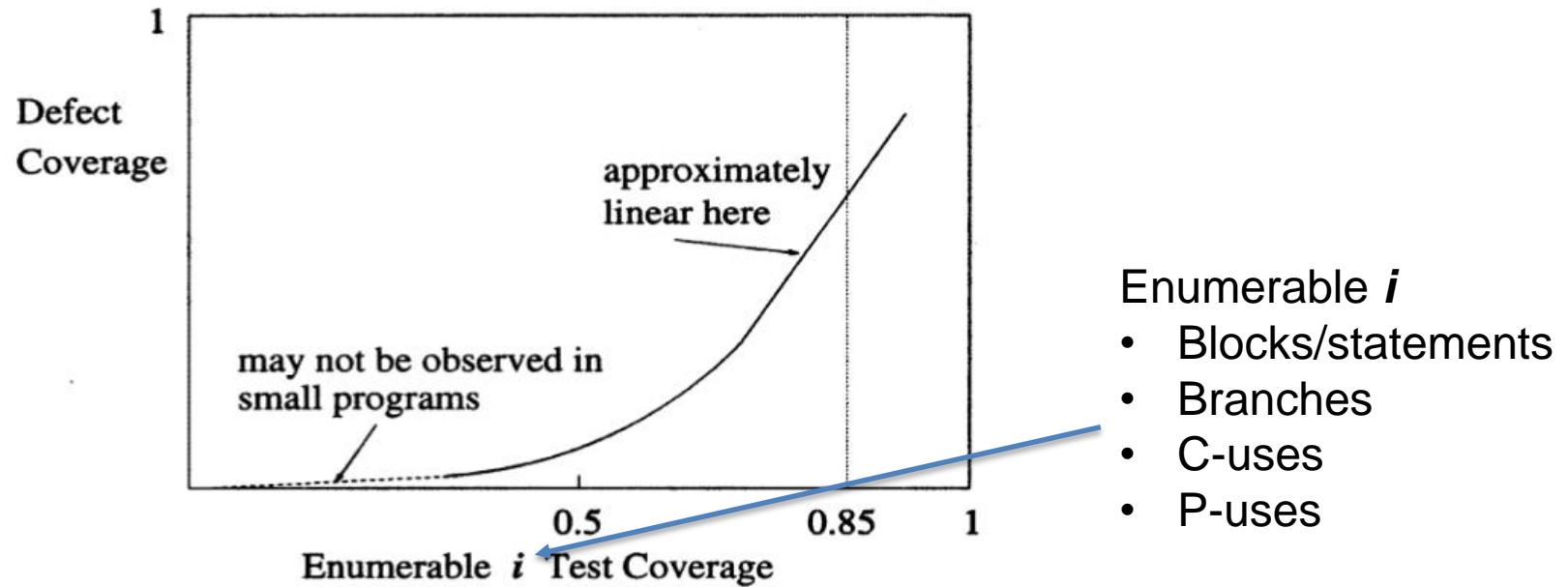


# Why do we measure test coverage?

- One way of using statement/branch/path/dataflow coverage is to use it as **a test acceptance criteria**
  - Run test cases
  - Have we reached, e.g., 85% statement coverage
    - Yes – stop testing
    - No – coverage measure will help us directly identify untested code and indirectly help develop/select new test cases

# Use test coverage to estimate defect coverage

- Defect coverage: the fraction of actual defects initially present that would be detected by a given test set

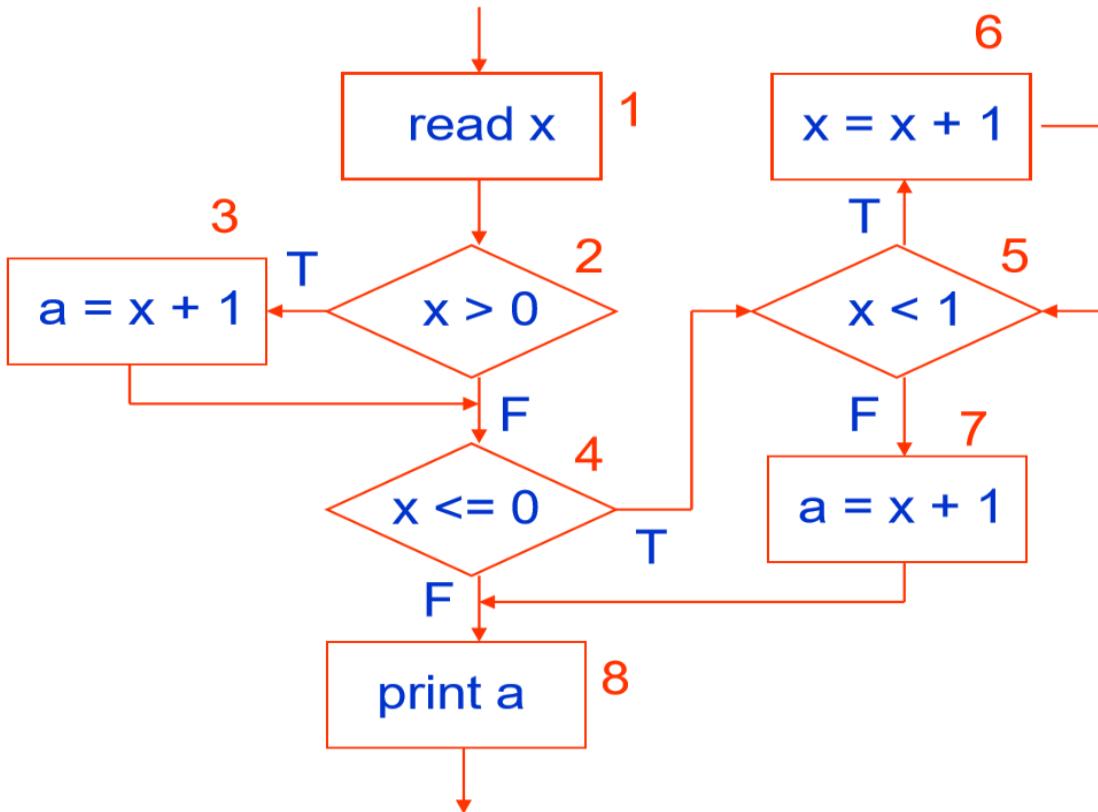


Logarithmic-exponential relationship between test coverage and defect coverage

# Use coverage for program slicing

- Static backward slicing
  - A slice with respect to a variable  $v$  at a certain point  $p$  in the program is **the set of statements that contributes** to the value of the variable  $v$  at  $p$
  - $S(v, p)$  denotes the set of **nodes** in the control flow graph that **contributes to the value of the variable  $v$  at point  $p$**

# Static backward slicing

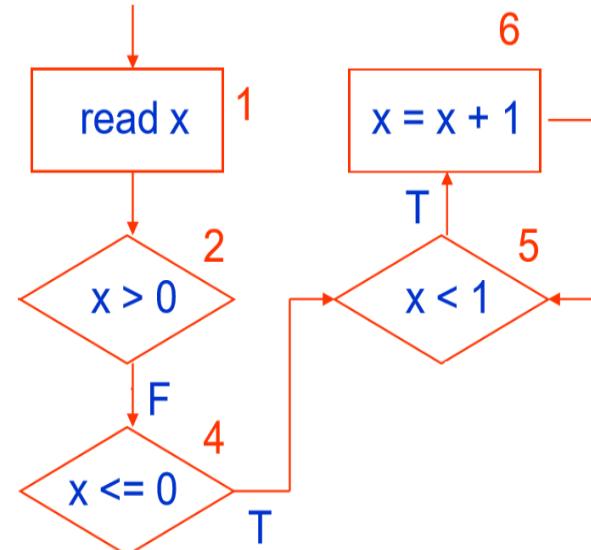
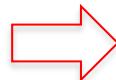
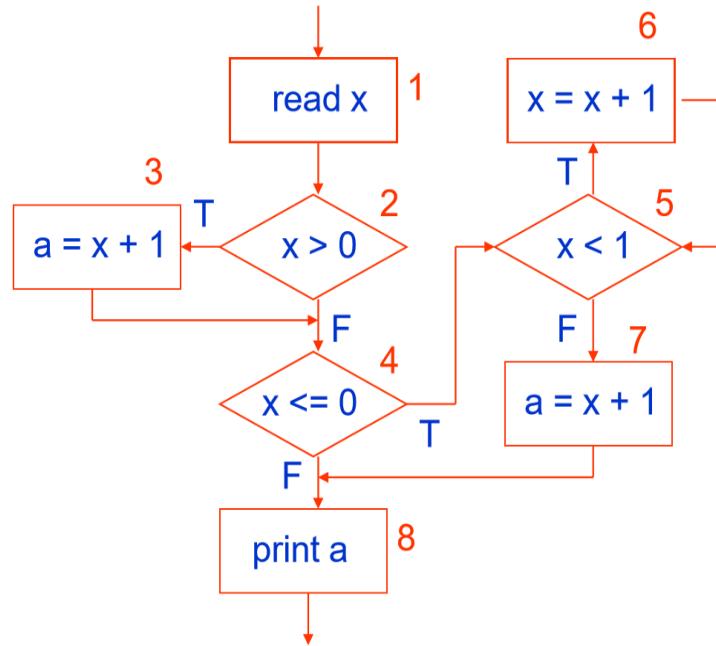


All **Defs** and All **P-uses** of a variable  $v$  before a certain point  $p$  are used for slicing\*

\* <http://www.cs.cmu.edu/afs/cs/academic/class/17654-f01/www/refs/Binkley.pdf>

# Program slicing can help many activities

- Facilitate debugging (Finding and localizing errors)
- Facilitate programming understanding



$$S(x, 6) = \{1, 2, 4, 5, 6\}$$

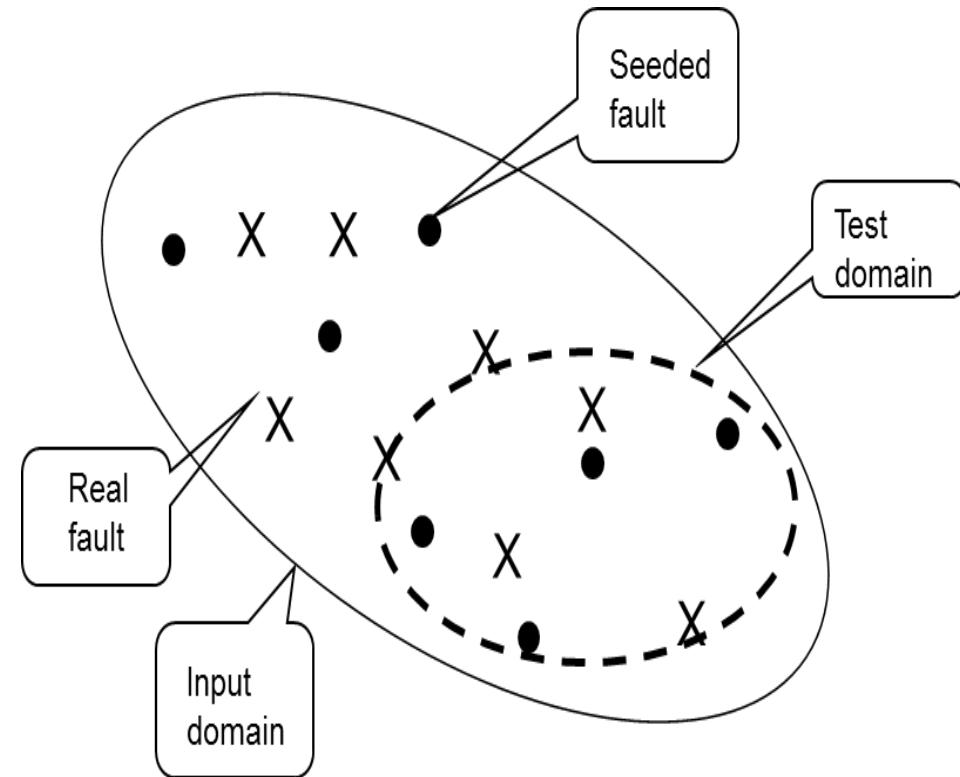
# Outline

- Control flow testing and coverage
- Data flow testing and coverage
- Mutation testing and coverage

# Mutation testing and coverage

- The tester can also measure **coverage of seeded faults** and use it as an indicator to measure whether the test set is adequate
  - Seed faults in the code
  - Have we found all the seeded faults
    - Yes – test set is adequate
    - No – One or more seeded faults are not found.  
This tells us which **locations and types of defects** have not been covered by test cases sufficiently.  
Then we need to develop/select more test cases.

# Fault seeding and estimation



- We assume  $N_0 / N = S_0 / S$
  - Thus  $N_0 = N * S_0 / S,$
- 
- $N_0$ : number of faults in the code
  - $N$ : number of faults found using a specified test set
- 
- $S_0$ : number of seeded faults
  - $S$ : number of seeded faults found using a specified test set

# Typical mutants

- Where and how to seed the faults
  - Save and seed faults identified during earlier project activities
  - Draw faults to seed from an experience database containing typical faults
    - Arithmetic operator replacement
    - Logical connector replacement
    - Relational operator replacement
    - Absolute value insertion, etc.
- More info. can be found at
  - <http://www.uio.no/studier/emner/matnat/ifi/INF4290/v10/undervisningsmateriale/INF4290-Mutest.pdf>
  - <https://www.softwaretestinghelp.com/what-is-mutation-testing/>
  - [https://pitest.org/java\\_mutation\\_testing\\_systems/](https://pitest.org/java_mutation_testing_systems/)

# Code to be tested

- This is what the code might look like:
  - 1)** *Read Age*
  - 2)** *If age>14*
  - 3)** *Doctor= General Physician()*
  - 4)** *End if*

# Seed faults (mutants)

- Mutation type 1: Relational operator replacement

## Mutant #1:

- 1) Read Age
- 2) If age<14 ‘Changing the > with <’
- 3) Doctor= General Physician()
- 4) End if

## Mutant #2:

- 1) Read Age
- 2) If age=14 ‘Changing the > with =’
- 3) Doctor= General Physician()
- 4) End if

# Mutation coverage

- If our test inputs are 14 and 15. Which of these mutants will be killed?

Test inputs	Expected result	Changing > with <	Changing > with =
		Mutant 1	Mutant 2
14	GP not assigned	Succeeds - GP not assigned	Fails - GP assigned
15	GP assigned	Fails - GP not assigned	Fails - GP not assigned

- 14 finds a failure when runs again mutant 2, not effective against mutant 1
- 15 finds failures when runs again mutants 1 and 2

# Summary

- We studied
  - Control flow test and related coverage measurement
  - Data flow test and related coverage measurement
  - Mutation test and coverage measurement

# Next lecture

- Black-box unit testing

# **Domain and function testing**

# Methods and measures for functional software integration testing

Methods and measures	Criteria	ASIL			
		A	B	C	D
Requirements based test	Test case have to be derived from the requirements. All SW component requirements are covered	++	++	++	++
External interface test	Test cases can be determined with the help of <b>equivalence classes</b> and <b>input partitioning</b> of the external interface. This can be completed by <b>boundary value analysis</b> . Boundary values for data types as well as plausible ranges of values for interface variables have to be considered.	+	++	++	++
Fault injection test	...	+	+	++	++
Error guessing test	...	+	+	++	++

# Outline

- Single variable
- Multiple variables
- Error guessing/Risk based
- Output coverage and testing

# The challenge



The total number of possible combinations of X and Y is  $2^{32} \times 2^{32}$

Combinatorial explosion!

- We need to **systematically** reduce the set of **all possible values** to few **manageable** subsets

# Partitioning of domain

- Identifying domain first
  - Linear (variables whose values can be mapped onto a number line, e.g., integer variables)
  - Non-linear (e.g., variables with enumeration values)
- Ideally, partitioning of input & output domain should result in non-overlapping or disjoint partitions

# Black-box domain testing approaches

- Identify input and output domains
- Generate test case to cover single variables in the domains and combination of the variables
  - Equivalence classes, boundary values
  - Decision tables, pairwise testing
  - Error guessing/risk-based

# Partitioning of domain of single variable

- Variable types
  - Linear domain variable (e.g., integer)
  - Linear domain variable with multiple ranges
  - String variable
  - Enumerated variable
  - Multidimensional variable

# Partitioning domain of single variable

- Step 1: Identify the variable
- Step 2: Determine the domain
  - All possible values of the variable
- Step 3: Identify risks
- Step 4: Partition the domain into equivalence classes based on the identified risks

# Linear domain variable

- Software application example

BankA issues Visa credit cards with credit limits in the range of \$4000 to \$40000. A customer is not to be approved for credit limits outside this range.

A customer can **apply for the card using an online application form** in which one of the fields requires that the customer type in his/her desired credit limit.

# Linear domain variable (cont')

- Identify the variable: “credit-limit”
- Determine the input domain:  $\$4000 \leq \text{credit-limit} \leq \$40000$
- Identify risks
  - Failure to process credit limit requests between \$4000 and \$40000 correctly
  - Failure to disapprove credit limit requests less than \$4000
  - Failure to disapprove credit limit requests greater than \$40000
  - Mishandling of negative credit limit requests
- Partition the input domain into equivalence classes based on risks
  - \$4000, \$40000
  - \$3999
  - \$4001
  - -\$4000

# Equivalence class testing

- Rationale: **complete** testing but **no redundancy**
- **Cover each partition at least once**
- E.g., to test whether software can identify the equilateral triangle
  - After using  $(5, 5, 5)$  as test input
  - We do not expect to learn much from using  $(6, 6, 6)$ ,  $(10, 10, 10)$  as test inputs

# Linear domain variable with multiple ranges

If Taxable Income Is Between:	The Tax Due Is:
0 - \$9,225	10% of taxable income
\$9,226 - \$37,450	\$922.50 + 15% of the amount over \$9,225
\$37,451 - \$90,750	\$5,156.25 + 25% of the amount over \$37,450
\$90,751 - \$189,300	\$18,481.25 + 28% of the amount over \$90,750
\$189,301 - \$411,500	\$46,075.25 + 33% of the amount over \$189,300
\$411,501 - \$413,200	\$119,401.25 + 35% of the amount over \$411,500
\$413,201 +	\$119,996.25 + 39.6% of the amount over \$413,200

# Linear domain variable with multiple ranges (cont')

- Identify risks
  - Failure to calculate the tax correctly for each of the income sub-ranges
  - Mishandling of low and high boundaries of each of the sub-ranges
  - Mishandling of values just beneath and beyond low and high boundaries respectively for each of the sub-ranges

# Equivalence class testing vs. boundary value testing

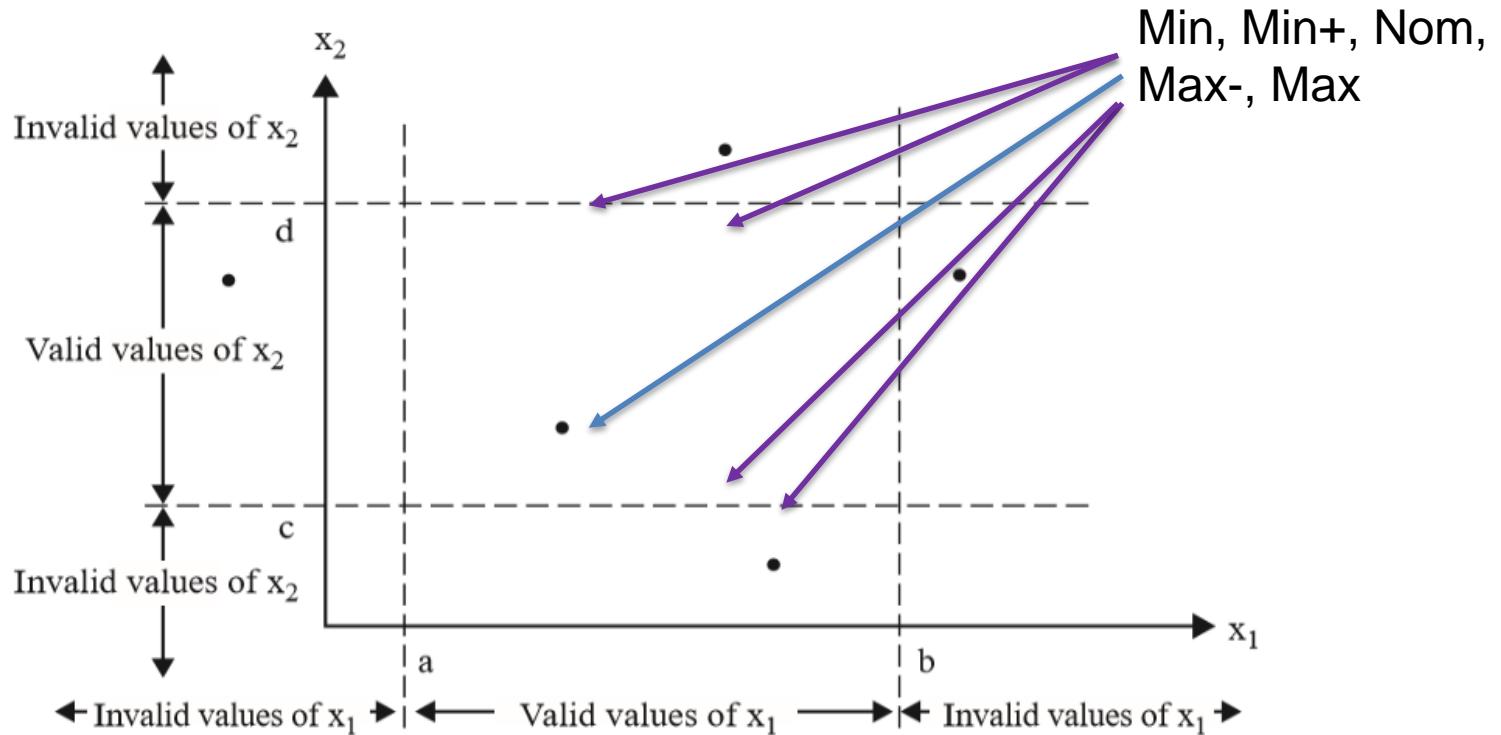


Figure 6.1 Traditional equivalence class test cases.

\*

\* Software Testing, A craftsman's approach, 4<sup>th</sup> Edition

# Boundary value testing

- Rationale: Errors tend to occur near the extreme values of an input variable, e.g.,
  - Loop condition, may test for  $<$  when they should test  $\leq$
- The idea is to use input values  
Min, min+, nom, max-, max
- Two variables (hold one in nom and vary the other one)  
$$<X_{1\text{nom}}, X_{2\text{min}}> <X_{1\text{nom}}, X_{2\text{min+}}> <X_{1\text{nom}}, X_{2\text{nom}}> <X_{1\text{nom}}, X_{2\text{max-}}> <X_{1\text{nom}}, X_{2\text{max}}>$$
$$<X_{1\text{min}}, X_{2\text{nom}}> <X_{1\text{min+}}, X_{2\text{nom}}> \quad <X_{1\text{max-}}, X_{2\text{nom}}> <X_{1\text{max}}, X_{2\text{nom}}>$$
- Several variables (hold one in nom and vary the others)

# Robust boundary value testing

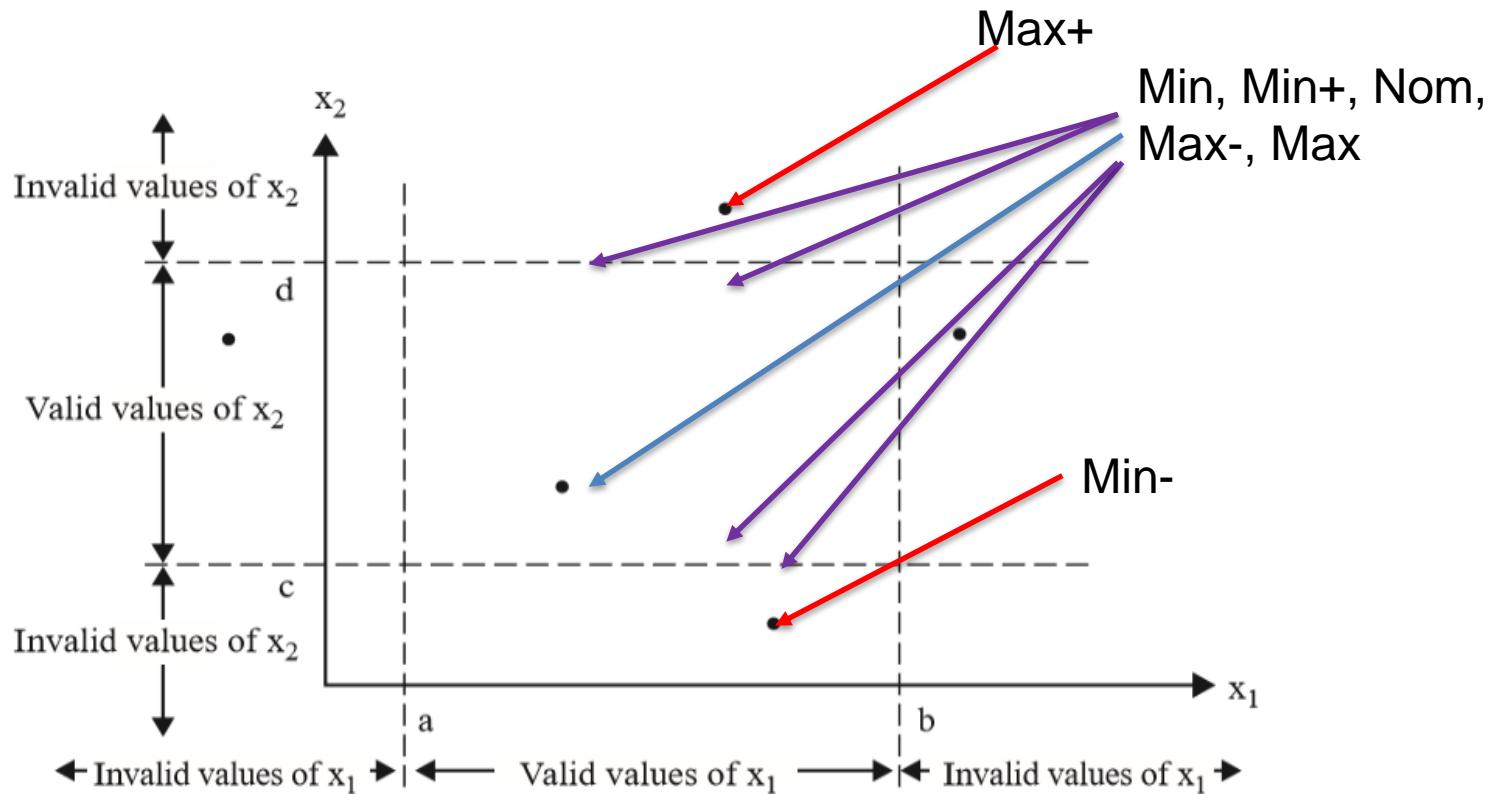


Figure 6.1 Traditional equivalence class test cases.

\*

\* Software Testing, A craftsman's approach, 4<sup>th</sup> Edition

# Special value testing

- Special value testing of the tax calculation system
  - Mishandling of **non-numbers**
  - Mishandling of **negative** incomes
  - Mishandling of **smallest and largest** value at the system level, and beyond
  - ...
- Another special value testing example
  - E.g., test how the system react if **29 Feb.** as the input of a date
- Most intuitive and least uniform (ad hoc)

# String variables

- For example, a string variable ‘s’ has to have a letter for the first character; the rest of them can be any printable graphic ASCII characters.
- Domain
  - First character
    - (A-Z)  $65 \leq \text{ASCII}(\text{first character}) \leq 90$
    - (a-z)  $97 \leq \text{ASCII}(\text{first character}) \leq 122$
  - Remaining characters
    - $32 \leq \text{printable graphic ASCII characters} \leq 126$

# String variables (cont')

- Identify risks
  - Failure to process strings correctly that have **letters as their first character** (B3, Zb)
  - Failure to process **boundary characters** (A~) (the ASCII code of ~ is 126)
  - Mishandling of string values that have **nonletters** for the first character ([A])
  - Mishandling of characters that belong to the **extended ASCII set** (Aβ)

# Enumerated variable

- A variable which takes only a set of values/set of options

## Effects

---

Strikethrough

Small Caps

Double Strikethrough

All Caps

Superscript

Offset: 0% 

Equalize Character Height

Subscript

- Risks

- Failure to present font effect correctly that has a Superscript/Subscript/... option selected
- Mishandling of **no option** selected
- Mishandling of **multiple** options selected

# Multidimensional variables

- Multidimensional variable

There is more than one dimension of analyzing such a variable

- Software application example

To log in a system, the user must input the correct username and password. The username can have **five to fifteen characters**. Also, only **digits and lowercase characters** are allowed for the username.

- Username variable has two dimensions

Length and String

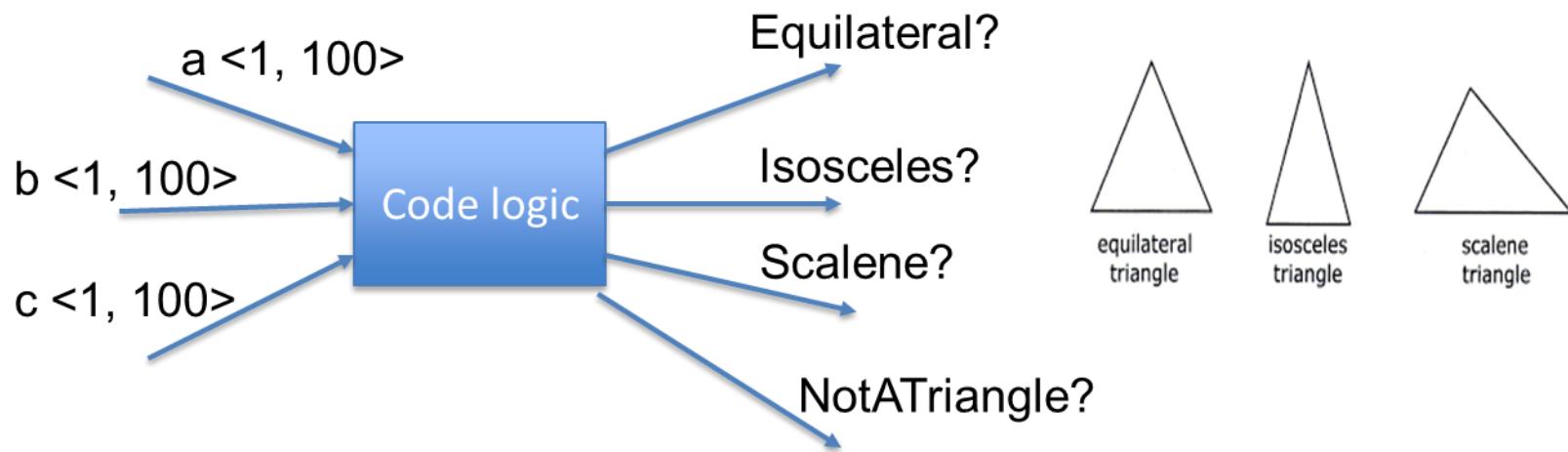
# Menti Exercise

Go to [www.menti.com](http://www.menti.com) and use the code **6438 5825**

# Outline

- Single variable
- **Multiple variables**
- Error guessing/Risk based
- Output coverage and testing

# Multivariable application example (1)



- Boundary value testing  $<1, 0, 1>$   $<1, 0, 2>$   $<1, 0, 50>$   $<1, 0, 99>$   $<1, 0, 100>$  ...  
 $<1, 1, 1>$   $<1, 1, 2>$   $<1, 1, 50>$   $<1, 1, 99>$   $<1, 1, 100>$  ...
- Equivalence class testing  $<1, 0, 1>$   $<1, 1, 0>$   $<101, 0, 1>$   $<101, 1, 0>$  ...
- Also need to analyze complex logical relationships of the input variables

# Decision table-based testing (cont')

<4, 1, 2>    <3, 3, 3>    <2, 2, 3>

Table 7.2 Decision Table for Triangle Problem

c1: a, b, c form a triangle?	F	T	T	T	T	T	T	T	T	T
c2: a = b?	—	T	T	T	T	F	F	F	F	F
c3: a = c?	—	T	T	F	F	T	T	F	F	F
c4: b = c?	—	T	F	T	F	T	F	T	F	F
a1: Not a triangle	X									
a2: Scalene										X
a3: Isosceles					X		X	X		
a4: Equilateral		X								
a5: Impossible			X	X		X				

\*

\* Software Testing, A craftsman's approach, 4<sup>th</sup> Edition

# Why test variable combination?

- All variables will **interact** as they are part of one functional unit and they have to unite to achieve the duty that the functional unit is designated for
- Hence most of these variables influence each other in some, or the other way
- Testing variables in combination may **reveal certain bugs that might not have been found by testing the variables in isolation**

# Multivariable application example (2)

- Online shopping system example
  - Parameters: Availability, Payment Method, Carrier, Delivery Schedule, Export Control

Availability	Payment	Carrier	Delivery Schedule	Export Control
Available	Credit	Mail	One Day	True
Not in Stock	Paypal	UPS	2-5 Working Days	False
Discontinued	Gift Voucher	Fedex	6-10 Working Days	
No Such Product			Over 10 Working Days	

$4 \times 3 \times 3 \times 4 \times 2 = 288$  combinations

# The challenge of testing the online shopping system example

- Suppose there is a bug, and **Credit** does not work well with **One Day delivery**
  - Any combination of inputs that include Credit and a One Day delivery will expose that bug
  - Interaction between 2 variables
- Suppose **Credit** does not work well with a **One Day delivery**, but only with **Fedex**
  - Any combination of inputs that include Credit, a One Day delivery, and not with Fedex will expose that bug
  - Interaction between 3 variables
- Do we really need to test all  **$4 \times 3 \times 3 \times 4 \times 2 = 288$**  combinations?

# Do we really need to test all combinations?

- The root cause analysis of many bugs shows they depend on the value of one variable (20%-68%)
- Most defects can be discovered in tests of the interactions between the values of two variables (65-97%)

Number of variables involved in triggering software faults\*

Vars	Medical Devices	Browser	Server	NASA GSFC	Network Security
1	66	29	42	68	20
2	97	76	70	93	65
3	99	95	89	98	90
4	100	97	96	100	98
5		99	96		100
6		100	100		

\*<http://csrc.nist.gov/groups/SNS/acts/ftfi.htm>

# What is n-way test?

- Given any  $n$  variables (out of all the variables) of a system, every combination of values of these  $n$  variables is covered in at least one test
- For example, an application has 5 variables
  - 2-way test
    - Every combination of values of any 2 variables (out of 5 variables) is covered in at least one test
  - 3-way test
    - Every combination of values of any 3 variables (out of 5 variables) is covered in at least one test

# What is n-way test (cont')?

- An application with 5 input variables A, B, C, D, E
  - A has values A1, A2
  - B has values B1, B2, B3,
  - C has values C1, C2
  - D has values D1, D2, D3, D4
  - E has values E1, E2
- Full combinations  $2 \times 3 \times 2 \times 4 \times 2 = 96$  test cases
- 2-way test (56 combinations)
  - Every combination of values of any 2 variables
  - AB, AC, AD, AE, BC, BD, BE, CD, CE, DE
  - AB (A1-B1, A1-B2, A1-B3, A2-B1, A2-B2, A2-B3)
  - AC (A1-C1, A1-C2, A2-C1, A2-C2)
  - ...

# All-Pairs combination testing

- 2-way testing
  - Every combination of values of 2 variables is covered in at least one test
- All-pair combination/pairwise testing is condensed 2-way testing
  - Many pairs can be tested at the same time in one combination

A1-B1, B1-C2, C2-D2 -> A1-B1-C2-D2

# All-pair combination/pairwise testing approaches

- In Parameter Order
- Variation of In Parameter Order

# In Parameter Order

- Initialization phase
- Horizontal growth phase
- Vertical growth phase

# Example of In Parameter Order

- Variables
  - X (True, False)
  - Y (0, 5)
  - Z (P, Q, R)

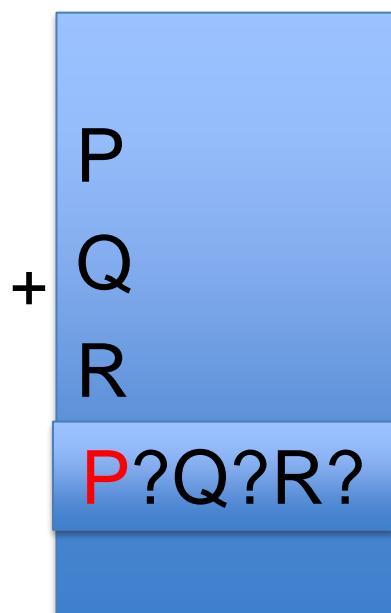
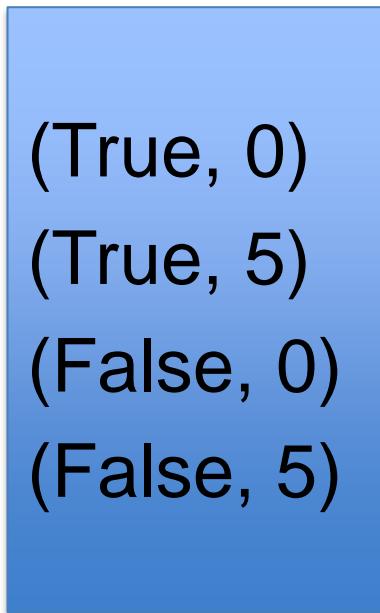
# In Parameter Order - Initialization

- X (True, False)
- Y (0, 5)
- T= (True, 0)  
(True, 5)  
(False, 0)  
(False, 5)

Find any two variables and generate test cases

# In Parameter Order - Horizontal growth

- X (True, False)
  - Y (0, 5)
  - Z (P, Q, R)
- Add one more variable in each step to generate more combination
  - Remove duplications



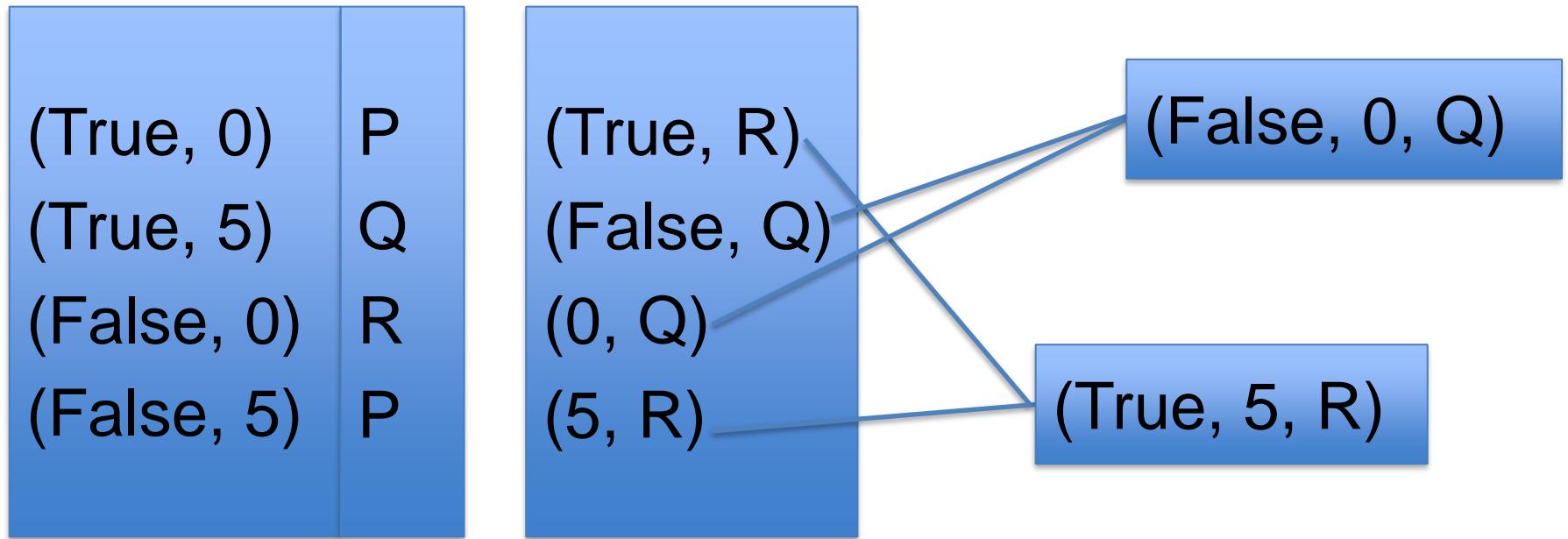
+

vs.

All 2-way test combinations

~~(True, P) (True, Q), (True, R)~~  
~~(False, P), (False, Q), (False, R)~~  
~~(0, P), (0, Q), (0, R)~~  
~~(5, P), (5, Q), (5, R)~~

# In Parameter Order - Vertical growth



Merge short combinations to longer ones

# A Variation of In Parameter Order

- Initialization phase
  - Order variables based on the number of their possible values
  - Generate test cases by full combinations of the first two variables
- Vertical growth phase
  - Add other variables one-by-one to generate a combination
  - **Re-order certain variable values if needed**
- Horizontal growth phase
  - If re-order cannot generate enough new combinations
  - **Extend horizontally to add possibility of using re-ordering to generate new combinations**

# A Variation of In Parameter Order - Example

- A car ordering application \*
- Variables
  - Order category (Buy, Sell)
  - Location (Oslo, Trondheim)
  - Car brand (BMW, Audi, Mercedes)
  - Registration numbers (Valid, Invalid)
  - Order type (E-Booking, In-store)
  - Order time (Working hours, Non-working hours)

\*[www.softwaretestinghelp.com/what-is-pairwise-testing/](http://www.softwaretestinghelp.com/what-is-pairwise-testing/)

# Variation of In Parameter Order – Initialization (reorder variables)

- Order variables based on their number of values
- The results after ordering
  - **Car brand (BMW, Audi, Mercedes)**
  - Order category (Buy, Sell)
  - Location (Oslo, Trondheim)
  - Registration numbers (Valid, Invalid)
  - Order type (E-Booking, In-store)
  - Order time (Working hours, Non-working hours)

# Vertical growth phase – iteration 1

Car Brand	Order category	Location	Registration number	Order type	Order time
BMW	Buy				
BMW	Sell				
Audi	Buy				
Audi	Sell				
Mercedes	Buy				
Mercedes	Sell				

# Vertical growth phase – iteration 2

- Add other variables one-by-one to generate a combination
- Re-order certain variable values if needed

Car brand	Order category	Location	Registration number	Order type	Order time
BMW	Buy	Oslo			
BMW	Sell	Trondheim			
Audi	Buy	Oslo			
Audi	Sell	Trondheim			
Mercedes	Buy	Oslo			
Mercedes	Sell	Trondheim			



# Vertical growth phase – iteration 2 (cont')

- After re-ordering

Car brand	Order category	Location	Registration number	Order type	Order time
BMW	Buy	Oslo			
BMW	Sell	Trondheim			
Audi	Buy	Trondheim			
Audi	Sell	Oslo			
Mercedes	Buy	Oslo			
Mercedes	Sell	Trondheim			

# Vertical growth phase – iteration 3

- Add other variables one-by-one to generate a combination
- Re-order certain variable values if needed

Car brand	Order category	Location	Registration number	Order type	Order time
BMW	Buy	Oslo	Valid		
BMW	Sell	Trondheim	Invalid		
Audi	Buy	Trondheim	Valid		
Audi	Sell	Oslo	Invalid		
Mercedes	Buy	Oslo	Valid		
Mercedes	Sell	Trondheim	Invalid		

# Vertical growth phase – iteration 3 (cont')

- After re-ordering

Car brand	Order category	Location	Registration number	Order type	Order time
BMW	Buy	Oslo	Valid		
BMW	Sell	Trondheim	Invalid		
Audi	Buy	Trondheim	Valid		
Audi	Sell	Oslo	Invalid		
Mercedes	Buy	Oslo	Invalid		
Mercedes	Sell	Trondheim	Valid		

# Vertical growth phase – iteration 4

- Add other variables one-by-one to generate a combination
- Re-order certain variable values if needed

Car brand	Order category	Location	Registration number	Order type	Order time
BMW	Buy	Oslo	Valid	In Store	
BMW	Sell	Trondheim	Invalid	E-booking	
Audi	Buy	Trondheim	Valid	E-booking	
Audi	Sell	Oslo	Invalid	In Store	
Mercedes	Buy	Oslo	Invalid	E-booking	
Mercedes	Sell	Trondheim	Valid	In Store	

# Vertical growth phase – iteration 5

If re-order cannot generate enough new combinations

Car brand	Order category	Location	Registration number	Order type	Order time
BMW	Buy	Oslo	Valid	In Store	Working hours
BMW	Sell	Trondheim	Invalid	E-booking	Non-working hours
Audi	Buy	Trondheim	Valid	E-booking	Working hours
Audi	Sell	Oslo	Invalid	In Store	Non-working hours
Mercedes	Buy	Oslo	Invalid	E-booking	Working hours
Mercedes	Sell	Trondheim	Valid	In Store	Non-working hours

# Horizontal growth

Extend horizontally to add possibility of using re-ordering to generate new combinations

Car brand	Order category	Location	Registration number	Order type	Order time
BMW	Buy	Oslo	Valid	In Store	Working hours
BMW	Sell	Trondheim	Invalid	E-booking	Non-working hours
	Buy				Non-working hours
Audi	Buy	Trondheim	Valid	E-booking	Working hours
Audi	Sell	Oslo	Invalid	In Store	Non-working hours
	Sell				Working hours
Mercedes	Buy	Oslo	Invalid	E-booking	Working hours
Mercedes	Sell	Trondheim	Valid	In Store	Non-working hours

# All-pairs combination testing tools

- [ACTS](#) – ‘Advanced Combinatorial Testing System’, provided by NIST, an agency of the US Government
- [VPTag](#) free Pairwise Testing Tool

# Outline

- Single variable
- Multiple variables
- Error guessing/Risk based
- Output coverage and testing

# Error guessing / Risk-based

- Based on experience
  - Different types of faults
  - Probability of the faults
  - Consequence of the faults
  - ...

# Bach's risk-based testing\*

- Bach's heuristics is based on his experience as a tester
- Two risk analysis approach
  - Inside-out: What can go wrong there?
  - Outside-in: What things are associate with this kind of risk?
- Based on this experience he has identified
  - A generic risk list – things that are important to test
  - A risk catalogue – things that often go wrong to a specific domain

\*<http://www.satisfice.com/articles/hrbt.pdf>

# Bach's generic risk list

- Complex – large, intricate or convoluted
- New – no past history in this product
- Changed – anything that has been tampered with or “improved”
- Upstream dependency – a failure here will cascade through the system
- Downstream dependency – sensitive to failure in the rest of the system
- Critical – a failure here will cause serious damage

# Bach's generic risk list (cont')

- Precise – must meet the requirements exactly
- Popular – will be used a lot
- Strategic – of special importance to the users or customers
- Third-party – developed outside the project
- Distributed – spread out over time or space but still required to work together
- Buggy – known to have a lot of problems
- Recent failure – has a recent history of failures

# Bach's risk catalogs example

- Wrong files installed
  - Temporary files not cleaned up
  - Old files not cleaned up after upgrade
  - Unneeded file is installed
  - Needed file not installed
  - Correct file installed in the wrong space
- Files clobbered
  - Old file replaces newer file
  - User data file clobbered during upgrade
- ...

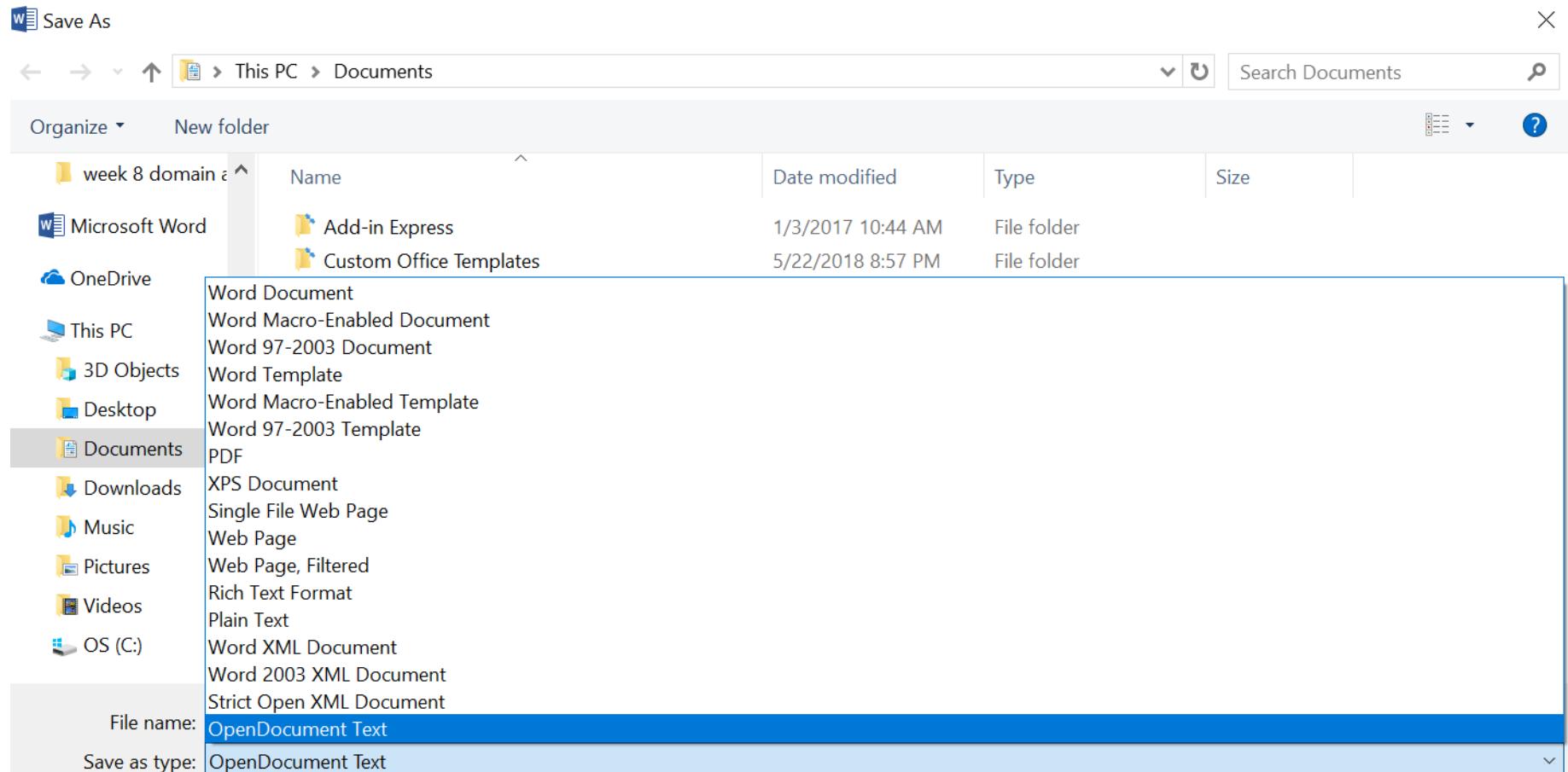
# Outline

- Single variable
- Multiple variables
- Random testing
- Error guessing/Risk based
- Output coverage and testing

# Output coverage

- All the coverage types that we have looked at so far have been related to input data
- It is also possible to define coverage based on output data
- Uncovered outputs help us define the new test cases

# Output coverage (cont')



# Output coverage (cont")

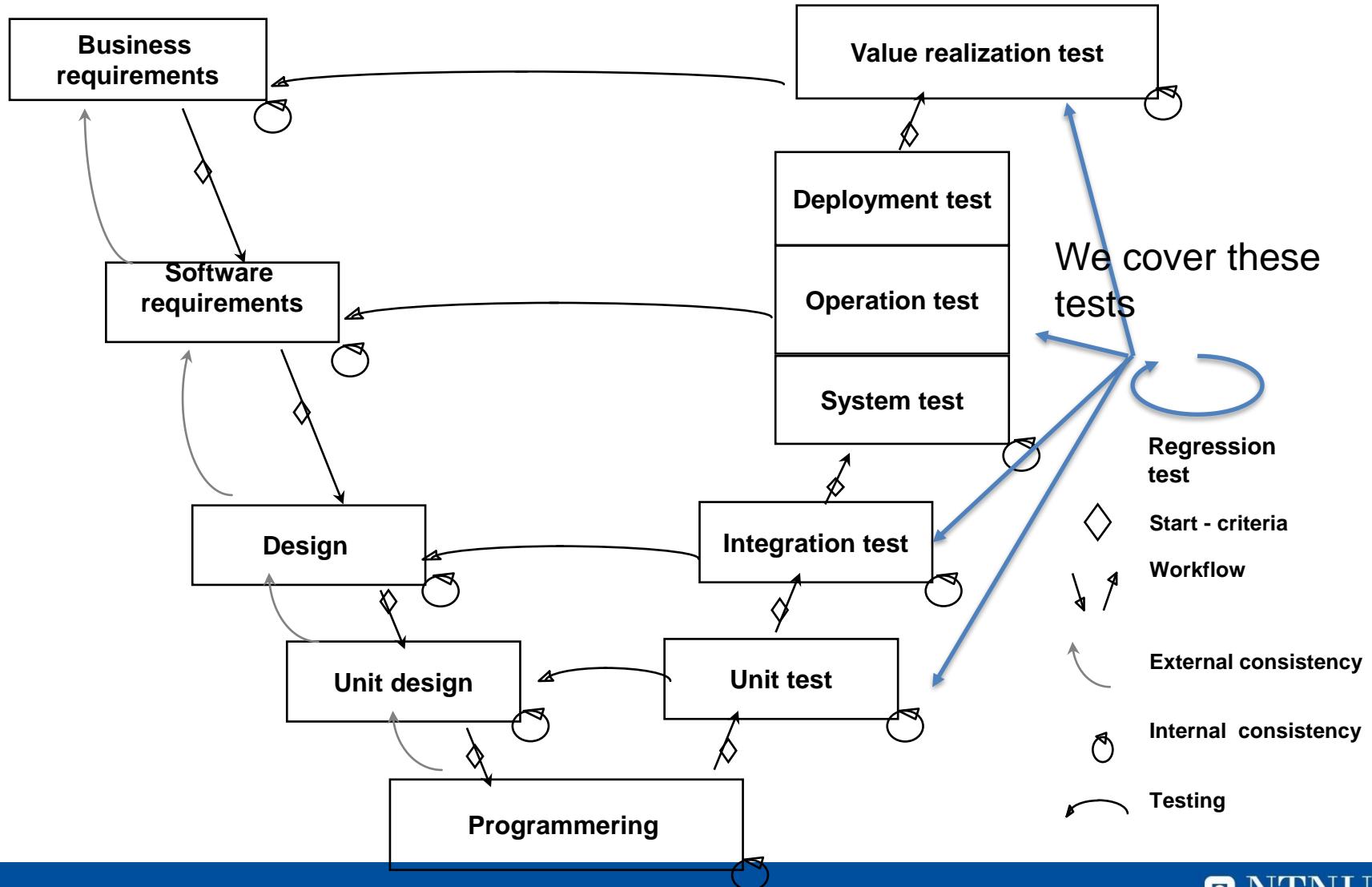
- The main challenge with using output coverage measure is that output can be defined at several levels of details
  - An account summary
  - An account summary for a special type of customer
  - An account summary for a special event

# Summary

- We studied
  - How to identify the domains of **input & output** variables
  - How to derive function test cases based on single and multiple domain variables
- Can be applied to
  - Unit test
  - Integration test
  - System test
  - Acceptance test

# **Integration, system, acceptance, and regression testing**

# V-model



# Integration testing

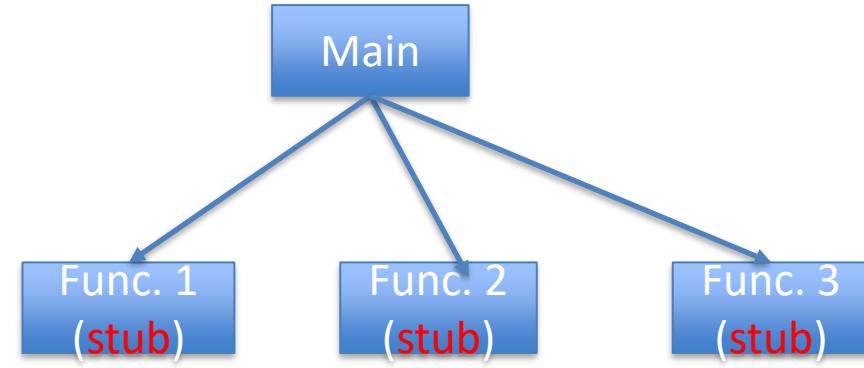
- Focus on testing interfaces between components
- Not as well understood as unit testing and system testing
- Usually poorly done
- Strategies
  - Decomposition-based
  - Call graph-based/interface matrix
  - Path-based

# Decomposition-based integration

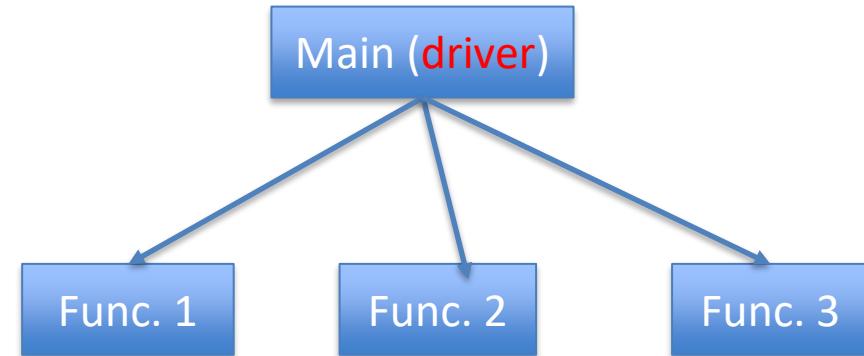
- Based on functional decomposition tree
- Need to know function dependencies between components
- Based on the order of integration testing, can be classified into four strategies
  - Top-down
  - Bottom-up
  - Sandwich
  - Big bang

# Decomposition-based integration (cont')

- Top-down
  - Begin with main
  - Use “**stubs**” to simulate called functions
  - Replace stubs with real functions one by one



- Bottom-up
  - Begin with leaves
  - Use “**drivers**” to emulate functions call the leaves
  - Replace “drivers” with real function later



- Sandwich
- Big bang

# Pros and cons of decomposition-based integration

- Pros
  - Incremental and intuitive
  - Easy fault isolation
- Cons
  - Need “stub” or “driver”

# Call graph-based/interface matrix-based integration

- Based on the call graph/interface matrix of components
- Use actual components rather than “stubs” or “drivers”
- Two strategies
  - Pairwise integration
  - Neighborhoods integration

# Pair-wise integration

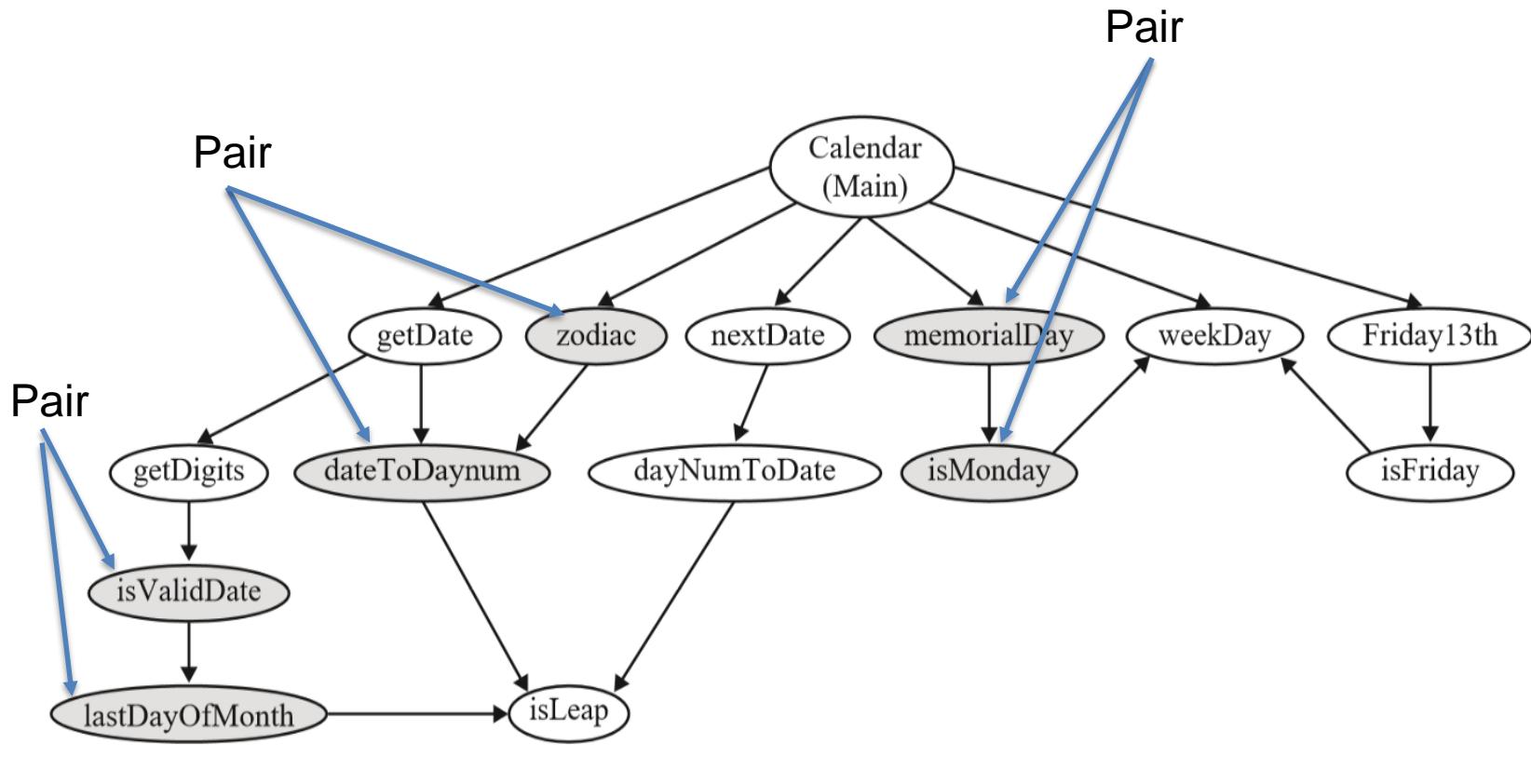


Figure 13.9 Three pairs for pairwise integration.

\* Software Testing, A craftsman's approach, 4<sup>th</sup> Edition

# Interface matrix

	Sys 1	Sys 2	Sys 3	Sys 4
Sys 1	X	X		
Sys 2	X	X		
Sys 3			X	X
Sys 4			X	X

# Neighborhoods integration

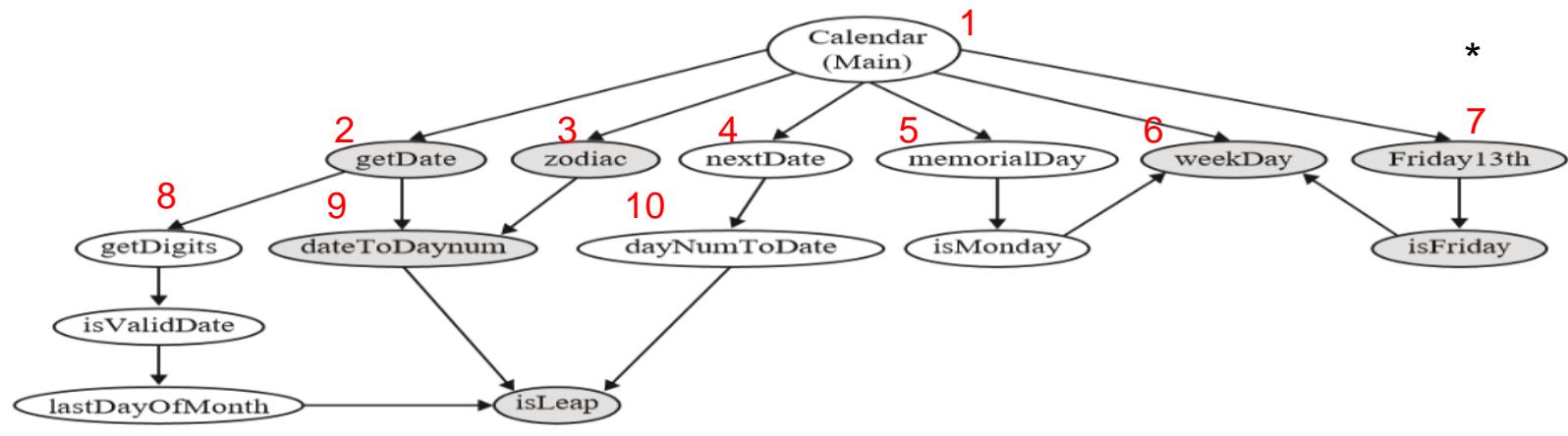


Figure 13.10 Three neighborhoods (of radius 1) for neighborhood integration.

Table 13.1 Neighborhoods of Radius 1 in Calendar Call Graph

Neighborhoods in Calendar Program Call Graph			
<i>Node</i>	<i>Unit Name</i>	<i>Predecessors</i>	<i>Successors</i>
1	Calendar (Main)	(None)	2, 3, 4, 5, 6, 7
2	getDate	1	8, 9
3	zodiac	1	9

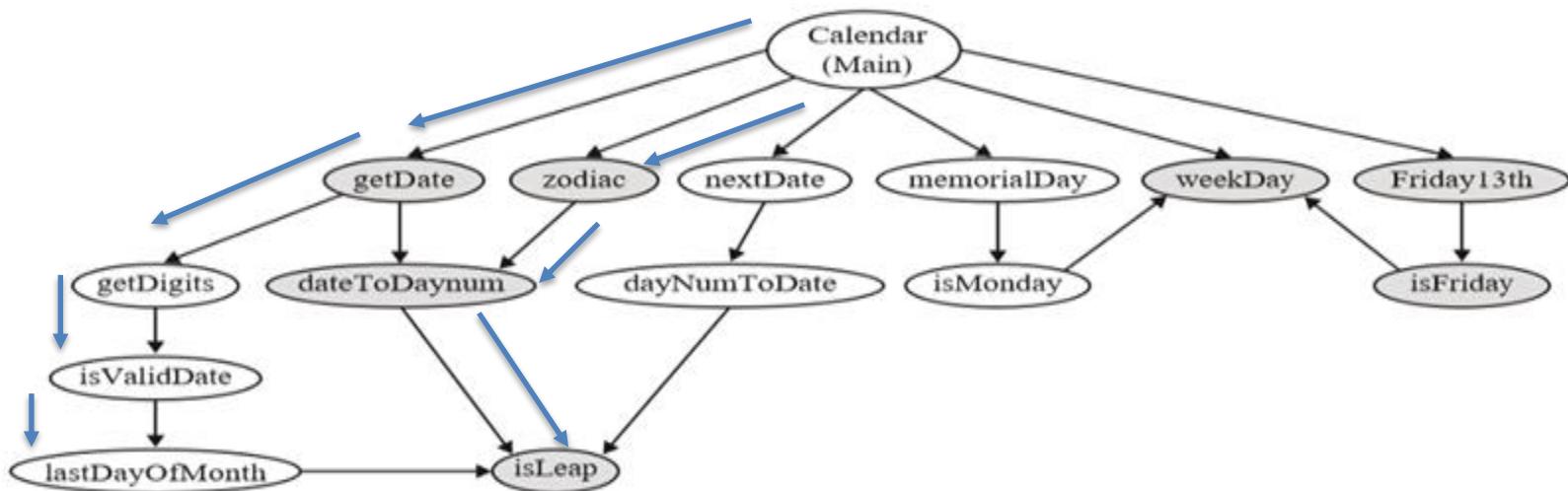
\* Software Testing, A craftsman's approach, 4<sup>th</sup> Edition

# Pros and cons of call graph-based integration

- Pros
  - Do not need “stub” or “driver”
- Cons
  - Limited to local integration

# Path-based integration

- Not focusing on testing interfaces among separately developed and tested unit
- Rather focus on interactions among these units



# Path-based integration (cont')

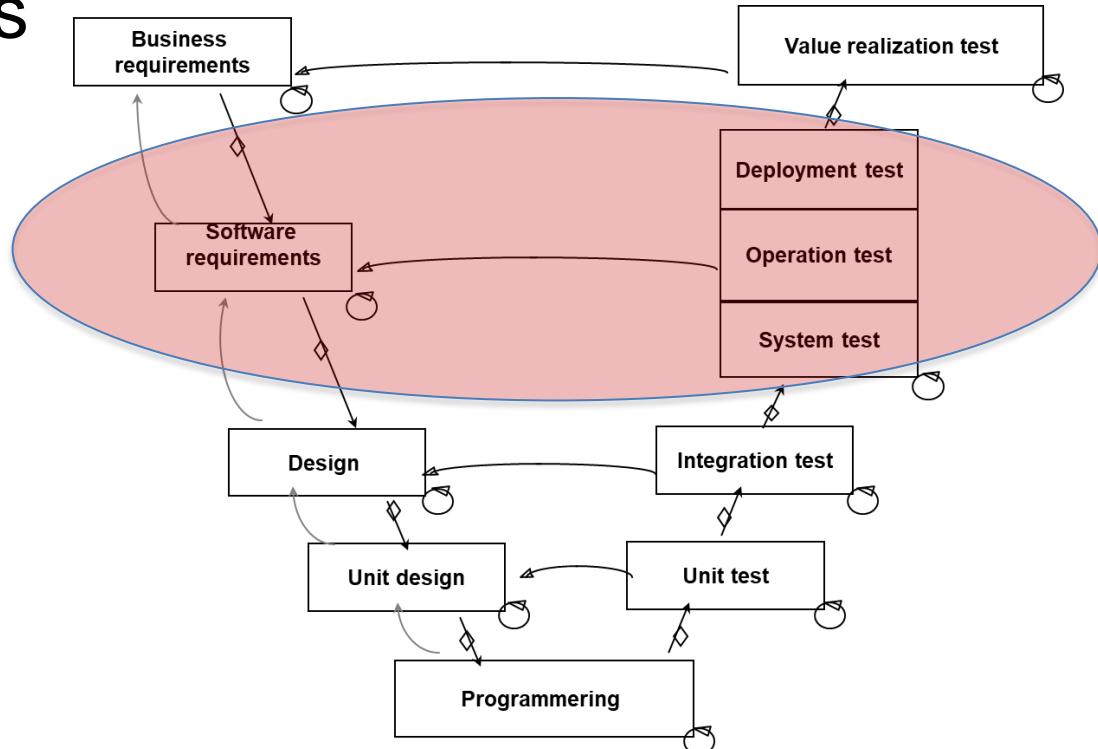
- Like enlarged unit testing
  - The node is not a statement
  - The node is a component/test unit
  - The edge is message transferred between components

# Pros and cons of path-based integration

- Pros
  - Test more global and complex integrations
  - Closely coupled with actual system behavior
- Cons
  - Difficult fault isolation
  - Extra effort is need to identify message path

# System tests categories

- Type of system tests
  - Functionality
  - Reliability
  - Usability
  - Performance
  - Robustness
  - Scalability
  - Stress
  - Load and stability
  - ...

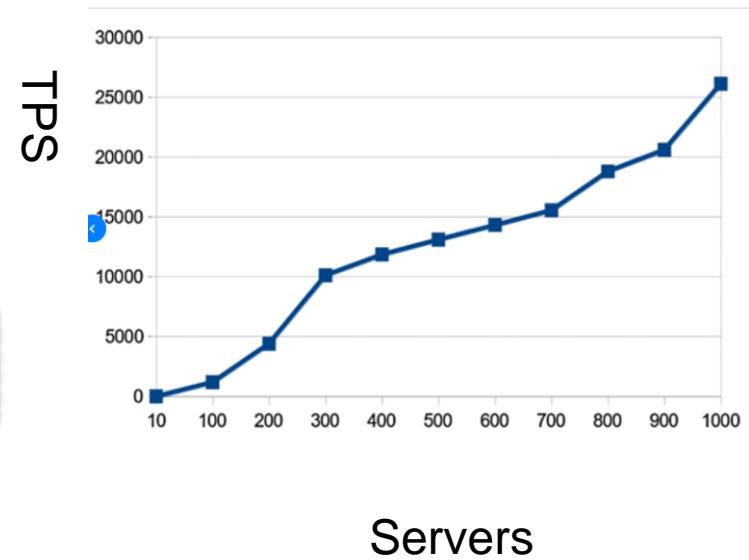
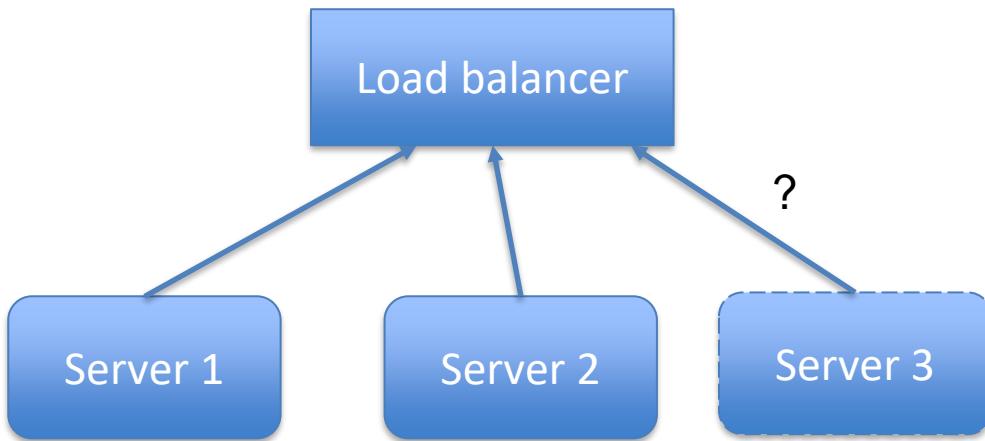


# Robustness testing

- How sensitive is a system to erroneous inputs or changes in the operational environment?
  - Stupid/uncommon inputs (like testing special values in boundary testing)
  - Failures from other systems
  - Degraded node
  - Etc.

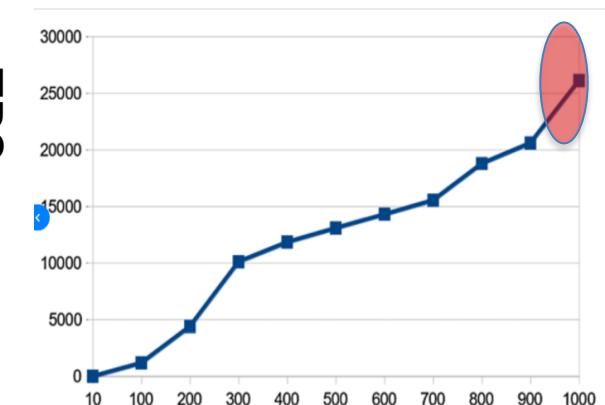
# Scalability testing

- To identify how well the system can scale, i.e., the magnitude of demand that can be placed on the system while continuing to meet performance requirements



# Stress testing

- It can ensure the system can perform acceptably under the worse-case condition
- The system is deliberately stressed by pushing it to and beyond its specific limits **for a while**
- For testing
  - Buffer allocation and memory carving
  - Network bandwidth



CULTURE

# Olympics ticket system crashes

Domestic ticket sales in China are halted after 8 million hits and 200,000 orders per second overwhelm the booking system.

BY SUZANNE TINDAL | NOVEMBER 1, 2007 6:18 AM PDT

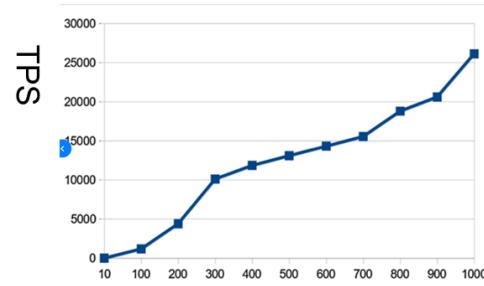


# Load and stability testing

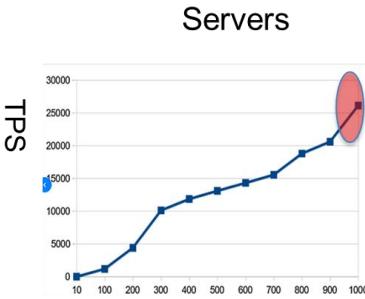
- Important for providing 24x7 services
- System and applications that run for months are likely to
  - Slow down
  - Encounter functionality problems
  - Silently failover
  - Crash altogether
- Ensure the system remains stable for **a long period of time** under full load

# Scalability vs. Stress vs. Load and stability testing

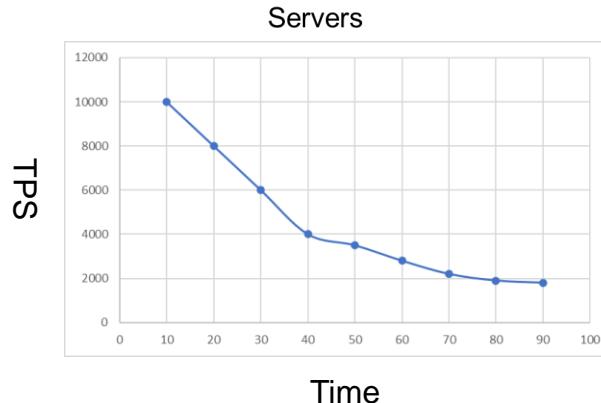
- Scalability



- Stress



- Load and stability



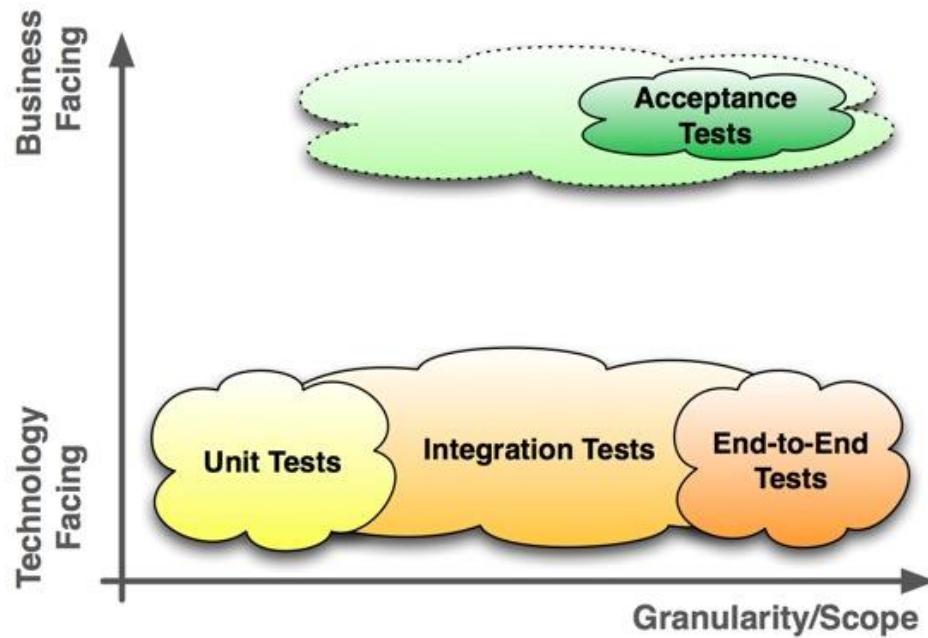
# Using Apache JMeter to run load and performance testing



<https://youtu.be/yNnbW2n9s8E>

# Acceptance testing

- ISTQB : **(user) acceptance testing**: Formal testing with respect to user needs, requirements, and business processes conducted to determine whether or not a system satisfies the **acceptance criteria** and to enable the **user, customers** or other **authorized entity** to determine whether or not to accept the system.



# Typical forms of acceptance testing



# User acceptance testing

- Functions are correct?
- Test basis
  - User/business requirements
  - System requirements
  - Use cases
  - Business processes
  - Risk analysis reports

# Operational acceptance test

- Read to operate?
  - Backup facilities
  - Procedures for disaster recovery
  - Training or manual for end-users
  - Maintenance procedures and manual
  - Security procedures

# Contract and regulation acceptance testing

- Test against contract
- Test against regulations
  - Governmental regulation
  - Legal standards
  - Safety standards

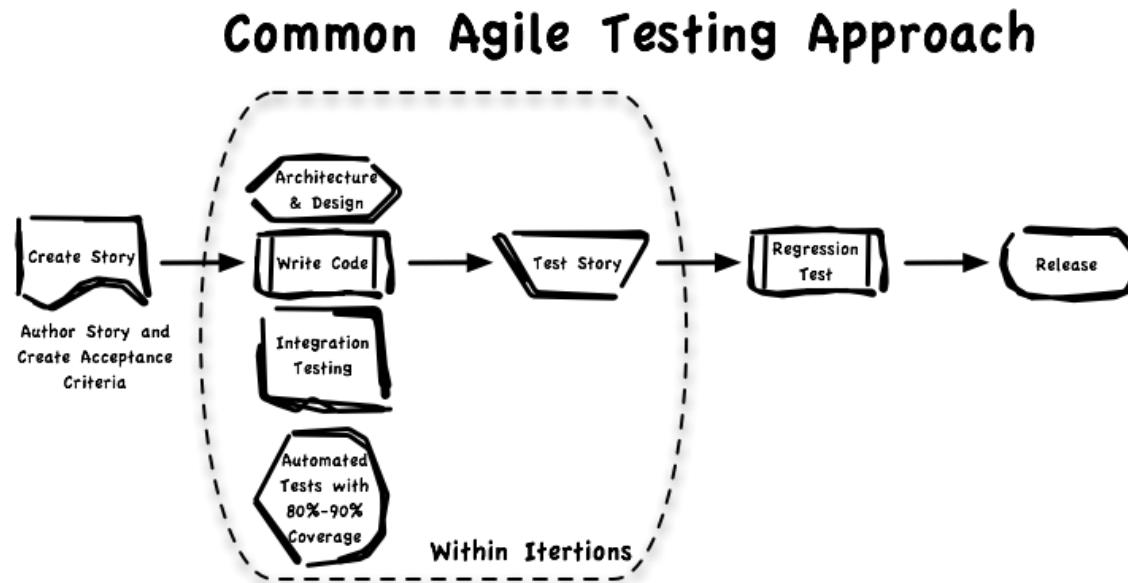
# Alpha and beta testing

- Alpha
  - At developers' sites
  - By internal staff
  - Before it is released to external customers
- Beta (field testing)
  - At customers' sites
  - Before the system is released to other customers



# Acceptance testing in agile

- The terms "functional test", "acceptance test" and "customer test" are used more or less interchangeably.
- Referring to user stories



# Outline

- Testing approaches
  - Integration tests
  - System tests
  - Acceptance tests

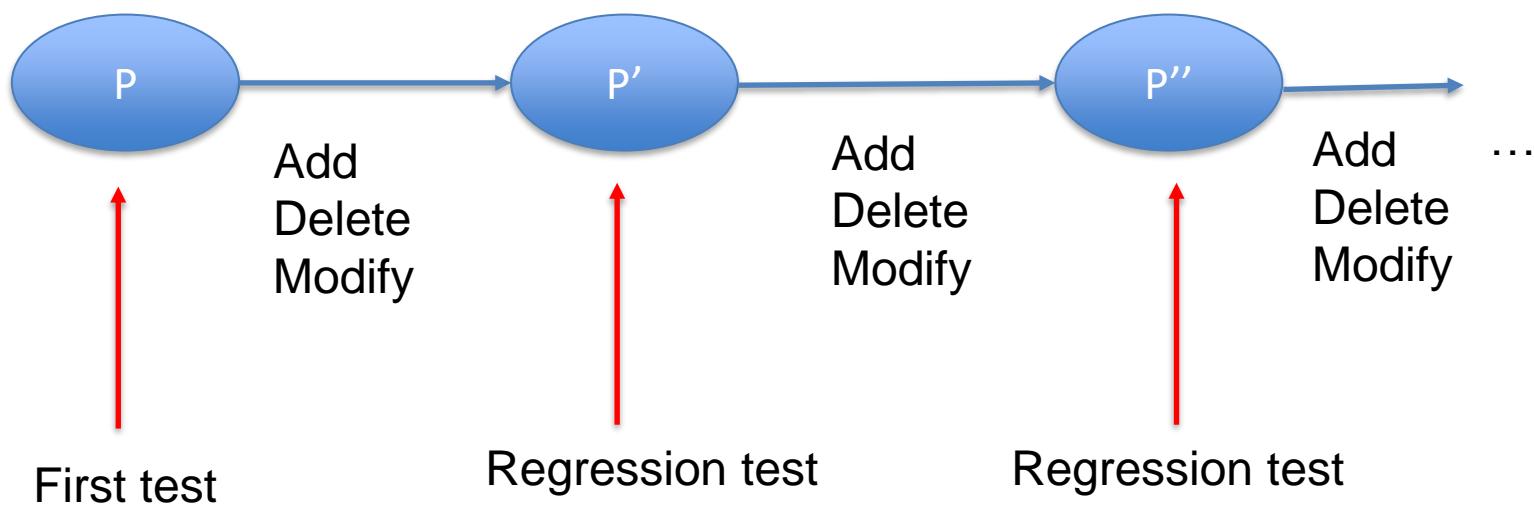
➤ Regression tests

# Regression testing

- Basic regression testing concepts
- Regression test case selection
- Regression minimization
- Regression test case prioritization

# What is regression testing?

- Constitute the vast majority of testing effort in many software development projects

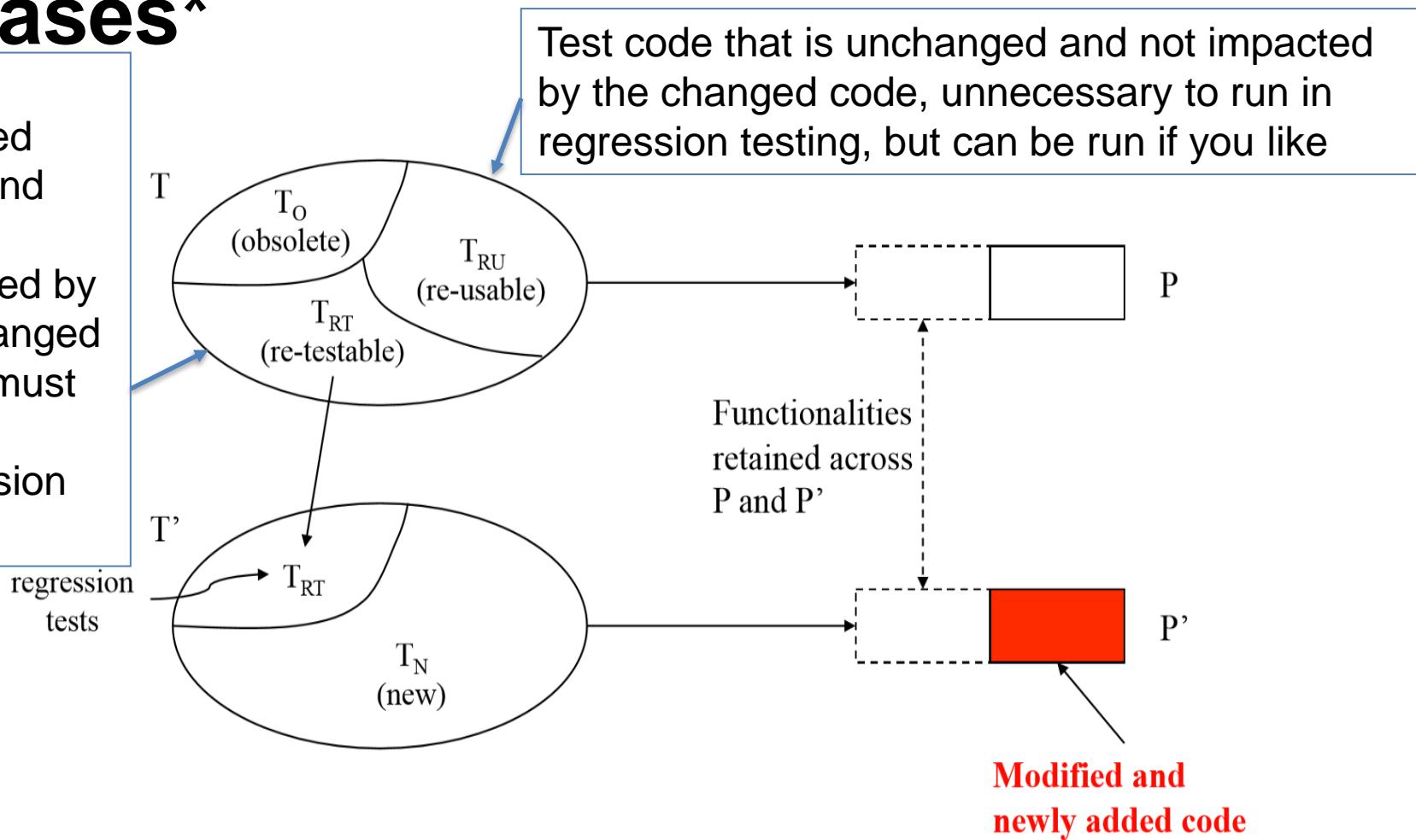


# Two types of regression testing

- Corrective regression testing
  - No requirements change
  - Modified code behaves correctly
  - Unchanged code continues to behave correctly
- Progressive regression testing
  - Requirements change
  - Newly added or modified code
  - Unchanged code continues to behave correctly

# Overview of first & regression test cases\*

Test changed code and code impacted by the changed code, must run in regression testing



\* <https://www.uio.no/studier/emner/matnat/ifi/INF4290/v11/undervisningsmateriale/INF4290-RegTest.pdf>

# Regression testing processes

- Test revalidation
  - Are test cases obsolete?
- (Regression) test selection
  - Verify changed and impacted code (corrective)
  - Verify new structure and requirements (progressive)
- Test minimization
  - Remove redundant test cases
- Test prioritization
  - Rank test cases and run them according to available resources

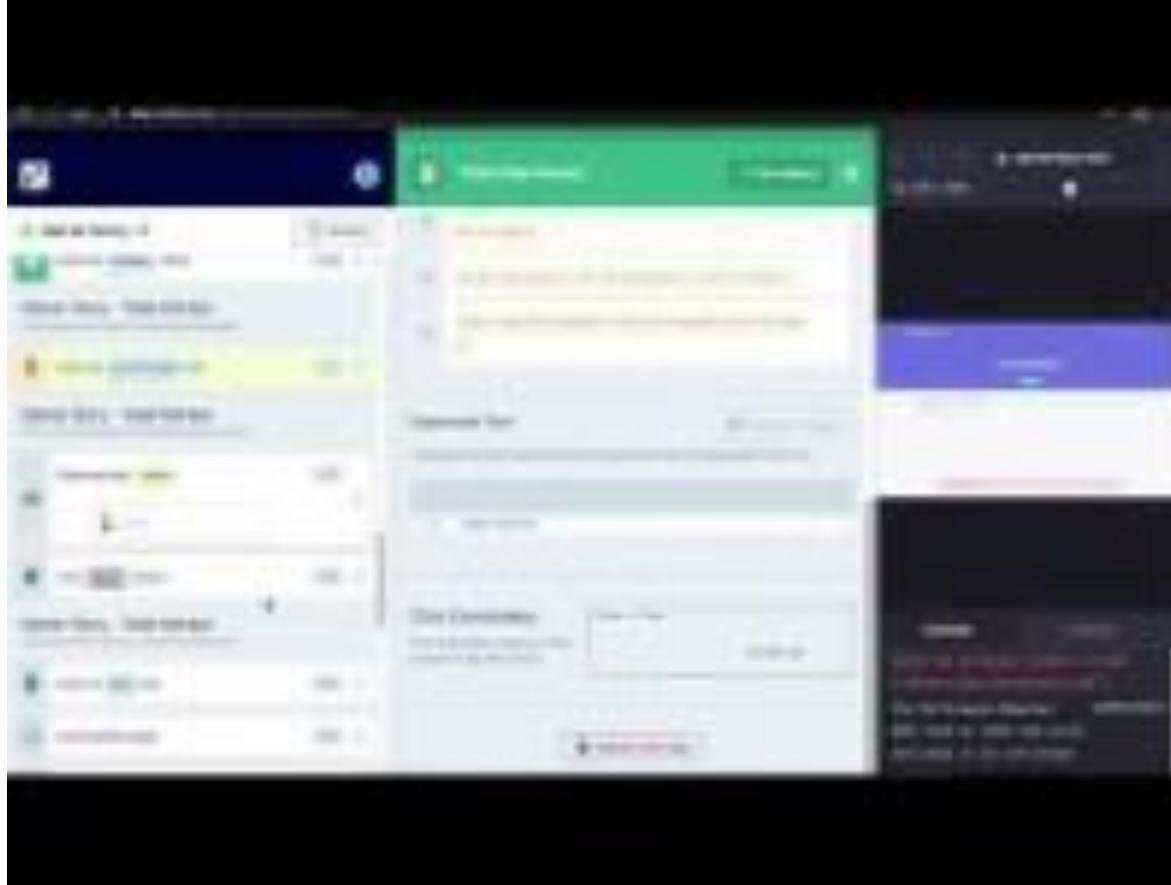
# Test revalidation

- How can a test case be obsolete?
  - Their input/output relation is no longer correct due to **changes in specification**
  - They no longer test what they were designed to test due to **modifications to the program**
  - They are “structural” test cases that no longer contribute to **structural coverage** of the program

# (Regression) test selection

- We now consider only **corrective regression testing**
- Strategies of (regression) test selection
  - Retest all
    - Can be costly, unless test execution is highly automated
  - Random selection
    - Better than no test
    - Hard to ensure coverage of the changed code
  - Selecting **modification traversing** tests
    - “**Safe**” regression test selection technique is preferred

# Automated regression testing example



<https://youtu.be/2jAy9cdbLaE>

# Safe regression test selection

A technique that **does not discard any test** that will **traverse a modified or impacted statement** is known as “safe” regression test selection technique

- Assume program **P** has been tested by test set **T** against specification **S**
- Assume **P** is modified to **P'** to fix some bugs (i.e., specification **S** is not changed)

Question:

- What information is needed in order to select a **safe regression test subset** from **T** to verify **P'**?

# General (regression) test selection process

- Establish **traces** between P and T
  - Execute P with test cases
  - Record program entities executed when running tests
- Compare P with P' to find **differences**
  - Identify program entities changed
- Select test cases from T that **traverse** the **changed program and impacted entities**

# General (regression) test selection process (cont')

- Based on program entities **traced** and **compared**, the test selection methods can be classified into
    - Dynamic slicing based approach (statements)
    - Graph-walk approach (nodes in control flow graph/program dependence graph)
    - Firewall approach (OO classes, COTS components, etc. )
    - ...
- 

# Graph-walk approach example

Code P

```
int M (int x, int y){  
    int z;  
    if (x<y)  
        z = f1 (x, y);  
    else  
        z = f2(x, y);  
    return z;  
}
```

Code P'

```
int f1( int a, int b){  
    if ((a+1) == b)  
        return a*a;  
    else  
        return b*b; }
```

```
int f2( int a, int b){  
    if (a == (b+1))  
        return b*b;  
    else  
        return a*a; }
```

If((a-1) == b)

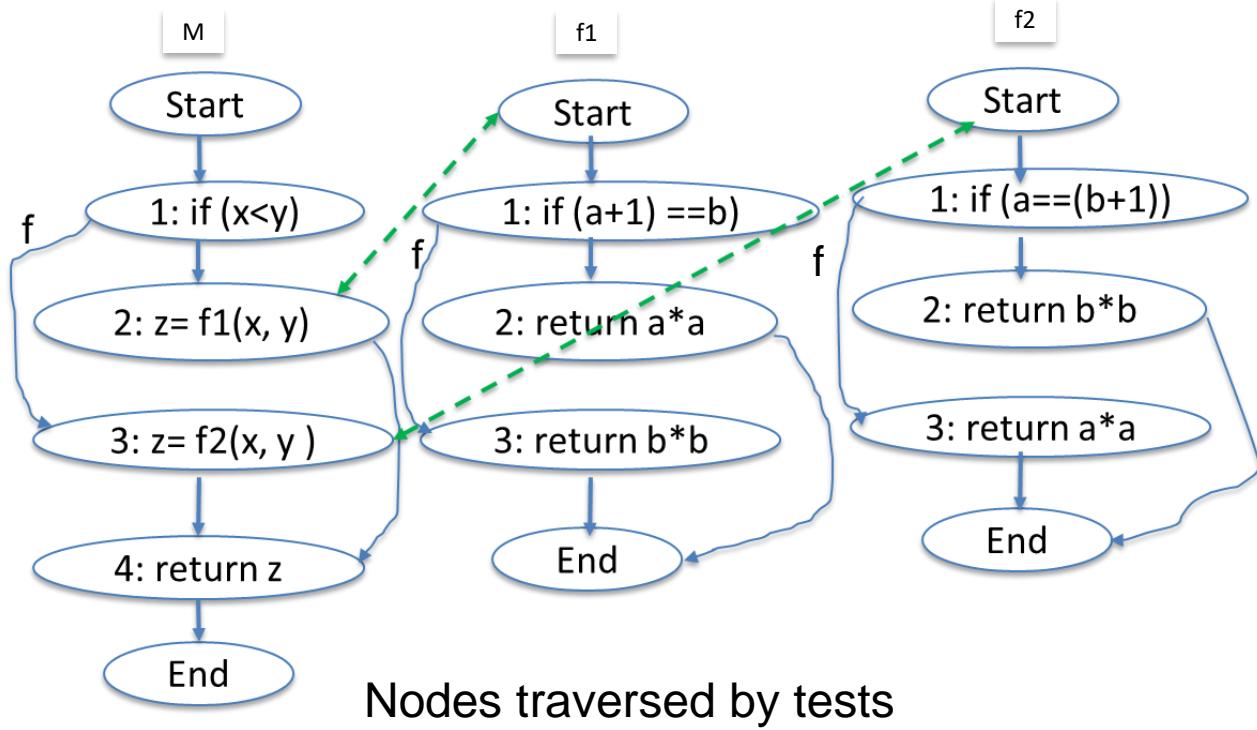
Test set T of P

t1: <x = 1, y = 2>  
t2 : <x = 1, y = 3>  
t3 : <x = 3, y = 1>

Which test cases to be included in the safe subset to test P'?

# Graph-walk approach example – step 1

- Establish trace between test case and CFG (control flow graph) nodes



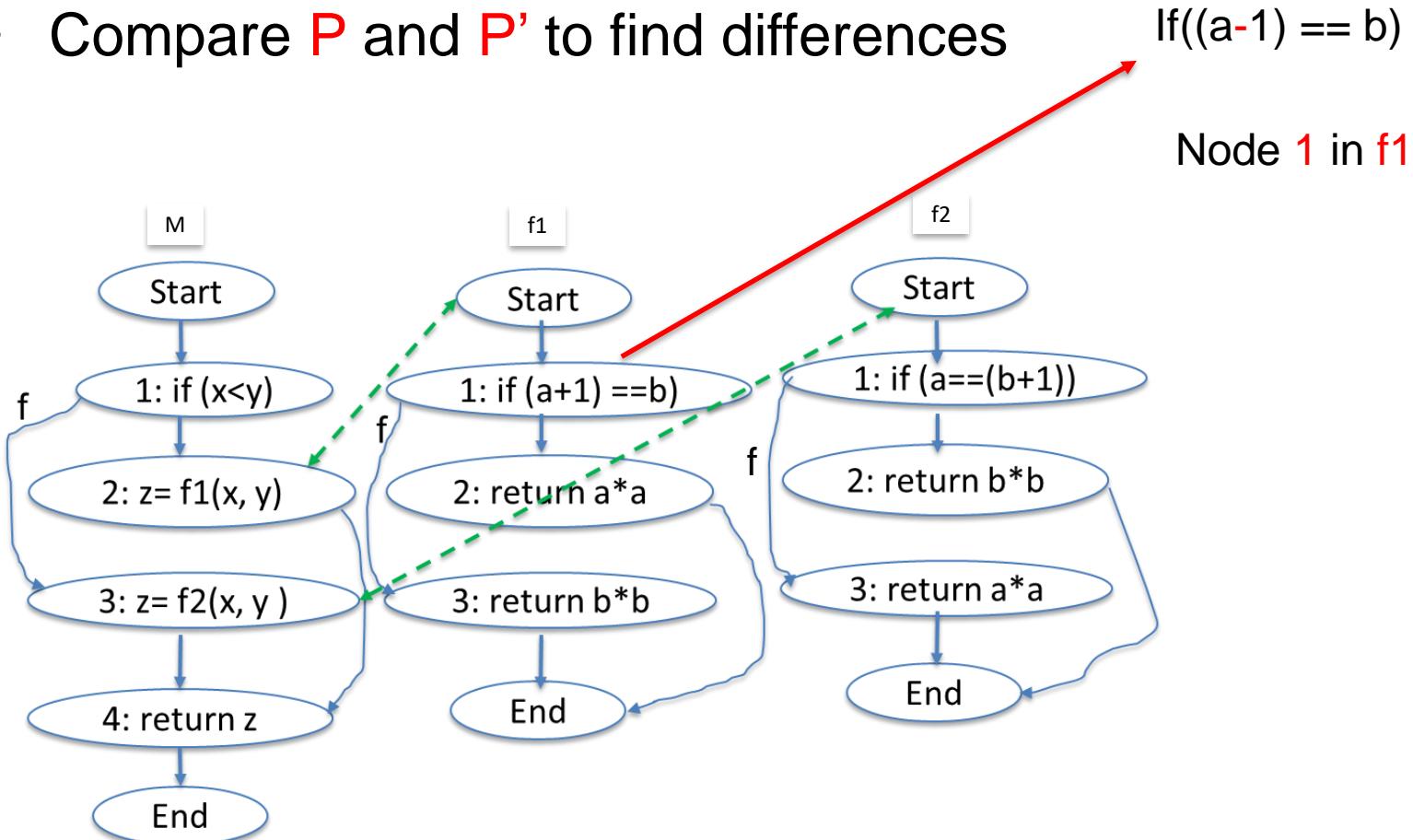
Test set T

- $t1 : \langle x = 1, y = 2 \rangle$
- $t2 : \langle x = 1, y = 3 \rangle$
- $t3 : \langle x = 3, y = 1 \rangle$

Functions	Nodes			
	1	2	3	4
M	t1, t2, t3	t1, t2	t3	t1, t2, t3
f1	t1, t2	t1	t2	-
f2	t3	None	t3	-

# Graph-walk approach example – step 2

- Compare  $P$  and  $P'$  to find differences



# Graph-walk approach example – step 3

- Select test cases from T that traverse the changed CFG nodes

		Set of tests that traverse a certain node			
Functions		Nodes			
	1	2	3	4	
M	t1, t2, t3	t1, t2	t3	t1,t2, t3	
f1	t1, t2	t1	t2	-	
f2	t3	None	t3	-	

Test set T

- t1:  $x = 1; y = 2$
- t2 :  $x = 1, y = 3$
- t3 :  $x = 3, y = 1$

A technique that **does not discard any test that will traverse a modified or impacted statement** is known as “safe” regression test selection technique.

- Here, we did not cover the impacted statement
- Test oracles of the tests may need to be changed

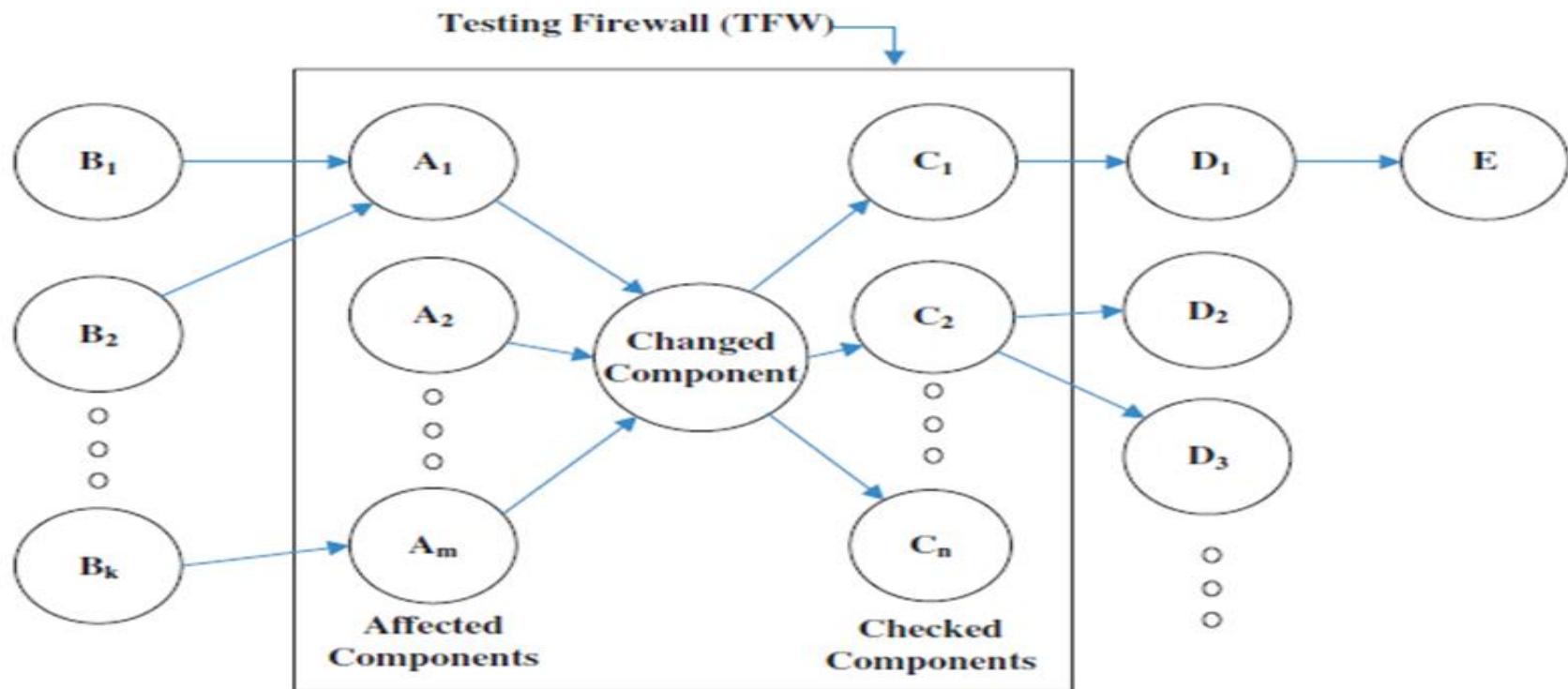
# General (regression) test selection process (cont')

- Based on program entities **traced** and **compared**, the test selection methods can be classified into
  - Dynamic slicing based approach (statements)
  - Graph-walk approach (nodes in control flow graph/program dependence graph)
  - Firewall approach (OO classes, COTS components, etc. )
  - ...

# Firewall approach

What is a firewall?

- A *firewall* separates the classes that depend on the class that is changed from the rest of the classes



# Firewall approach (cont')

Why use a firewall?

- The ***firewall*** approach is based on the **first-level dependencies** of modified components
- The most severe effects are in those components that send messages to the changed component or receive messages from the changed component

# Process for determining “Firewall” in OO systems

1. Identify classes changed
2. Inheritance? Then **descendants** of the changed classes
3. Classes send messages to or receive messages from  
the changed class

# One empirical study of using firewall approach [1]

- System studied
  - Distributed component-based J2EE sys. owned by Swedbank
  - 1.2M line of code from 27 000 classes
  - 2 new releases of the system
- Steps 1: Coverage analysis and establishing trace
  - Instrumentation of Java byte code and run-time collection of **coverage data** during original test
  - Used a tool called Emma<sup>[2]</sup>

# One empirical study of using firewall approach [1] (Cont')

## Step 2: Find classes in firewall

- Identify changes – produced MD5 signatures for each compiled class file. If code changes, class signature will be different.
- Own tool

## Step 3: Extracting dependency between classes

- Dependency finder [3]

## Step 3: Select test cases that traverse classes in firewall

# Another empirical study of using firewall approach [4]

- Studied an OO telecommunication software system at ABB
- Compared two firewall approaches
  - One level from changed component
  - More than one level from the changed component, depending on their logical dependencies

# Another empirical study of using firewall approach [4] (Cont')

- Observation
  - Extended firewall approach finds more faults than classical firewall approach, but more costly to run
- Recommendations
  - Routine incremental changes (one level approach)
  - Major release (> one level approach)

# Test minimization

- To reduce redundancies in the **safe subset**
- If every entity covered by **t2** is a subset of entities covered by another test case **t1**, then we remove **t2**
- The entities could be
  - Statement, CFG nodes, functions, etc.

# Test minimization example

- $t_1$  covers the following statement
- $t_2$  also covers the same statement
- Then, we can remove  $t_2$

$$(a < b) \parallel (a < c)$$

# Risk of test minimization

- Test minimization is **risky** and is not necessarily safe
  - It depends on the modifications (from P to P'), the faults, the entities used

$$(a < b) \parallel (a < c)$$

t2 may test the  $(a < c)$  part and t1 may test the  $(a < b)$  part

If the t2 is removed because this statement is covered by t1, then, we will miss the opportunity to test  $(a < c)$

# Test prioritization

- Different from test minimization
- Test prioritization
  - Is not going to remove any test cases from the **safe subset**
  - Is to rank regression tests based on some criteria
- The goal is to reveal faults early

# Test prioritization approaches

- Unlike test prioritization of the first test
  - **More information available**
  - Fault finding effectiveness
  - Test coverage
  - Cost of running the test
- Prioritization strategies [5]
  - Coverage based (high coverage)
  - Cost-aware based (low cost to run, high fault-finding probability)
  - ...

# Summary

- We studied the following testing approaches
  - Integration
  - System
  - Acceptance
- We studied regression test
  - Selection
  - Minimization
  - Prioritization

# References

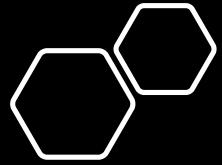
- [1] White L., Abdullah K. A firewall approach for the regression testing of object-oriented software. Proc. of the Software Quality Week, 1997.
- [2] <http://emma.sourceforge.net/>
- [3] Skoglund M. and Runeson P. A Case study of The Class Firewall Regression Test Selection Technique on Large Scale Distributed Software System. Proc. of empirical software engineering conf. 2015.
- [4] White L. Jaber K. Robinson B. and Rajich V. Extended firewall for regression testing: an experience report. Journal of software maintenance and evolution. Vol. 20. 2008
- [5] Yoo S., Harman M. Regression testing minimization, selection and prioritization: a survey, Software testing, verification, and reliability, 2007.

# System Testing Planning

Theory and Practice  
Chapter 12  
System Test Planning and Automation



**Software Testing and QA** Theory and Practice (Chapter 12: System Test Planning and Automation)



# AGENDA

1

Testing Planning

2

Testing Execution

# Why do we have to plan tests?

- The purpose is to get ready and organized for test execution
- Without a plan it is highly unlikely that
  - the desired level of system testing is performed within the stipulated time and without overusing resources
  - Expected Costs and deadlines will be reached
- Planning for system testing IS part of the overall planning for a software project!
- It provides the framework, scope, resources, schedule, and budget for the system testing part of the project.

# A System Test Plan

- Provides guidance for the executive management to support the test project
- Establishes the foundation of the system testing part of the overall software project
- Provides assurance of test coverage by creating a requirement traceability matrix
- Outlines an orderly schedule of events and test milestones that are tracked
- Specifies the personnel, financial, equipment, and facility resources required to support the system testing part of a software project

# Planning the Test (Outline)

---

**Introduction and Feature Description**

---

**Assumptions**

---

**Test Approach**

---

**Test Suite Structure**

---

**Test Environment**

---

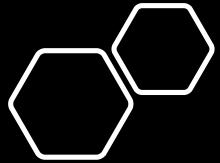
**Test Execution Strategy**

---

**Test Effort Estimation**

---

**Scheduling and Milestones**



# Introduction

The *introduction* section of the system test plan includes:

- Test project name
- Revision history
- Terminology and definitions
- Name of the approvers and the date of approval
- References
- Summary of the rest of the test plan

The *feature description* section summarizes the system features that will be tested during the execution of this plan.

# Planning the Test

---

**Introduction and Feature Description**

---

**Assumptions**

---

**Test Approach**

---

**Test Suite Structure**

---

**Test Environment**

---

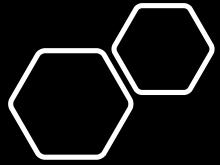
**Test Execution Strategy**

---

**Test Effort Estimation**

---

**Scheduling and Milestones**



# Assumptions

- The *assumptions* section describes the areas for which test cases will not be designed
- Examples:
  - The necessary equipments to carry out scalability testing may not be available
  - It may not be possible to procure third-party equipments in time to conduct interoperability testing
  - It may not be possible to conduct compliance test for regulatory bodies and environment tests in the laboratory

# Planning the Test

---

**Introduction and Feature Description**

---

**Assumptions**

---

**Test Approach**

---

**Test Suite Structure**

---

**Test Environment**

---

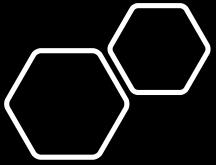
**Test Execution Strategy**

---

**Test Effort Estimation**

---

**Scheduling and Milestones**



# Test Approach



Issues discovered by customers that were not caught during system testing in the past project



Outstanding issues that need to be tested differently need to be discussed here



Discuss test automation strategy



Test cases from the database that can be re-used in this test plan

# Planning the Test

---

Introduction and Feature Description

---

Assumptions

---

Test Approach

---

Test Suite Structure

---

Test Environment

---

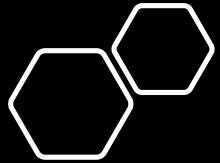
Test Execution Strategy

---

Test Effort Estimation

---

Scheduling and Milestones



# Test Suite Structure

- Detail *test groups* and *subgroups* based on the test categories identified in the *test approach* section
- Test objectives are created for each test group and subgroups based on the system requirements and functional specification documents
- Identification of test objectives provides a clue to the total number of test cases needs to be developed
- A traceability matrix is generated to make an association between requirements and test objectives to provide the test coverage

# Planning the Test

---

Introduction and Feature Description

---

Assumptions

---

Test Approach

---

Test Suite Structure

---

Test Environment

---

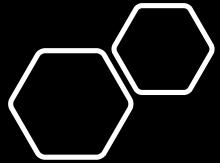
Test Execution Strategy

---

Test Effort Estimation

---

Scheduling and Milestones



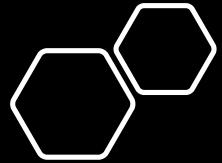
# Test Environment

Multiple test environments are constructed in practice

Test activities can not interfere with development activities

To run security tests you need an environment that can be put down.

To run scalability tests one may need more resources than to run functionality tests



# Test Environment

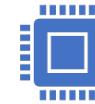
## Test Bed Preparations



Get Information about the customer deployment architecture including hardware, software and their manufacturers



List of third-party products to be integrated with the SUT.



Third party test tools to be used to monitor, simulate, and /or generate real traffic.



Hardware equipment necessary to support special features specified in the requirements, for example backup, some network switches etc.



Analysis of the non-functional requirements for checking what is necessary to perform these tests



List of small but necessary networking, cables hubs, needed for the test.

# Planning the Test

---

Introduction and Feature Description

---

Assumptions

---

Test Approach

---

Test Suite Structure

---

Test Environment

---

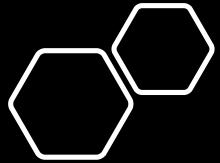
Test Execution Strategy

---

Test Effort Estimation

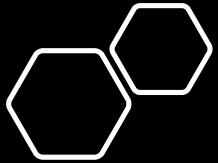
---

Scheduling and Milestones



# Test Execution Strategy

- It's a game plan that addresses the following concerns:
  - How many times are the tests cases executed and when?
  - What does one do with the failed test cases?
  - Which test cases groups are blockers?
  - What happens when too many test cases fail?
  - In which order are the test cases executed?

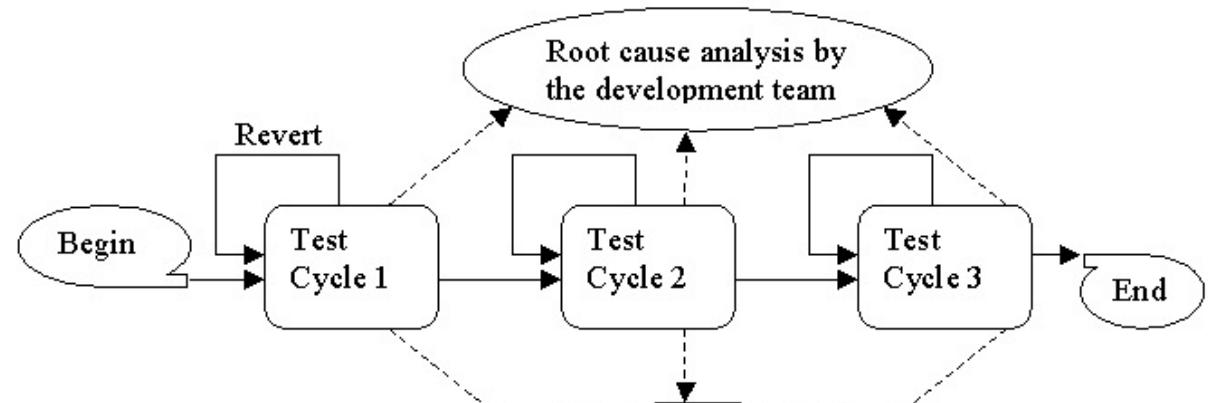


# Test Execution Strategy

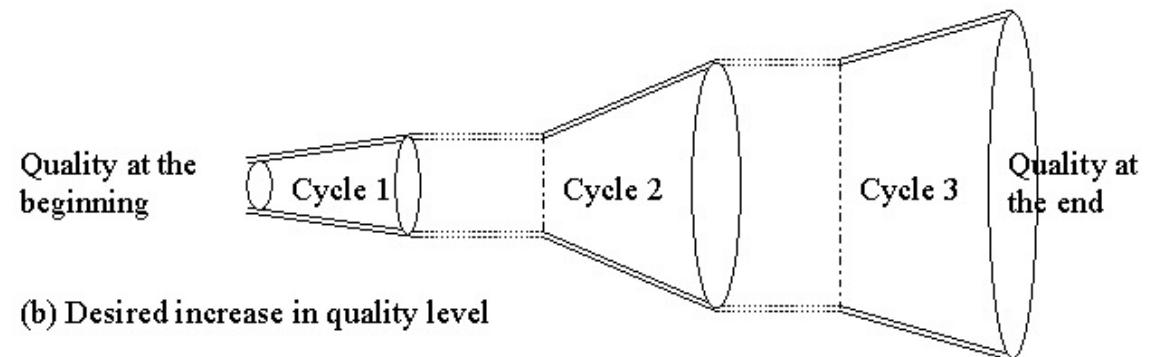
- The processes of system test execution, defect detection, and fixing defects are intricately intertwined
  - Some test cases cannot be executed unless certain defects are detected and fixed
  - A programmer may introduce new defects while fixing one defect, which may not be successful
  - The development team releases a new build for system testing by working on a subset of the reported defects, rather than all the defects
  - It is a waste of resources to run the entire test set T on a build if too many test cases fail
  - When the system is more stable it may be wise to start prioritizing the tests for regression tests

An effective and efficient execution strategy must take into account the characteristics of the test execution, defect detection and defect removal!

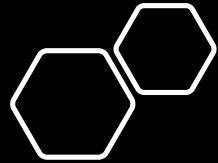
# A Multi-Cycle System Test Strategy



(a) Progress of system testing in terms of test cycles



(b) Desired increase in quality level



## Characterization of Test Cycles

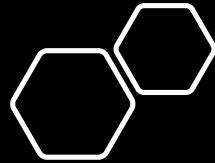
Goals and Assumptions

Test execution

Revert and Extension criteria

Actions

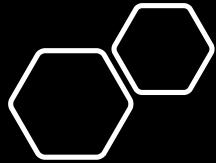
Exit criteria



## Characterization of Test Cycles

### Goals and Assumptions

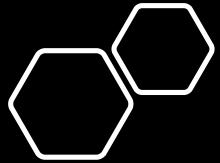
- **System test team sets its own goals to be achieved in each test cycle**
- **Goals are specified in terms of the number of test cases to pass in a cycle;**
- **Assumptions specify some “tolerance level for the test”**



## Characterization of Test Cycles

### Test Execution

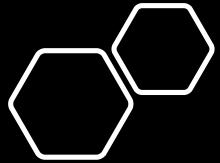
- **Test Cases are executed in multiple test environments as specified in the test plan.**
- Prioritization of the test execution changes between test cycles.
- **Some good practices for prioritization of test cases**
  - **Test cases that exercise basic functionalities have higher priority than the rest in the first test cycle**
  - **Test cases that have failed in one test cycle have a higher priority in the following test cycle**
  - **Test cases in certain groups have higher priority than others**



# Characterization of Test Cycles

## Revert and Extension criteria

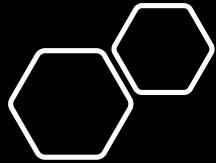
- The conditions for prematurely terminating a test cycle and for extending a test cycle must be precisely stated
- It may not be useful to continue a test cycle if it is found that a software is of poor quality
- Often a test cycle is extended due to various reasons
  - A need to re-execute all the test cases in a particular test group because a large fraction of the test cases within the group failed
  - A significantly large number of new test cases were added while test execution was in progress



# Characterization of Test Cycles

## Actions (Examples)

- Too many test cases may fail during a test cycle
  - The development team needs to be alerted
  - The developers should take action in the form of root cause analysis (RCA)
  - Corrective actions are taken by updating the design specification, reviewing the code, and adding new test cases to the unit and integration test plan
- The system test team has to design a large number of new test cases during a test cycle
  - Tester have then to see why this happened.
    - Mismatches between what the tester expected and what was delivered?
    - Poor planning of the tests ?
    - Different expectations of the tests objectives for that cycle?



## Characterization of Test Cycles

### Exit Criteria

- An exit criterion specifies the termination of a test cycle
  - Mere execution of all the test cases in a test suite does not mean that a cycle has completed maybe there is a need to add more “quality” criteria:
    - a percentage of passed/not passed
    - Number of environment that is acceptable
  - Example of exit criteria
    - 95% of test cases passed and all the known defects are in CLOSED state

# Planning the Test

---

Introduction and Feature Description

---

Assumptions

---

Test Approach

---

Test Suite Structure

---

Test Environment

---

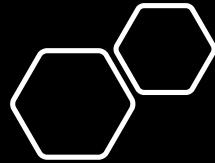
Test Execution Strategy

---

Test Effort Estimation

---

Scheduling and Milestones



# Test Effort Estimation

## Cost of the test and the time to complete the test

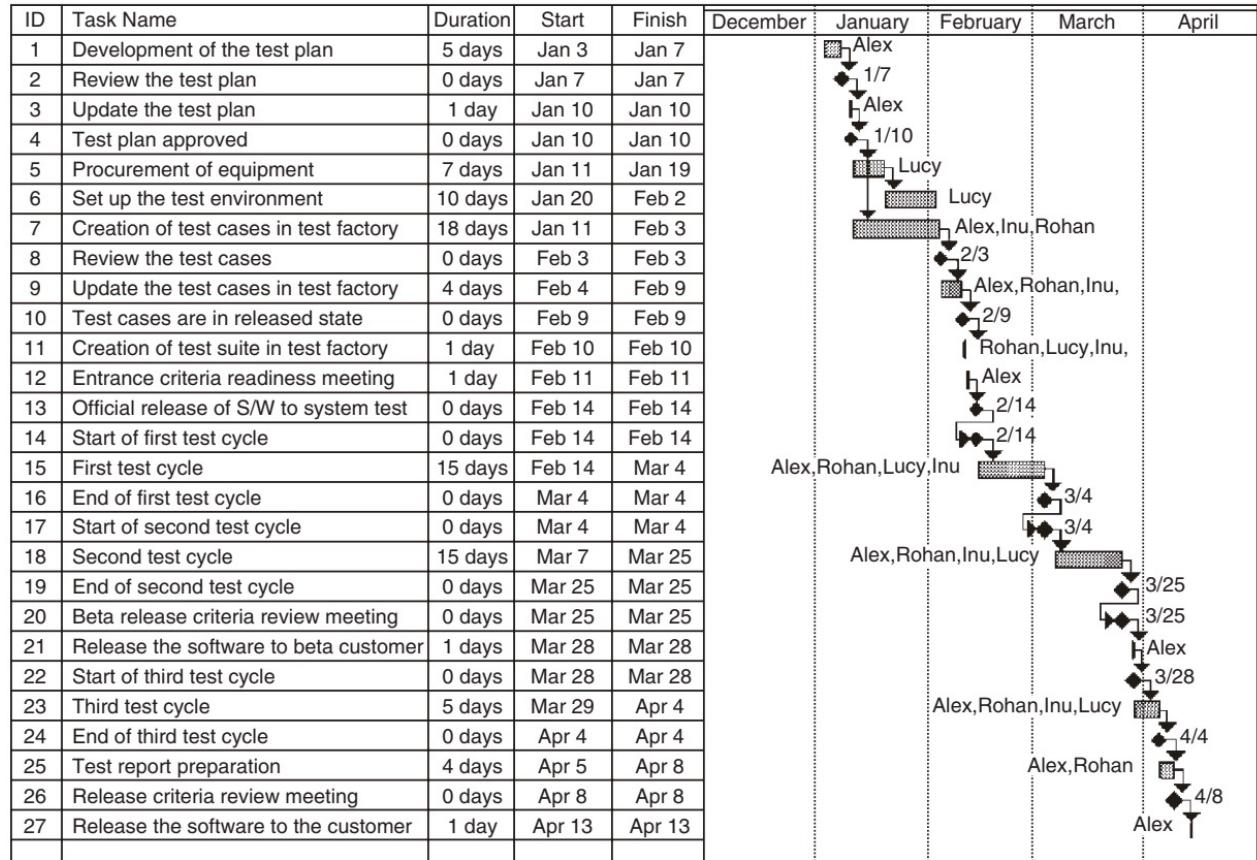
- The number of test cases to be designed for the software project based on:
  - Test Group Category
  - Function Points
- Effort required to create a detailed test case
- Effort required to execute a test case and analyze the results of execution
- Effort needed to create the test environments
- Effort needed to report the defects, reproduce the failures and follow up with development team

# Test Effort Estimation

**TABLE 12.6 Test Effort Estimation for FR-ATM PVC Service Interworking**

Test (Sub)Groups	Estimated Number of Test Cases	Person-Day to Create	Person-Day to Execute
Configuration	40	4	6
Monitoring	30	4	4
Traffic Management	3	2	3
Congestion	4	2	4
SIWF Translation Mode Mapping	12	4	4
Alarm	4	1	1
Interface	9	5	3
Robustness	44(4 + 40)	4	6
Performance	18	6	10
Stress	2	1.5	2
Load and stability	2	1.5	2
Regression	150(60 + 90)	0	15
<b>Total</b>	<b>318</b>	<b>35</b>	<b>60</b>

# Scheduling and Test Milestones



# Planning the Test

---

Introduction and Feature Description

---

Assumptions

---

Test Approach

---

Test Suite Structure

---

Test Environment

---

Test Execution Strategy

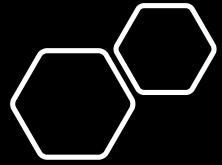
---

Test Effort Estimation

---

Scheduling and Milestones

---



# AGENDA

1

Testing Planning

2

Testing Execution

# System Test Execution

Theory and Practice  
Chapter 13  
System Test Execution



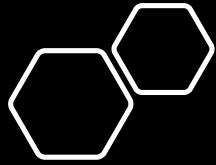
**Software Testing and QA** Theory and Practice (Chapter 12: System Test Planning and Automation)

# Outline

- Lifecycle of a Defect
- Metrics for Monitoring Test execution
- Metrics for Monitoring Defect Reports
- Defect Analysis Techniques:
- Measuring Test Effectiveness
- System Test Report

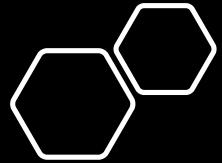
# Test Execution

- It is important to monitor the processes of test execution, tracking defects and measure test effectiveness.
  - There is a pressure to meet a tight schedule close to the delivery date;
  - There is a need to discover most of the defects before delivering the product;
  - It is essential to verify that defect fixes are working and have not resulted in new defects

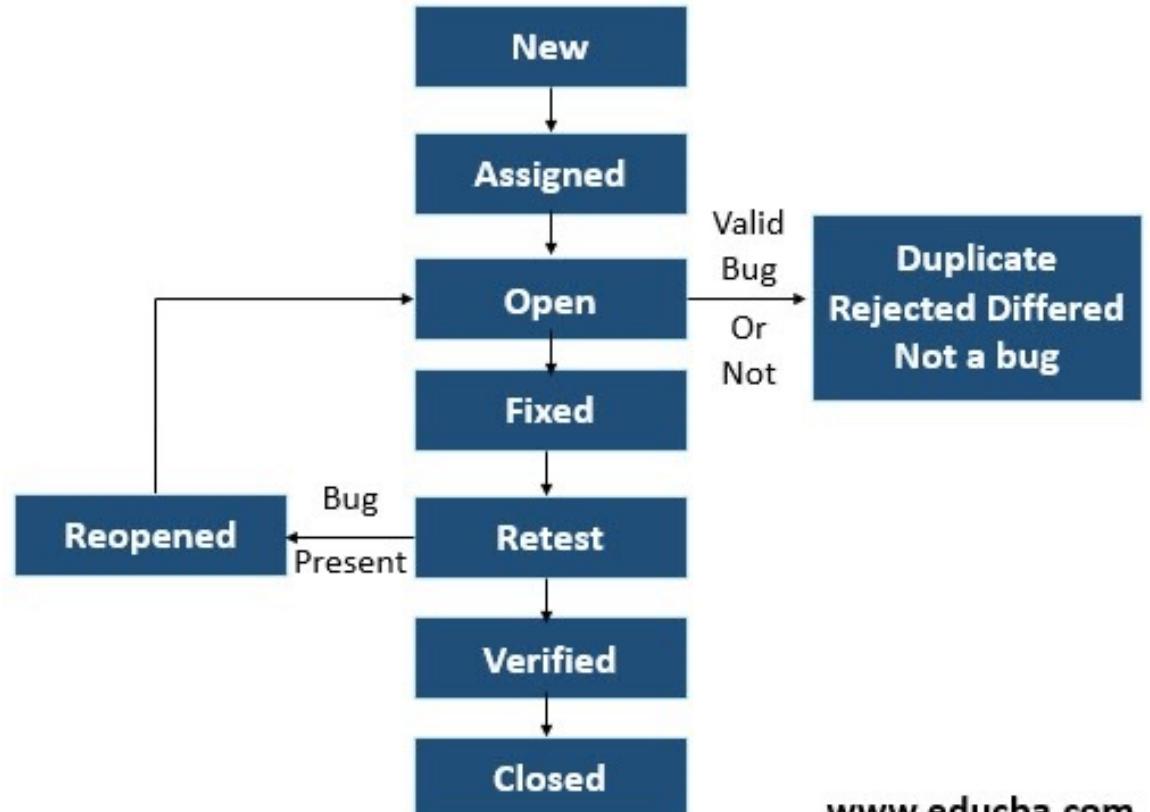


# Lifecycle of a Defect

- The life-cycle of a defect is usually simplistically represented by a state-transition diagram with five states:
  - NEW
    - A problem report with a severity level and priority is filled
  - ASSIGNED
    - The problem is assigned to an appropriate person
  - OPEN
    - The assigned person is actively working on the problem to resolve it
  - RESOLVED
    - The assigned person has resolved the problem and is waiting for the submitter to verify and close it.
  - CLOSED
    - The submitter has verified the resolution of the problem and accepted it.

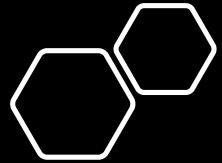


# Lifecycle of a Defect Jira



[www.educba.com](http://www.educba.com)

<https://www.educba.com/jira-bug-life-cycle/>



# Defects Characteristics



## Priority - A Measure of how soon the defect needs to be fixed (urgency)

Critical – must be resolved as soon as possible

High – must be resolved in a high priority

Medium – must be resolved but there is no pressure of time

Low – should be resolved when possible.



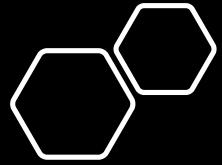
## Severity - A measure of the extent of the detrimental effect of the defect on the operation of the product.

Critical : when one or more critical system functionalities are impaired and there is no workaround;

High: some functionalities of the system are impaired but there are workarounds;

Medium : no critical functionalities of the system is impaired.

Low: the problem is mostly cosmetic.



# Defects Characteristics Jira

## View Priorities

The table below shows the priorities used in this version of JIRA, in order from highest to lowest.  
• [Translate priorities](#)

Name	Description	Icon	Color	Order	Operations
<b>Highest</b>	This problem will block progress.	↑	█	▲	<a href="#">Edit</a> · <a href="#">Delete</a> · Default
<b>High</b>	Serious problem that could block progress.	↑	█	▲	<a href="#">Edit</a> · <a href="#">Delete</a> · Default
<b>Medium</b>	Has the potential to affect progress.	▲	█	▲	<a href="#">Edit</a> · <a href="#">Delete</a> · Default
<b>Low</b>	Minor problem or easily worked around.	↓	█	▲	<a href="#">Edit</a> · <a href="#">Delete</a> · Default
<b>Lowest</b>	Trivial problem with little or no impact on progress.	↓	█	▲	<a href="#">Edit</a> · <a href="#">Delete</a> · Default

## Add New Priority

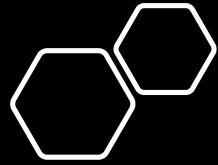
Name\*

Description

Icon URL\*  [ select image ]  
(relative to the JIRA web application e.g. /images/icons OR starting with http://)

Priority Color\*

<https://www.cwiki.us/display/JIRA064/Defining+Priority+Field+Values>



# Defects Characteristics Jira

Create Issue Configure Fields ▾

Project\*

Issue Type\*  Bug ?

Summary\*

Priority  ?

Due Date

Component/s   
Start typing to get a list of possible matches or press down to select.

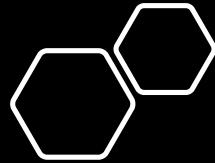
Affects Version/s   
Start typing to get a list of possible matches or press down to select.

Fix Version/s   
Start typing to get a list of possible matches or press down to select.

Assignee

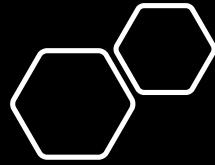
Create another Create Cancel

<https://www.softwaretestinghelp.com/jira-bug-tracking/>



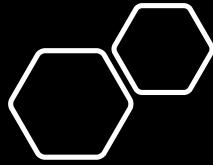
# Metrics for Tracking the Test Execution

- Test execution brings forth 3 different facets of software development:
  - Developers would like to know the degree to which the system meets the explicit as well as implicit requirements.
  - The delivery date cannot be precisely predicted due to the uncertainty in fixing problems.
  - The customer is excited to take the delivery of the product.



# Metrics for Tracking the Test Execution

- It is very important to monitor certain metrics which truly represent the progress of system testing and reveal the quality level of the system!
  - Monitoring the test execution
  - Monitoring the defects
- This is very important for timely decisions about the test!



# Metrics for Monitoring Test Execution

## Test case Escapes (TCE)

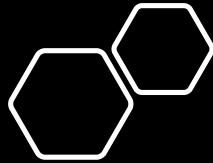
- A significant increase in the number of NEW test cases during execution implies that deficiencies in the test design

## Planned versus Actual Execution (PAE) Rate

- Compare the actual number of test cases executed every week with the planned number of test cases

## Execution Status of Test (EST) Cases

- Periodically monitor the number of test cases lying in different states
  - *Failed, Passed, Blocked, Invalid and Untested*



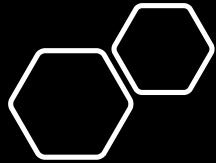
# Monitoring Test Execution Jira

project IN (BM, AB) AND issuetype = Bug and assignee=currentuser() and priority in (blocker, critical) and status in ("To Do", open)

1–9 of 9

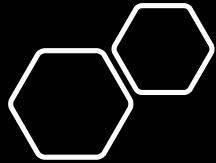
T	Key	Summary	Assignee	Reporter	P	Status	Resolution	Created	Updated
	BM-132	Test	Swati Seela [Administrator]	Swati Seela [Administrator]		OPEN	Unresolved	17/Feb/16	03/Mar/16
	AB-38	Rest_validatAccount_creditcard_ ; credit card validation gives incorrect error message when Expiry date is prior than the present date.	Swati Seela [Administrator]	Test		TO DO	Unresolved	13/Aug/15	19/Aug/15
	AB-37	Update functionality for SSN.accts input'0'with success.but does not update in DB	Swati Seela [Administrator]	Test		TO DO	Unresolved	13/Aug/15	19/Aug/15
	AB-35	Timeout Error for server socket	Swati Seela [Administrator]	Test		TO DO	Unresolved	13/Aug/15	19/Aug/15
	AB-29	Buy credit card accepts negative amount and it's adding in available limit in database	Swati Seela [Administrator]	Test		TO DO	Unresolved	13/Aug/15	19/Aug/15
	AB-27	In CC card restful service "Add account" Credit limit ,available limit and cash limit accepts 0 and show "msg". "Successfully saved credit card information".	Swati Seela [Administrator]	Test		TO DO	Unresolved	13/Aug/15	19/Aug/15
	AB-26	In CC restful service "add account" CC expiry date should be greater than the current date but accepts the date prior to the current date.	Swati Seela [Administrator]	Test		TO DO	Unresolved	13/Aug/15	19/Aug/15
	AB-19	If we give non matching name and credit card number i.e name of one person and registered credit card of other person the amount in the request get deposited in the account of name provided.	Swati Seela [Administrator]	Test		TO DO	Unresolved	13/Aug/15	19/Aug/15

<https://www.softwaretestinghelp.com/jira-bug-tracking/>



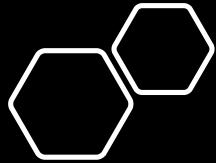
# Metrics for Monitoring Defect Reports

- Function as Designed (FAD) Count
  - False positives
  - If the number of false positives are high, maybe the testers have inadequate understanding of the system.
- Irreproducible Defects (IRD) Count
  - If a defect cannot be reproduced, then the developers may not be able to gain useful insight into the cause of the failure.
  - If many defects are irreproducible, may mean that the system is unreliable for some reason.
- Defects Arrival Rate (DAR) Count
  - Measures who is finding the defects. For example, if the marketing and users are the ones that are mostly reporting bugs, it may mean that the test is ineffective.



# Metrics for Monitoring Defect Reports

- Outstanding Defects (OD) Count
  - Defects that are always coming back. Or never resolved.
  - If defects are not resolved and only increasing, it may mean that the system is not increasing the quality level.
- Crash Defects (CD) Count
  - Defects causing a system to crash must be recognized as an important category of defects because a system crash is a serious event leading to complete unavailability of the system and possibly loss of data.
- Lead time to Resolution of Defects Count
  - If the defects that appear in the system are usually easy to fix and fast to fix it means that the system has better quality.

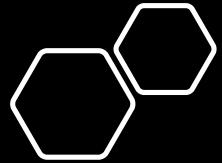


# Orthogonal Defect Classification (ODC)

- A methodology for rapid capturing of the semantics of each software defect.
- In the NEW state, the submitter needs to fill out the following ODC attributes or fields:
  - *Activity*: This is the activity that was being performed at the time the defect was discovered
  - *Trigger*: The environment or condition that had to exist for the defect to surface
  - *Impact*: This refers to the effect, the defect would have on the customer, if the defect had escaped to the field.
- The owner needs to fill out the other ODC attributes or fields when the defect is moved to RESOLVED state.
- The ODC analysis can be combined with Pareto analysis to focus on error-prone parts of the software
  - 80% of the problems can be fixed with 20% of the effort

# Orthogonal Defect Classification

- Target: The target represents the high-level identity, such as design, code, or documentation, of the entity that was fixed
- Defect type: The defect type represents the actual correction that was made
- Qualifier: The qualifier specifies whether the fix was made due to missing, incorrect, or extraneous code
- Source: The source indicates whether the defect was found in code developed in house, reused from a library, ported from one platform to another, or provided by a vendor
- Age: The history of the design or code that had the problem. The age specifies whether the defect was found in new, old (base), rewritten, or refixed code.
  - New: The defect is in a new function which was created by and for the current project.
  - Base: The defect is in a part of the product which has not been modified by the current project. The defect was not injected by the current project
  - Rewritten: The defect was introduced as a direct result of redesigning and/or rewriting of old function in an attempt to improve its design or quality.
  - Refixed: The defect was introduced by the solution provided to fix a previous defect.



# Monitoring Test Execution Jira

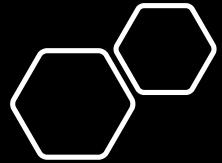
Configure Fields ▾

Too

Show Fields: All | Custom

<input checked="" type="checkbox"/> Affects Version/s	<input checked="" type="checkbox"/> Epic Link
<input checked="" type="checkbox"/> Assignee	<input checked="" type="checkbox"/> Fix Version/s
<input checked="" type="checkbox"/> Attachment	<input checked="" type="checkbox"/> Labels
<input checked="" type="checkbox"/> Component/s	<input checked="" type="checkbox"/> Priority
<input checked="" type="checkbox"/> Description	<input checked="" type="checkbox"/> Summary
<input checked="" type="checkbox"/> Due Date	<input checked="" type="checkbox"/> Time Tracking
<input checked="" type="checkbox"/> Environment	

<https://www.softwaretestinghelp.com/jira-bug-tracking/>

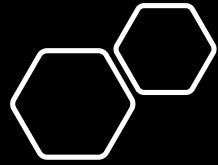


# Defect Causal Analysis (DCA)

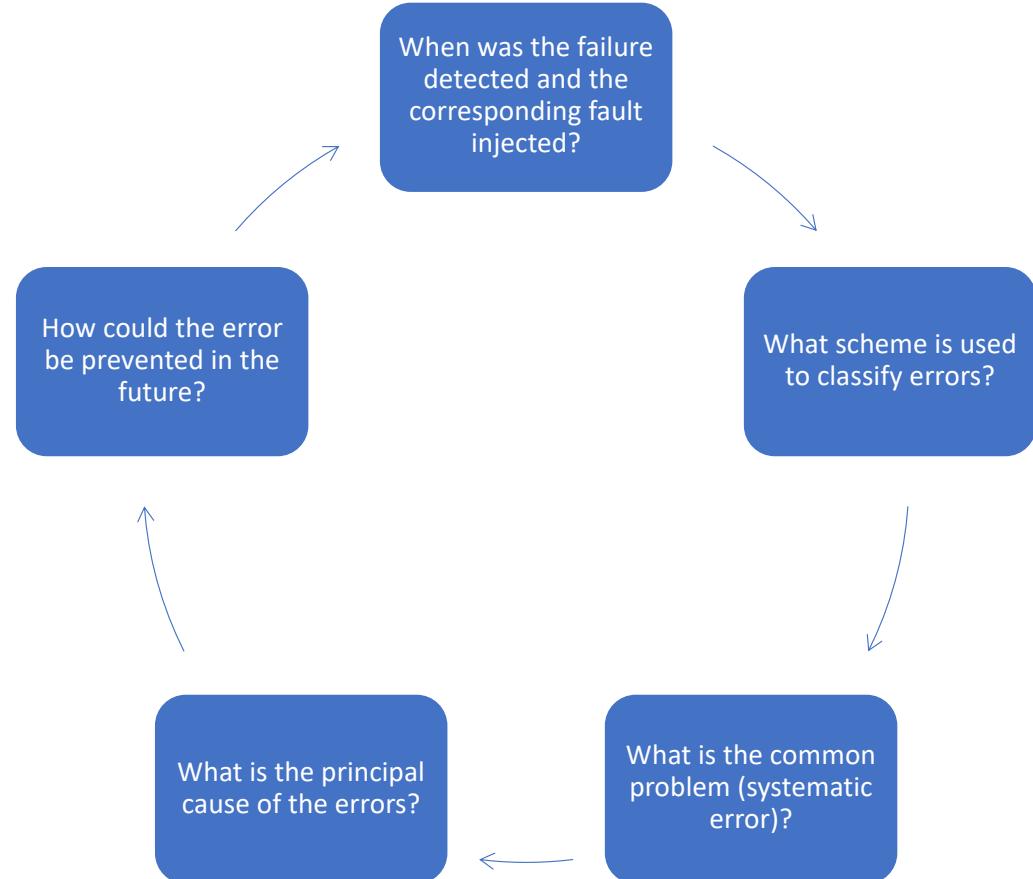
Used to raise the quality level of the products at a lower cost.

Focuses on understanding of cause-effect relationship of an error to

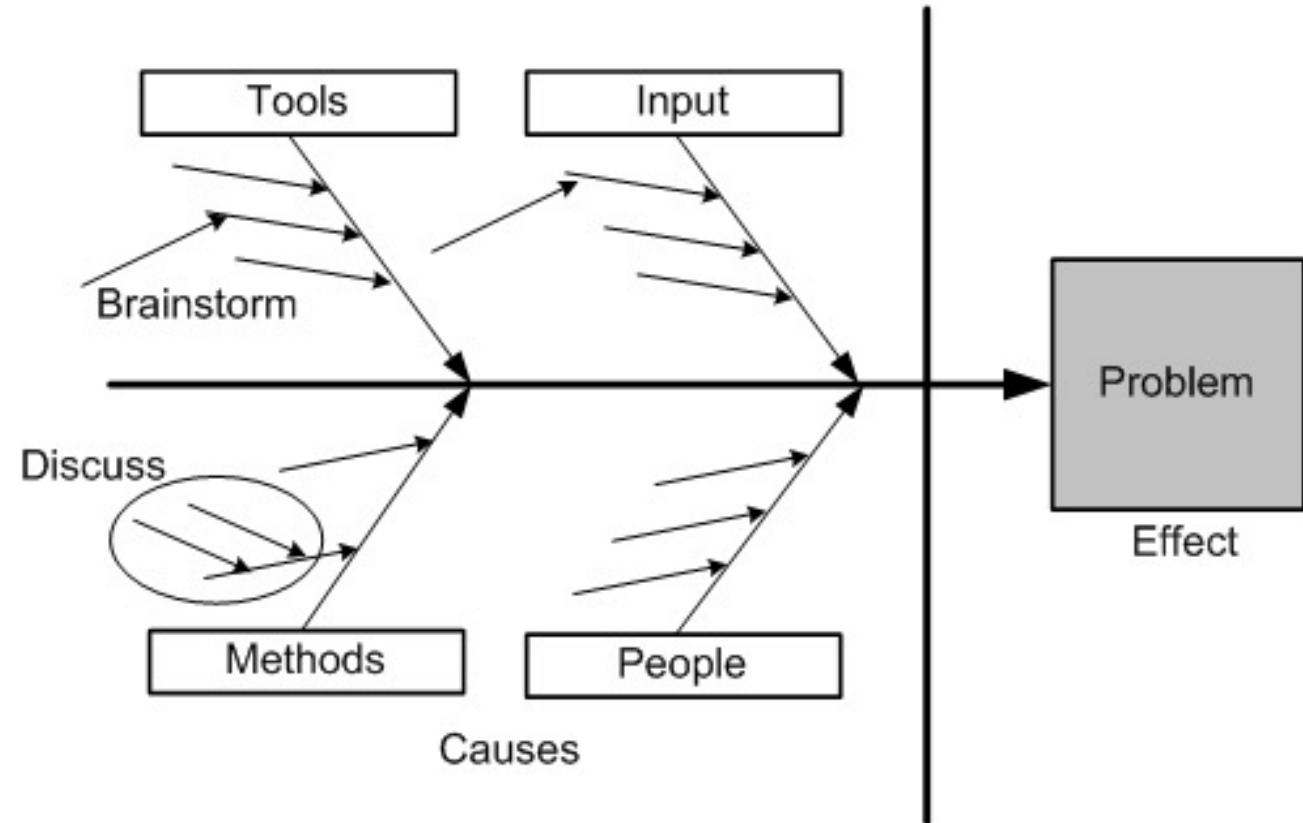
- Take actions to prevent similar errors from occurring in the future;
- Reduce the number of defects;
- Focus on systematic errors;

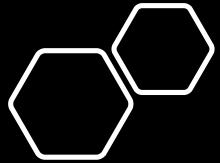


# Defect Causal Analysis (DCA) Five Steps



# Cause-effect diagram for defect causal analysis (DCA)

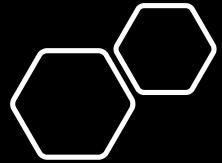




# Defect Causal Analysis (DCA)

- Examples of Recommended Actions can come from the DCA:
  - Training
    - People improving their knowledge about the product
  - Improvement in Communication
    - Communication between testers and developers can be of a big help in cases of too many false positives or duplicated defects reporting for example.
  - Using Tools
    - Recommend the use of tools to monitor memory for example
  - Process improvements
    - Actions to improve existing processes or defining new processes.

<https://www.knowledgehut.com/blog/agile/root-cause-analysis-agile-teams>



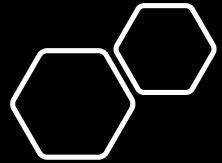
# Defect Causal Analysis (DCA) in Agile

- Best Place for this are on retrospectives!

<https://www.benlinders.com/2013/getting-to-the-root-causes-of-problems-in-a-retrospective/>

# System Test Report

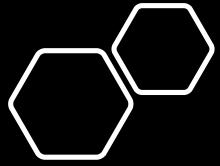
A final summary report is created after the completion of all the tests



# System Test Report

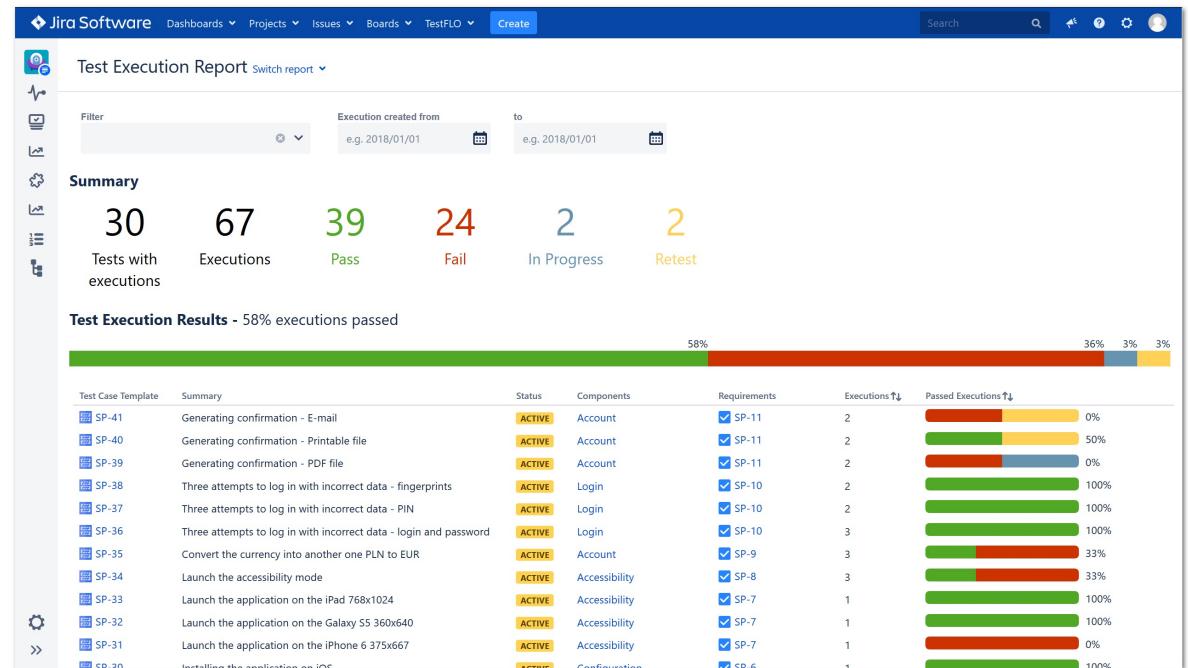
1. Introduction to the Test Project
2. Summary of Test Results
3. Performance Characteristics
4. Scaling Limitations
5. Stability Observations
6. Interoperability of the System
7. Hardware/Software Compatible Matrix
8. Compliance Requirement Status

Table 13.14: Structure of the final system test report.

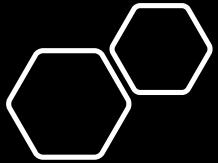


# System Test Report

Testflo example

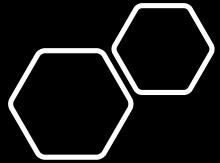


<https://deviniti.com/support/addon/server/testflo-83/latest/test-execution-report/#&gid=null&pid=2>



# Readiness Criteria Example

- All the test cases from the test suite should have been executed
- Test case results are updated with Passed, Failed, Blocked, or Invalid status
- The requirements are updated by moving each requirement from the Verification state to either the Closed or the Decline state, as discussed in Chapter 11
- The pass rate of test cases is very high, say, 98%
- No crash in the past two weeks of testing has been observed
- No known defect with critical or high severity exists in the product
- Not more than a certain number of known defects with medium and low levels of severity exist in the product
- All the resolved defects must be in the CLOSED state
- The user guides are in place
- Trouble shooting guide is available
- The test report is completed and approved

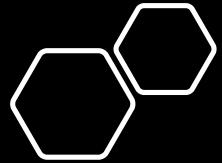


# Measuring Test Effectiveness

- After a product is deployed measure the number of defects found by the customers that were not found by the testers and development team.
  - ESCAPED DEFECTS

Defect Removal Efficiency (DRE) metric defined as follows:

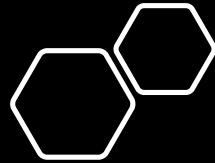
$$DRE = \frac{\text{Number of Defects Found in Testing}}{\text{Number of Defects Found in Testing} + \text{Number of Defects Not Found}}$$



# Measuring Test Effectiveness with Spoilage Metric

A new metric called spoilage is defined as

$$\text{Spoilage} = \frac{\sum (\text{Number of Defects} \times \text{Discovered Phage})}{\text{Total Number of Defects}}$$

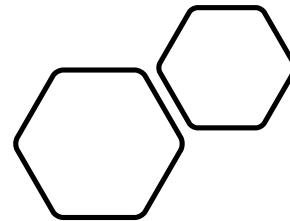


# Measuring Test Effectiveness with Spoilage Metric

Phase Injected	Phase Discovered								Total Defects	Spoilage = Weight/Total Defects
	Requirements	High-Level Design	Detailed Design	Coding	Unit Testing	Integration Testing	System Testing	Acceptance Testing		
Requirements	0	7	6	3	0	0	12	28	56	17 3.294117647
High-Level		0	8	8	3	8	30	6	63	22 2.863636364
Detailed Design			0	13	6	12	20	0	51	25 2.04
Coding				0	63	48	111	48	270	136 1.985294118
Summary	0	7	14	24	72	68	173	82	440	200 2.2

A spoilage value close to 1 is an indication of a more effective defect discovery process

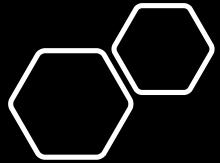
# How is this performed in Agile?



Discuss in groups how to do a test planning in testing management tool you know.

You can also use the example of TestFlo

<https://deviniti.com/support/addon/server/testflo-84/latest/test-planning/>



## Further Material

### How is this performed in Agile?

**Certified Tester**

**Foundation Level Extension Syllabus**  
**Agile Tester**

Version 2014

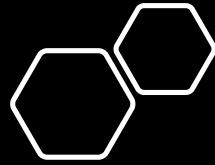
---

International Software Testing Qualifications Board

---

Copyright Notice  
This document may be copied in its entirety, or extracts made, if the source is acknowledged.

<https://www.istqb.org/downloads/send/5-foundation-level-agile-tester/41-agile-tester-extension-syllabus.html>



Further work

A complete quiz  
about Agile Testing  
from ISTB with 40  
Questions

[https://softwaretester.net/istqb\\_sample\\_exam\\_agile\\_tester\\_f1/](https://softwaretester.net/istqb_sample_exam_agile_tester_f1/)

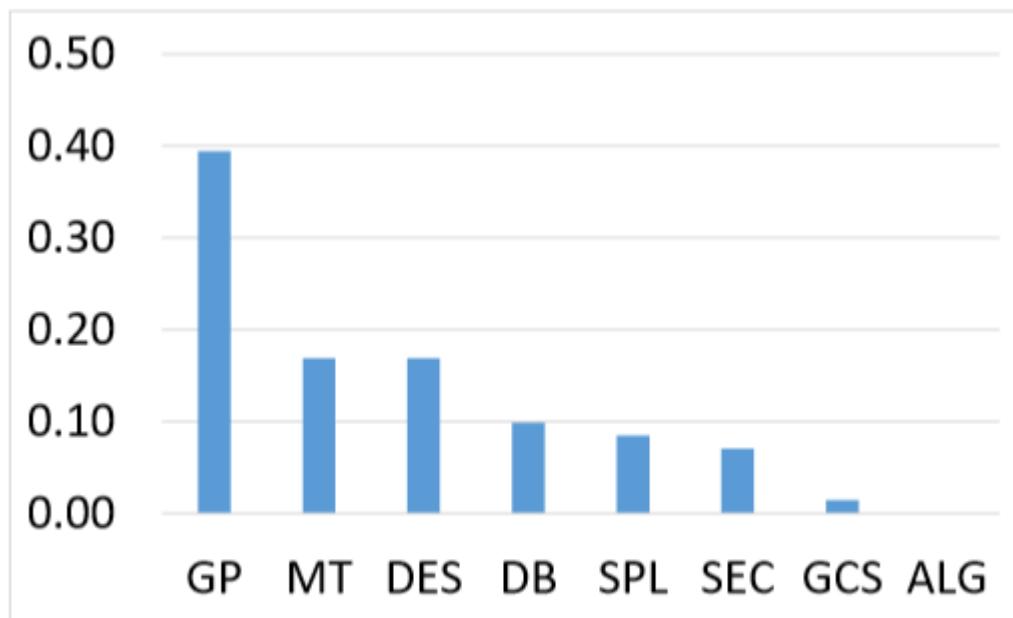
# **Code refactoring and review**

# Outline

- Code refactoring principles
  - Why refactoring
  - Bad smells in code
  - General refactoring principles and techniques
- Refactoring of Python code
- Code inspection and review
  - Testing vs. inspection
  - Reading techniques

# **Code refactoring principles**

# Readable and maintainable code is desired by industry



For software development (S), general programming knowledge (GP) and maintenance (MT) are the top two items.

**GP:** Clean and readable code with good documentation

**MT:** experience, especially with maintaining code

What competence do software companies want

Tor Stålhane, IDI, NTNU  
Guttorm Sindre, IDI, NTNU  
Extended version of conference paper "Hva vil programvareindustrien ha?", MNT-konferansen 2019  
Stålhane, Deraas, Sindre and Abrahamsson:

# Bad smells in code (code smells)

- Code smells are any violation of fundamental design principles that decrease the overall quality of the code.
- Not bugs or errors
- Can certainly add to the chance of bugs and failures down the line.

# Code smells categories

- Duplicated Code
- Long Method
- Large Class
- Long Parameter List
- Divergent Change
- Shotgun Surgery
- Feature Envy
- Data Clumps
- Primitive Obsession
- Switch Statements
- Lazy Element
- Speculative Generality
- Temporary Field
- Message Chains
- Middle Man
- Data Class
- Refused Bequest

# Cod refactoring goals and properties

- Change the internal structure without changing external behavior
- Eliminate code smells for
  - Readability
  - Consistency
  - Maintainability
- Properties
  - Preserve correctness
  - One step at a time
  - Frequent testing

# Code refactoring steps

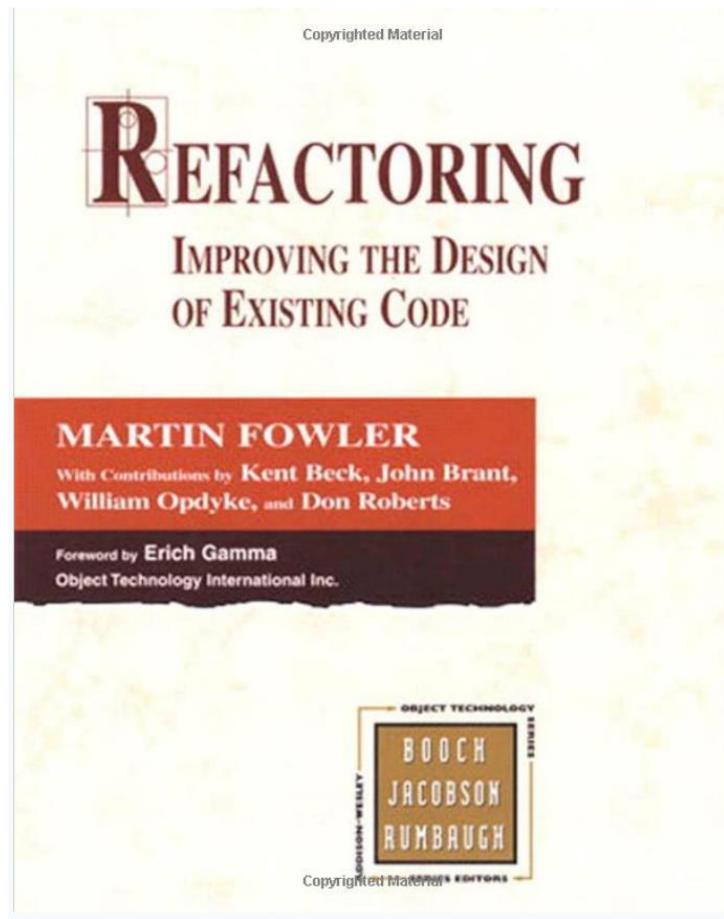
- Designing solid tests for the section to be refactored
- Reviewing the code to identify bad smells of code
- Introducing refactoring and running tests (One step at a time)

# Refactoring risks and countermeasures

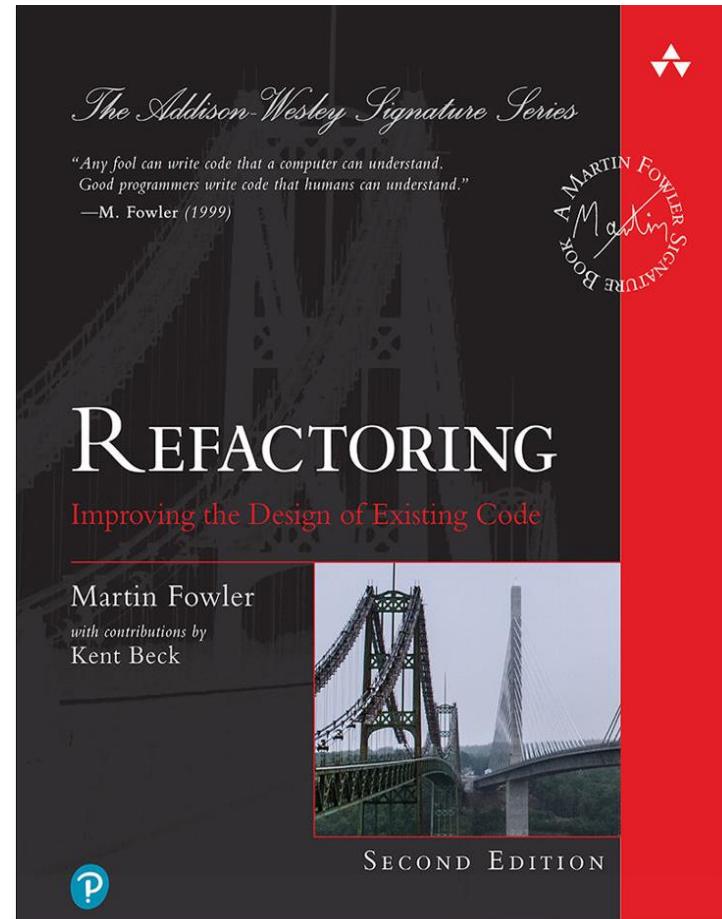
- Refactoring is regarded as an overhead activity
- Introducing failures with refactoring
- Outdated comments and documents

- Balance cost and benefits
- Better to start from day one! Not at the end of the project/delivery
- Should have sufficient and efficient regression tests

# Relevant books



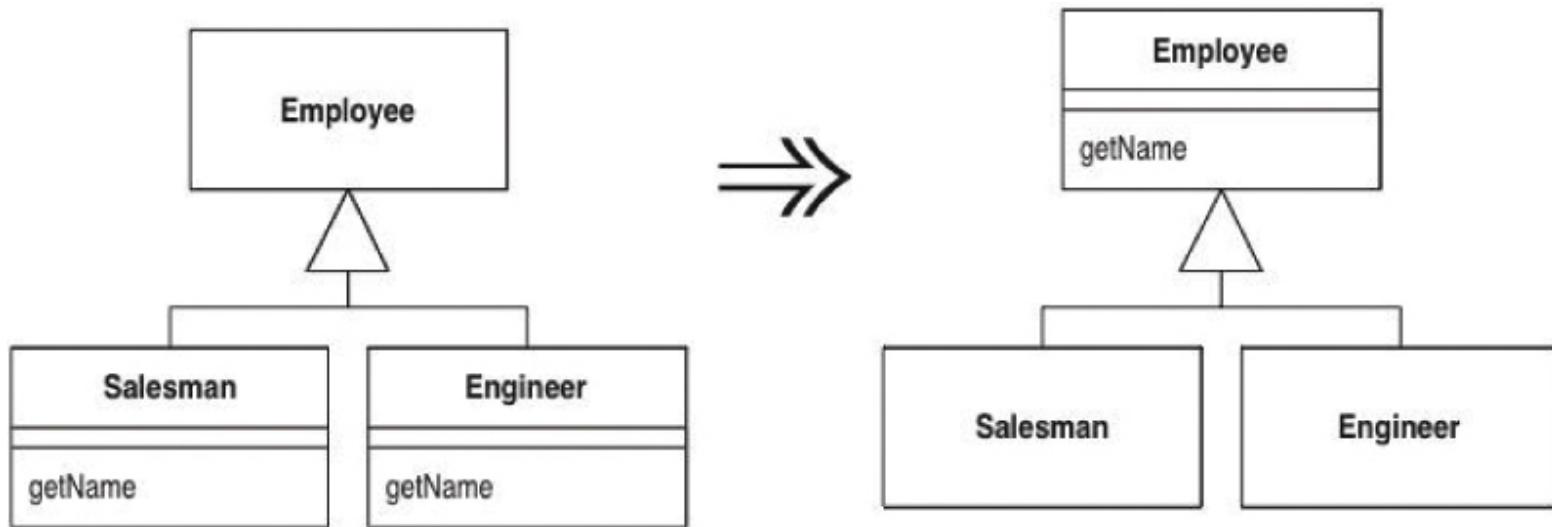
Old edition, available online at NTNU lib.



New edition

# Remove Duplicated Code – Pull up method (Don't repeat yourself (DRY))

- You have methods with identical results on subclasses.



# Remove Duplicated code - Substitute Algorithm

- You want to replace an algorithm with one that is clearer.

```
String foundPerson(String[] people){  
  
    for (int i = 0; i < people.length; i++) {  
        if (people[i].equals ("Don"))  
        { return "Don"; }  
  
        if (people[i].equals ("John"))  
        { return "John"; }  
  
        if (people[i].equals ("Kent"))  
        { return "Kent"; }  
    }  
    return "";  
}
```



```
String foundPerson(String[] people){  
    List candidates = Arrays.asList(new  
    String[] {"Don", "John", "Kent"});  
  
    for (int i=0; i<people.length; i++)  
        if (candidates.contains(people[i]))  
            return people[i];  
  
    return "";  
}
```

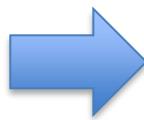
# Reduce size – shorten method/class

- **Long Method**
  - E.g., Extract method
- **Large Class**
  - E.g., Extract class, subclass, interface

# Extract method example

If you have to spend effort looking at a fragment of code and figure out what it is doing, then you should extract it into a function/method and name it after “what.”

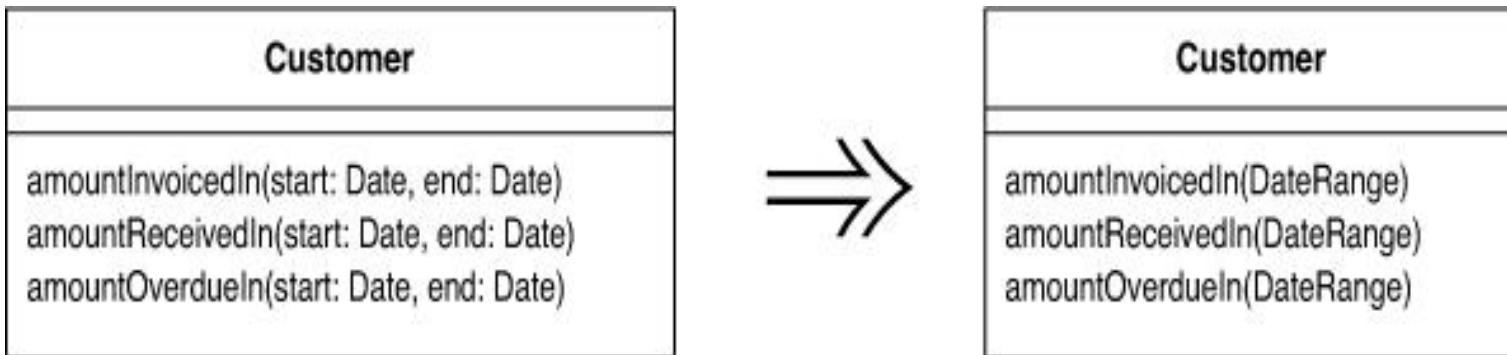
```
void printOwing()  
{  
    printBanner(); //print details  
    System.out.println ("name:  
        " + _name);  
    System.out.println ("amount  
        " + getOutstanding());  
}
```



```
void printOwing() {  
    printBanner();  
    printDetails(getOutstanding());  
}  
  
void printDetails (double outstanding) {  
    System.out.println ("name:" + _name);  
    System.out.println ("amount:" +  
        outstanding);  
}
```

# Reduce size – shorten parameter list

- **Long Parameter List**
  - E.g., Introduce Parameter Object



# Divergent Change

- **Code smell**
  - One module is often changed in different ways for **different reasons**.
  - Classes have more than distinct responsibilities that it **has more than one reason to change**
  - Violation of **single responsibility design** principle
- **Refactoring**
  - You identify everything that changes for a particular cause and put them all together

```
public class Account {  
    private int accountNumber;  
    private double balance = 0;  
  
    public Account(int accountNumber){  
        this.accountNumber = accountNumber;  
    }  
  
    public int getAccountNumber() {  
        return accountNumber;  
    }  
  
    public double getBalance() {  
        return balance;  
    }  
  
    public void credit(double amount) {  
        balance += amount;  
    }  
  
    public void debit(double amount) {  
        balance -= amount;  
    }  
  
    public String toXml() {  
        return "<account><id>" + Integer.toString(getAccountNumber()) + "</id>" +  
            "<balance>" + Double.toString(getBalance()) + "</balance></account>";  
    }  
}
```

More than one responsibility, operation  
of a bank account + serializing the  
account

<https://www.slideshare.net/nadeembtech/code-craftsmanship>

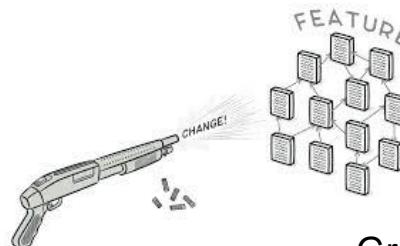
```
public class Account {  
    private int accountNumber;  
    private double balance= 0;  
  
    public Account(int accountNumber){  
        this.accountNumber = accountNumber;  
    }  
  
    public int getAccountNumber() {  
        return accountNumber;  
    }  
  
    public double getBalance() {  
        return balance;  
    }  
  
    public void credit(double amount) {  
        balance += amount;  
    }  
  
    public void debit(double amount) {  
        balance -= amount;  
    }  
}
```

```
public class AccountXMLSerializer {  
  
    public String toXml(Account account) {  
        return "<account>" +  
            "<id>" + Integer.toString(account.getAccountNumber()) + "</id>" +  
            "<balance>" + Double.toString(account.getBalance()) + "</balance>" +  
            "</account>";  
    }  
}
```

<https://www.slideshare.net/nadeembtech/code-craftsmanship>

# Shotgun Surgery

- **Code smell**
  - A single change is made to multiple classes simultaneously
  - When changes are all over the place, they are hard to find, and it's easy to miss important changes.



Graph from refactoring.guru

- **Refactoring**
  - You put all the changes into a single class
  - Once and only once (OAOO)

```

1 package com.example.codesmell;
2
3 public class Account {
4
5     private String type;
6     private String accountNumber;
7     private int amount;
8
9     public Account(String type, String accountNumber, int amount)
10    {
11        this.amount=amount;
12        this.type=type;
13        this.accountNumber=accountNumber;
14    }
15
16
17    public void debit(int debit) throws Exception
18    {
19        if(amount <= 500)
20        {
21            throw new Exception("Mininum balance shuold be over 500");
22        }
23
24        amount = amount-debit;
25        System.out.println("Now amount is" + amount);
26
27    }
28
29    public void transfer(Account from, Account to, int cerditAmount) throws Exception
30    {
31        if(from.amount <= 500)
32        {
33            throw new Exception("Mininum balance shuold be over 500");
34        }
35
36        to.amount = amount+cerditAmount;
37
38    }
39
40    public void sendWarningMessage()
41    {
42        if(amount <= 500)
43        {
44            System.out.println("amount should be over 500");
45        }
46
47    }

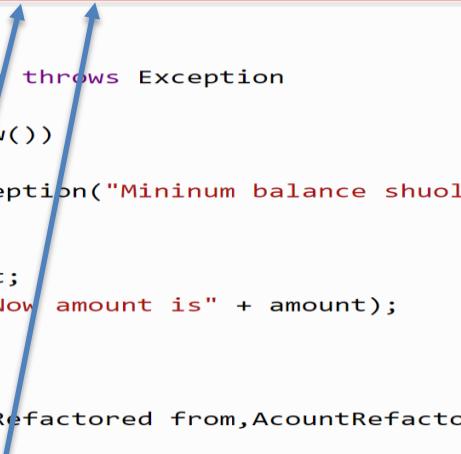
```

<https://dzone.com/articles/code-smell-shot-surgery>

When we add another criterion in the validation logic: if the account type is personal and the balance is over 500

We have to make changes to all methods

```
1 package com.example.codesmell;
2
3 public class AccountRefactored {
4
5     private String type;
6     private String accountNumber;
7     private int amount;
8
9
10    public AccountRefactored(String type, String accountNumber, int amount)
11    {
12        this.amount=amount;
13        this.type=type;
14        this.accountNumber=accountNumber;
15    }
16
17
18    private boolean isAccountUnderflow()
19    {
20        return amount<=500;
21    }
22
23
24
25    public void debit(int debit) throws Exception
26    {
27        if(isAccountUnderflow())
28        {
29            throw new Exception("Mininum balance shuold be over 500");
30        }
31
32        amount = amount-debit;
33        System.out.println("Now amount is" + amount);
34    }
35
36
37    public void transfer(AccountRefactored from, AccountRefactored to, int creditAmount) throws Exception
38    {
39        if(isAccountUnderflow())
40        {
41            throw new Exception("Mininum balance shuold be over 500");
42        }
43
44        to.amount = amount+creditAmount;
45
46    }
}
```



# Menti: code refactoring performed?

Go to [menti.com](http://menti.com), code 6344 0068

```
1public class Account {  
2    double principal,rate;  int daysActive,accountType;  
3  
4    public static final int STANDARD=0, BUDGET=1,  
5        PREMIUM=2, PREMIUM_PLUS=3;  
6}  
7  
8...  
9  
10public static double calculateFee(Account[] accounts)  
11{  
12    double totalFee= 0.0;  
13    Account account;  
14    for (int i=0;i<accounts.length;i++) {  
15        account=accounts[i];  
16        if (account.accountType== Account.PREMIUM||  
17            account.accountType== Account.PREMIUM_PLUS)  
18            totalFee+= .0125 * (          // 1.25% broker's fee  
19                account.principal* Math.pow(account.rate,  
20                    (account.daysActive/365.25))  
21                    -account.principal); // interest-principal  
22    }  
23  
24    return totalFee;  
25}  
26}  
27
```

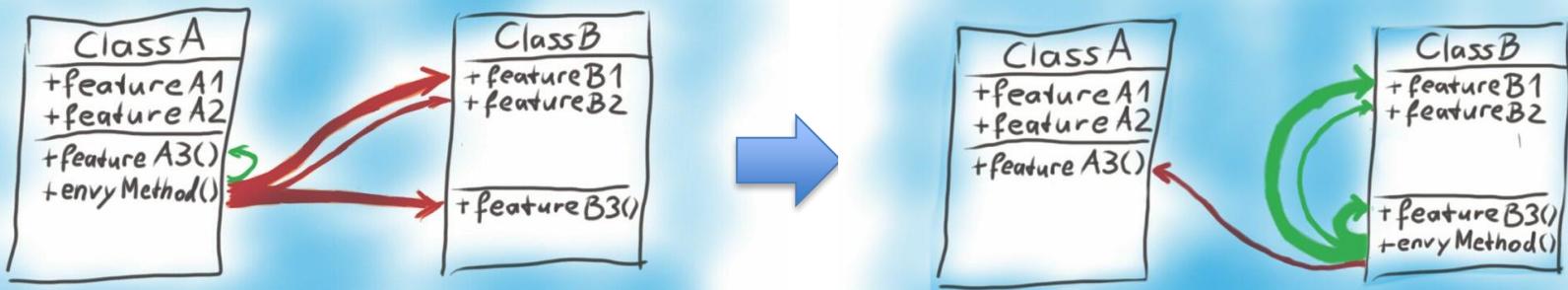
[http://www.ifi.uzh.ch/seal/teaching/courses/softwareWartung-FS16/CodeReviewExercise.pdf\\*](http://www.ifi.uzh.ch/seal/teaching/courses/softwareWartung-FS16/CodeReviewExercise.pdf)



```
1/** An individual account. Also see CorporateAccount. */  
2  
3public class Account {  
4  
5    private double principal;  
6    /** The yearly, compounded rate (at 365.25 days per year). */  
7  
8    private double rate; /** Days since last interest pay out */  
9    private int daysActive;  
10   private Type type;  
11}  
12  
13/** The varieties of account our bank offers. */  
14public enum Type {STANDARD, BUDGET, PREMIUM, PREMIUM_PLUS}  
15  
16  
17/** Compute interest. **/  
18public double interest() {  
19    double years = daysActive / 365.25;  
20    double compoundInterest = principal * Math.pow(rate, years);  
21    return compoundInterest - principal;  
22}  
23  
24/** Return true if this is a premium account. **/  
25public boolean isPremium() {  
26    return accountType == Type.PREMIUM ||  
27        accountType == Type.PREMIUM_PLUS;  
28}  
29  
30/** The portion of the interest that goes to the broker. **/  
31public static final double BROKER_FEE_PERCENT = 0.0125;  
32  
33/** Return the sum of the broker fees for all the given accounts. **/  
34public static double calculateFee(Account accounts[]) {  
35    double totalFee = 0.0;  
36    for (Account account : accounts) {  
37        if (account.isPremium()) {  
38            totalFee += BROKER_FEE_PERCENT * account.interest();  
39        }  
40    }  
41    return totalFee;  
42}  
43}
```

# Increase cohesion

- **Feature envy**
  - A function in one module spends more time communicating with functions or data inside another module than it does within its own module.
  - Move function to give it a dream home



<https://waog.wordpress.com/2014/08/25/code-smell-feature-envy/>

# Increase cohesion (Cont')

- **Data clumps**
  - Bunches of data often hang around together
  - Consolidate the data together, e.g., Introduce Parameter Object or Preserve Whole Object

```
int low = daysTempRange().getLow();
int high = daysTempRange().getHigh();
withinPlan = plan.withinRange(low, high);
```



```
withinPlan =
plan.withinRange(daysTempRange());
```

# Primitive Obsession

- Primitive fields are basic built-in building blocks of a language, e.g., int, string, or constants
- Primitive Obsession is when the code relies too much on primitives and when uses primitive types to represent an object in a domain

```
class contactUs
{
    public function addressUsa()
    {
        $address = new Array();
        $address['streetNo'] = 2074;
        $address['streetName'] = 'JFK street';
        $address['zipCode'] = '507874';

        return $address['streetName']. ' '. $address['streetNo']. ', '. $address['zipCode'];
    }

    public function addressGermany()
    {
        $address = new Array();
        $address['streetNo'] = '25';
        $address['streetName'] = 'Frankfurter str.';
        $address['zipCode'] = '80256';

        return $address['streetName']. ' '. $address['streetNo']. ', '. $address['zipCode'];
    }

    public function hotLine(){
        return '+49 01687 000 000';
    }
}
```

The address is defined as an array.  
Every time we need the address we will  
have to hard code it

<https://codeburst.io/write-clean-code-and-get-rid-of-code-smells-aea271f30318>

```
class address{  
  
    private $streetNo;  
    private $streetName;  
    private $zipCode;  
  
    public function addressUsa()  
    {  
        $this->streetNo = 2074;  
        $this->streetName = 'JFK street';  
        $this->zipCode = '507874';  
  
        return $this->streetName . ' ' . $this->streetNo . ', ' . $this->zipCode;  
    }  
    public function addressGermany()  
    {  
        $this->streetNo = 25;  
        $this->streetName = 'Frankfurter str.';  
        $this->zipCode = '80256';  
  
        return $this->streetName . ' ' . $this->streetNo . ', ' . $this->zipCode;  
    }  
}
```

We create a new class called Address  
Every time we need to add/edit an address we hit the Address class

<https://codeburst.io/write-clean-code-and-get-rid-of-code-smells-aea271f30318>

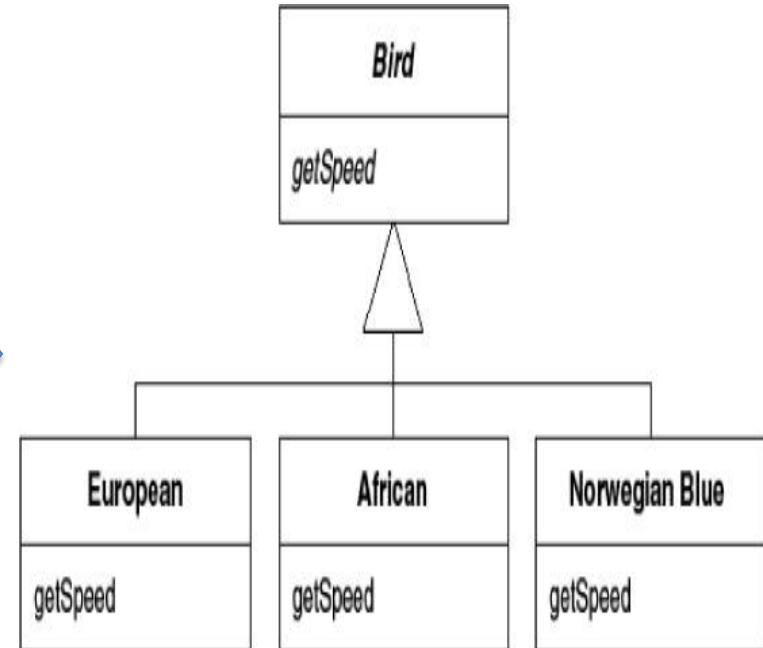
# Code smells categories

- **Duplicated Code** ← Decrease size
- **Long Method**
- **Large Class**
- **Long Parameter List**
- **Divergent Change** ← Localize change
- **Shotgun Surgery**
- **Feature Envy** ← Increase cohesion
- **Data Clumps**
- **Primitive Obsession**
- **Switch Statements**
- **Lazy Element**
- **Speculative Generality**
- **Temporary Field**
- **Message Chains**
- **Middle Man** ← Simplify logic within and between objects
- **Data Class**
- **Refused Bequest**
- Proper OO practice

# Simplify logic within object – Switch statement

- Replace Conditional with Polymorphism

```
double getSpeed() {  
    switch (_type) {  
        case EUROPEAN:  
            return getBaseSpeed();  
        case AFRICAN:  
            return getBaseSpeed() - getLoadFactor();  
        case NORWEGIAN_BLUE:  
            return getBaseSpeed(_voltage);  
    }  
    throw new RuntimeException ("error");  
}
```



# Simplify logic within object – Switch statement (cont')

- Replace Parameter with Explicit Methods

```
void setValue (String name, int value) {  
  
    if (name.equals("height")) {  
        _height = value;  
        return;  
    }  
  
    if (name.equals("width")) {  
        _width = value;  
        return;  
    }  
  
    Assert.shouldNeverReachHere();  
}
```



```
void setHeight(int arg) {  
  
    _height = arg;  
}  
  
void setWidth (int arg) {  
  
    _width = arg;  
}
```

# Simplify logic within object – Lazy Element

- E.g., Lazy class: A class that isn't doing enough to pay for itself should be eliminated

```
class Person {  
    get officeAreaCode ()  
    { return this._telephoneNumber.areaCode; }  
    get officeNumber ()  
    {return this._telephoneNumber.number;}  
}  
  
Class TelephoneNumber {  
    get areaCode () { return this._areaCode; }  
    get number () { return this._number; }  
}
```



```
Class Person {  
    get officeAreaCode ()  
    {return this._officeAreaCode;}  
  
    get officeNumber () {return  
    this._officeNumber;}  
}
```

# Simplify logic within object - Speculative Generality

- When you have code that **isn't actually needed today**.
- Such code often exist to support future behavior, which may or **may not be necessary in the future**
- Remove unnecessary delegation, unused parameters, dead code

```
Public class Customer {  
    private String name;  
    private String address;  
  
    Public Customer (String name, String add)  
    { this.name = name;  
        this.address = add;  
    }  
  
    String GetName () {  
        return name;  
    }  
}
```

# Simplify logic within object – Temporary Field

- Class has a variable which is only used in some situation
- Trying to understand why a variable is there when it doesn't seem to be used

```
class User
{
    private $id;
    private $name;
    private $privileges;
    private $content;
    private $contactDetails;

    public function __construct()
    {
        $this->contactDetails = new UserContactDetails();
    }

    function notify($message)
    {
        $messageBody = 'Dear ' . $this->name;
        $messageBody .= ' (' . $this->contactDetails->getStreetNumber() . ' ' .
                      $this->contactDetails->getStreetName() . ')';
        $messageBody .= $message;
        $notificationService->sendSMS($this->contactDetails->getPhoneNumber(), $messageBody);
    }

    function delete()
    {
        systemDelete($this->id);
    }

    function update() {
        systemUpdate($this->id, $this->privileges, $this->content);
    }
}
```

\$name and \$contactDetails are only used in the notify() method.

<https://codeburst.io/write-clean-code-and-get-rid-of-code-smells-aea271f30318>

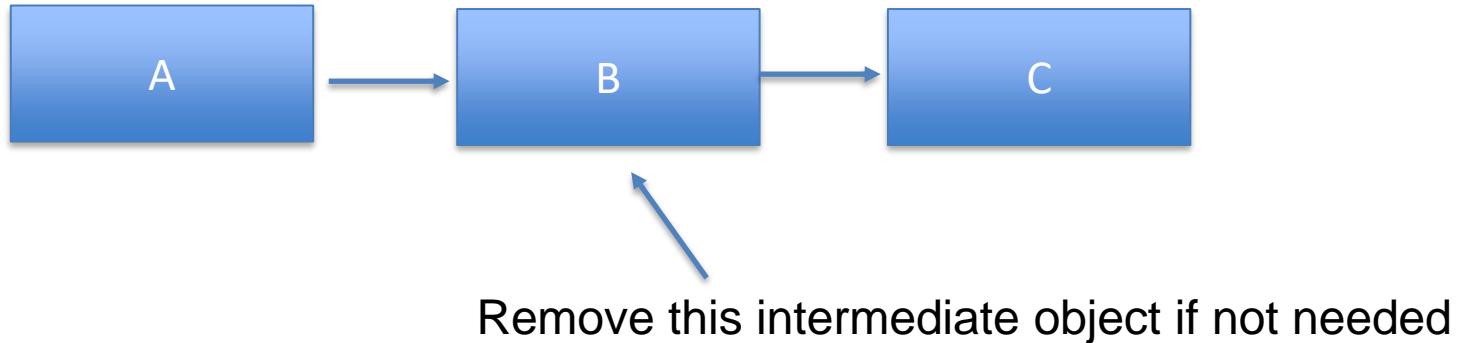
Why not pass them as a method parameters

```
function notify($userName, $contactDetails, $message) {  
  
    $messageBody = 'Dear ' . $userName;  
    $messageBody .= ' (' . $contactDetails->getStreetNumber() . ' ' .  
    |   $contactDetails->getStreetName() . ') .';  
    $messageBody .= $message;  
  
    $notificationService->sendSMS($contactDetails->getPhoneNubmer(), $messageBody);  
}
```

<https://codeburst.io/write-clean-code-and-get-rid-of-code-smells-aea271f30318>

# Simplify logic between objects

- **Message Chains**
- **Middle Man**



Employee->EmployeeConfig->Config

Employee->Config

```
Class Employee{
    public function getConfiguration() {
        $this->employeeConfig->getConfiguration();
    }
}
```

```
Class EmployeeConfig{
    public function getConfiguration() {
        $this->config->getConfiguration();
    }
}
```

```
Class Config{
    public function getConfiguration() {
        $this->loadConfiguration();
    }
}
```



```
Class Employee{
    public function getConfiguration() {
        $this->config->getConfiguration();
    }
}
```

# Proper OO practice

- **Data Class**
  - Make a public field private and provide accessors

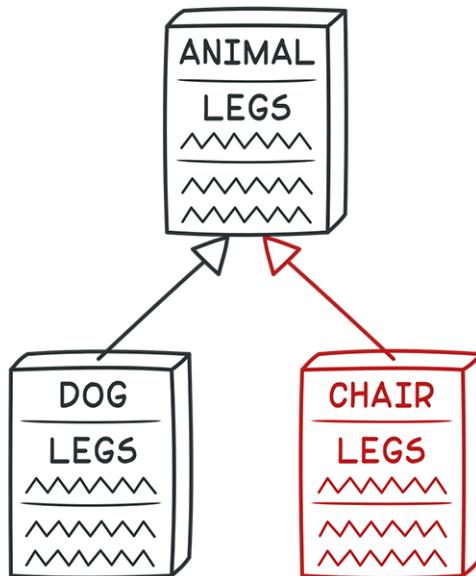
```
public String _name
```



```
private String _name;  
public String getName() {return _name;}  
public void setName(String arg) {_name = arg;}
```

# Proper OO practice (Cont')

- **Refused Bequest**
  - Only some of the inheritances are need, i.e., hierarchy is wrong
  - Inheritance makes no sense and the subclass really does have nothing in common with the superclass



<https://refactoring.guru/smells/refused-bequest>

# Comments in code

- Good to have, can be simplified
- **Best if code is self-explanatory**
  - If you need a comment to explain what a block of code does, **simply the code**
  - If the method is already extracted but you still need a comment to explain what it does, **rename the method**
  - If you need to state some rules about the required state of the system, **use Assertion**

# **Python code refactoring**

# Python code smell\* – code conventions

- Some are relevant to python code conventions\*\* , e.g.,
  - Function names should comply with a naming convention
  - Class names should comply with a naming convention
- Tool support
  - Pylint (Checking structure of the code is compliant with PEP-8)



\* <https://rules.sonarsource.com/python/type/Code%20Smell/RSPEC-1720>

\*\* <https://www.python.org/dev/peps/pep-0008/>

# Pylint demo



<https://www.youtube.com/watch?v=fFY5103p5-c>

# Python code smell\*

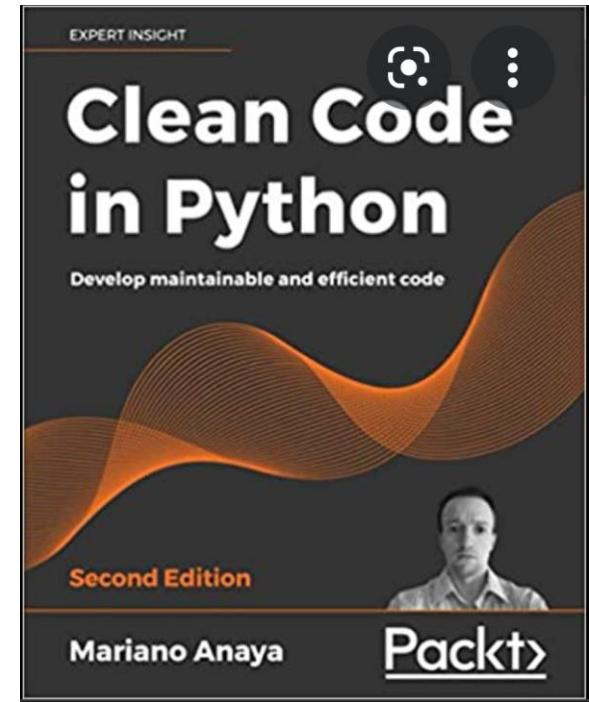
- Remove duplications
  - Redundant pairs of parentheses should be removed
  - Using features of Python, e.g., Decorator\*\*
- Reduce size
  - Functions, methods and lambdas should not have too many parameters
  - Lines should not be too long
- Speculative Generality, e.g.,
  - Nested blocks of code should not be left empty
  - Unused local variables should be removed

\* <https://rules.sonarsource.com/python/type/Code%20Smell/RSPEC-1720>

\*\*[https://www.youtube.com/watch?v=n\\_Y-\\_7R2KsY](https://www.youtube.com/watch?v=n_Y-_7R2KsY)

# Python code smell\* (cont')

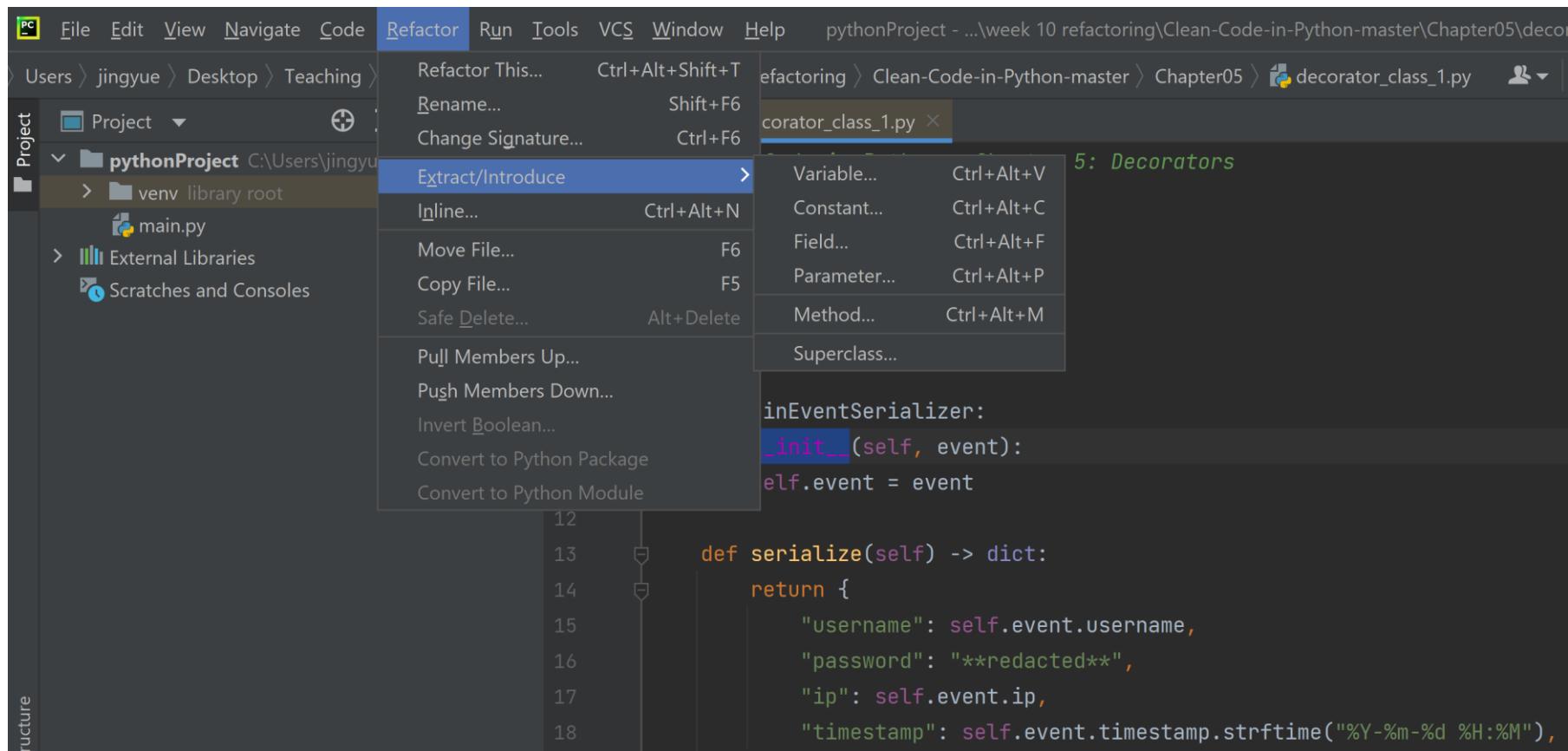
- Reduce complexity and simplify logics
  - Cognitive Complexity of functions should not be too high
  - Functions should not be too complex
  - Control flow statements "if", "for", "while", "try" and "with" should not be nested too deeply
  - Functions should not contain too many return statements



\* <https://rules.sonarsource.com/python/type/Code%20Smell/RSPEC-1720>

# Python code smell – tool support

E.g., PyCharm



# **Inspection and code review**

# What is inspection?

- Visual examination of software product
- Identify software anomalies
  - Errors
  - Code smells
  - Deviations from specifications
  - Deviations from standards
    - E.g., Java code conventions

[www.oracle.com/technetwork/java/codeconventions-150003.pdf](http://www.oracle.com/technetwork/java/codeconventions-150003.pdf)

# Why do we need inspection?

- Many software artifacts cannot be verified by running tests, e.g.,
  - Requirement specification
  - Design
  - Pseudocode
  - User manual
- Inspection reduces defect rates
- Inspection complements testing
- Inspection identifies code smells
- Inspection provides additional benefits

# Inspection finds types of defects different from testing

Number of different types of defects detected by testing vs. inspection [1]

Some defect types	Testing	Inspection
Uninitialized variables	1	4
Illegal behavior, e.g., division by zero	49	20
Incorrectly formulated branch conditions	2	13
Missing branches, including both conditionals and their embedded statements	4	10

# Inspection reduces defect rates

Removal Step	Lowest Rate	Modal Rate	Highest Rate
Informal design reviews	25%	35%	40%
Formal design inspections	45%	55%	65%
Informal code reviews	20%	25%	35%
Formal code inspections	45%	60%	70%
Modeling or prototyping	35%	65%	80%
Personal desk-checking of code	20%	40%	60%
Unit test	15%	30%	50%
New function (component) test	20%	30%	35%
Integration test	25%	35%	40%
Regression test	15%	25%	30%
System test	25%	40%	55%
Low-volume beta test (<10 sites)	25%	35%	40%
High-volume beta test (>1,000 sites)	60%	75%	85%

Source: Adapted from *Programming Productivity* (Jones 1986a), "Software Defect-Removal Efficiency" (Jones 1996), and "What We Have Learned About Fighting Defects" (Shull et al. 2002).

# Testing vs. Inspection

	Testing	Inspection
Pros	<ul style="list-style-type: none"><li>• Can, at least partly, be automated</li><li>• Can consistently repeat several actions</li><li>• Fast and high volume</li></ul>	<ul style="list-style-type: none"><li>• Can be used on all types of documents, not just code</li><li>• Can see the complete picture, not only a spot check</li><li>• Can use all information in the team</li><li>• Can be innovative and inductive</li></ul>
Cons	<ul style="list-style-type: none"><li>• Is only a spot check</li><li>• Can only be used for code</li><li>• May need several stubs and drivers</li></ul>	<ul style="list-style-type: none"><li>• Is difficult to use on complex, dynamic situations</li><li>• Unreliable (people can get tired)</li><li>• Slow and low volume</li></ul>

# Inspection provides additional benefits [2]

- Code and change understanding
- Knowledge transfer
- Increased team awareness
- Creation of alternative solutions to problems

# Code review at Google [3]

- "All code that gets submitted needs to be **reviewed by at least one other person**, and either the code writer or the reviewer needs to have readability in that language. Most people use **Mondrian** to do code reviews, and obviously, **we spend a good chunk of our time reviewing code.**"

--Amanda Camp, Software Engineer, Google

# Code reviews in OSS projects [4]

- Asynchronous and distributed review
  - Lightweight with tool support
  - Not limited to the existence of defects
  - Encourage discussing the best solution
- Frequent review
  - Most peer review happens **within hours** of completing a change
- Incremental review
  - Reviews should be of changes that are small, independent, and complete

*The potential system benefit of accepting code that hasn't been discussed by a group of experts doesn't outweigh the risks. (Linus Torvalds)*

# Tools to support inspection

- Google's Mondrian
- Facebook's Phabricator
- GitHub
- ...

# GitHub review tool

# Software reading techniques

- Unstructured: Ad hoc reading
- Semi-structured: checklist-based reading
- Structured or systematic reading
  - Defect-based reading
  - Perspective-based reading

# A sample (partial) checklist for requirement review [4]

1. Requirements specifications shall be **testable**.
2. Requirements specifications shall not **conflict** with other requirements specifications.
3. Conditional requirements specifications shall cover **all cases**.
4. Numerical values in requirements specifications **shall include physical units** if applicable.

# A sample (partial) checklist for code review [4]

1. Have resources (e.g., memory, file descriptor, database connection) properly been freed?
2. Are shared variables protected/thread-safe?
3. Is logging implemented?
4. Are comments updated and consistent with the code?
5. Is data unnecessarily copied, saved, or reloaded?
6. Is the number of cores checked before spawning threads?

# Checklist-based reading with active guidance

- Use a tailored checklist
- Guide reader how to use the checklist
  - Where to find?
  - To find what?
  - How to detect?



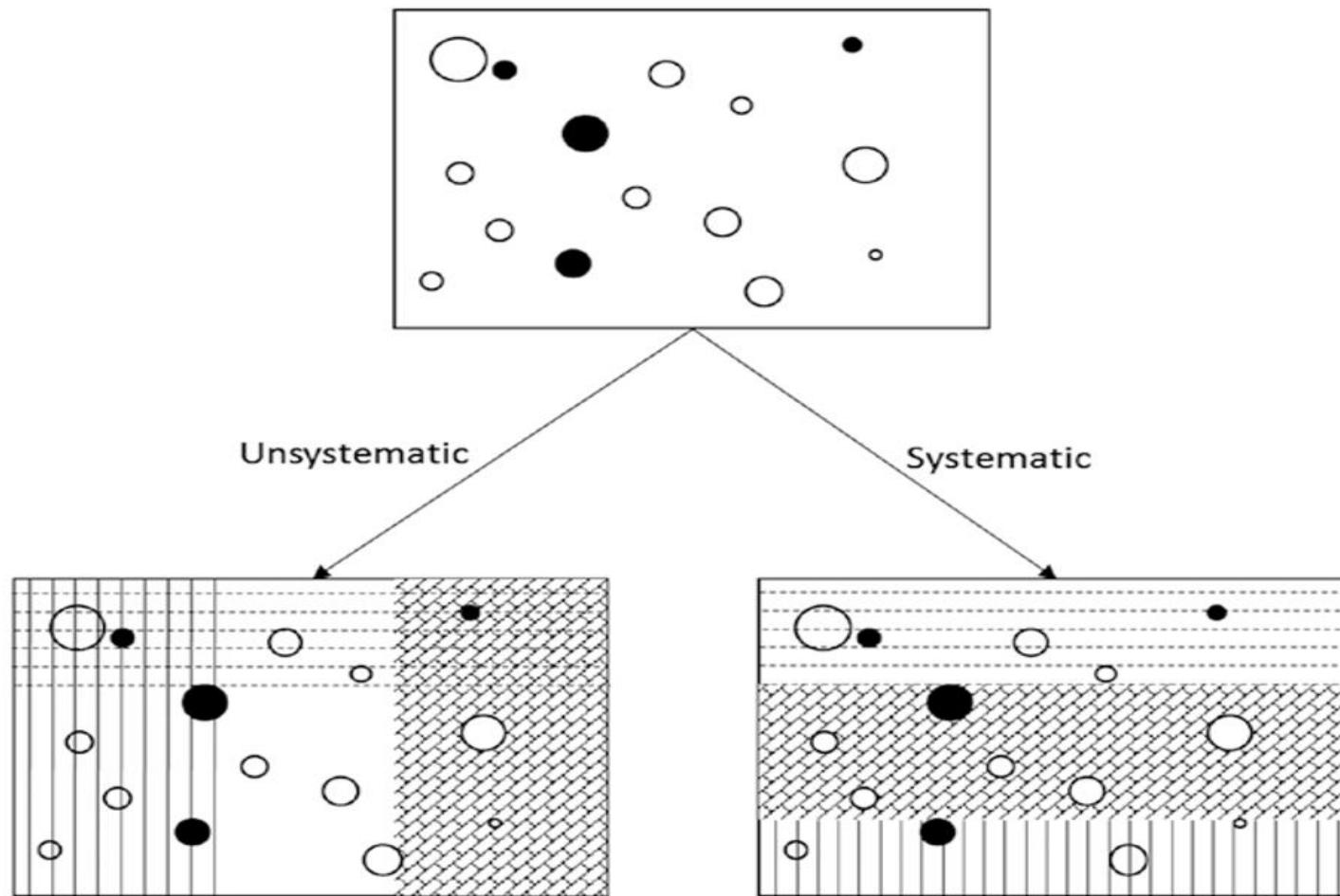
# An example of a checklist with active guidance [4]

Section	Feature	Checklist item
For each class	Inheritance	<p>Q1. Is all inheritance required by the design implemented in the class?</p> <p>Q2. Is the inheritance appropriate?</p> <p>Q3. Are all instance variables initialized with meaningful values?</p> <p>Q4. If a call to base class is required in the constructor, is it present?</p>
For each method	Constructor	<p>Q5. Are all parameters used within a method?</p> <p>Q6. Are the correct class constants used?</p> <p>Q7. Are indices of data structures operating within the correct boundaries?</p>
	Data referencing	

# Best practice of checklists

- Should be periodically revised
- Should be concise and fit on one page
- Should not be too general
- Should not be used for conventions which are better checked or enforced with software tools

# Unsystematic vs. Systematic reading



# Defect-based reading

- Each reader is given specific responsibility to discover a particular class of defects
- Specific responsibility from defect taxonomy
  - Wrong or missing assignment
  - Wrong or missing data validation
  - Wrong timing or sequencing
  - Interface problems
  - Etc.

# Perspective-based reading

- Each reader reviews software artifact from a particular perspective
- Then combine results from different perspectives
- Perspective examples
  - Novice user and expert user (GUI)
  - User, designer, tester (Requirements)

# Template for perspective-based reading

- Introduction
  - Explain stakeholder's interests
- Instruction
  - How to read and extract information
- Questions
  - For reviewers to answer by following the instruction

# **Novice use usability reading guideline [5]**

- **Introduction**
  - You review the usability of the software to make PPT from the perspective of users who are new to use software to make slides.
- **Instruction 1**
  - Scan through menu, sub-menu, help info. in GUI
- **Questions**
  - Are there instructions or online help?
  - Are items in menu, sub-menu unambiguous in meaning?

# Expert use usability reading guideline

- Introduction
  - You review the usability of the software to make PPT from the perspective of an expert who wants to use the software for making slides quickly.
- Instruction
  - Execute actions through finding and using shortcuts.
- Questions related to the instruction
  - Is it easy to find the frequently used shortcuts?
  - Is there any online help to guide the usage of shortcuts?

# Benefits of perspective-based reading

- Focused
  - Particular coverage of part of the document
- Detailed
  - Instruction and questions
- Adaptable
  - To particular software artifacts
- Tailorable
  - To organizational and project setting

# **Effective factors affecting inspection**

[6]

- Individual expertise and training
  - Most important factor
- Amount of materials
  - Not to attempt to inspect too many materials in one cycle
  - E.g., less than 125 lines of code per hour
- Team size
  - 4 inspectors may not find significantly more defects than 2 inspectors
  - 4 or 2 usually find more defects than 1
- Process and tool support
  - With good tool support, inspection meeting may not be necessary

# Summary

- We have studied
  - Why and how to do code refactoring
  - Why and how to perform code review

# References

- [1] S.S. So et al. An empirical Evaluation of Six Methods to Detect Faults in Software, *Software Testing, Verification, and Reliability*, 12, 3, 2002, pp. 155-171.
- [2] A. Bacchelli and C. Bird: Expectations, outcomes, and challenges of modern code review. *Proc. ICSE* 2013.
- [3] CSE 403 course at University of Washington, available at <https://courses.cs.washington.edu/courses/cse403/11sp/lectures/lecture15-reviews.pdf>
- [4] Y. M. Zhu: Software reading techniques. Apress, 2016.
- [5] Z. Zhang, V. Basili, and B. Shneiderman: Perspective-based usability inspection: An empirical validation of efficacy. *Journal of empirical software engineering*, 4, 1999, pp. 43-69.
- [6] S. Kollanus and J. Koskinen: Survey of software inspection research, *The Open Software Engineering Journal*, 3, 2009, pp. 15-34.
- [7] D. Rombach et al. Impact of research on practice in the field of inspections, reviews, and walkthroughs, *ACM SIGSOFT SE notes*, 33, 6, 2008, pp. 26-35.

# **Software estimation – cost and quality**

Mar 2022

Anh Nguyen Duc



# Agenda

---

- ❖ The role of project estimation
- ❖ Traditional approaches for effort estimation
- ❖ Machine Learning in effort prediction
- ❖ Defect prediction models
- ❖ Within vs. Between project prediction
- ❖ Just-in-time defect prediction

# What is a Project Estimate?

A declaration about needed

- cost and
- time
- quality
- human skill
- for delivering the project scope
- What else?



How productive is this team/ developer  
in this project?

# Importance of project estimation

- ✓ Ressource allocation decisions
- ✓ Basis for the decision to start (or not to start)a project
- ✓ Foundation for project planning and set-up (business case)
- ✓ Foundation for project controlling
- ✓ If project time is a given, number of ressources can be determined
- ✓ Owner of an estimate is an indication about who is taking the project risk
- ✓ Decision and ressource allocation implications => Estimates are often part of political games
- ✓ Estimating is a core task of project management

# Challenges for estimation

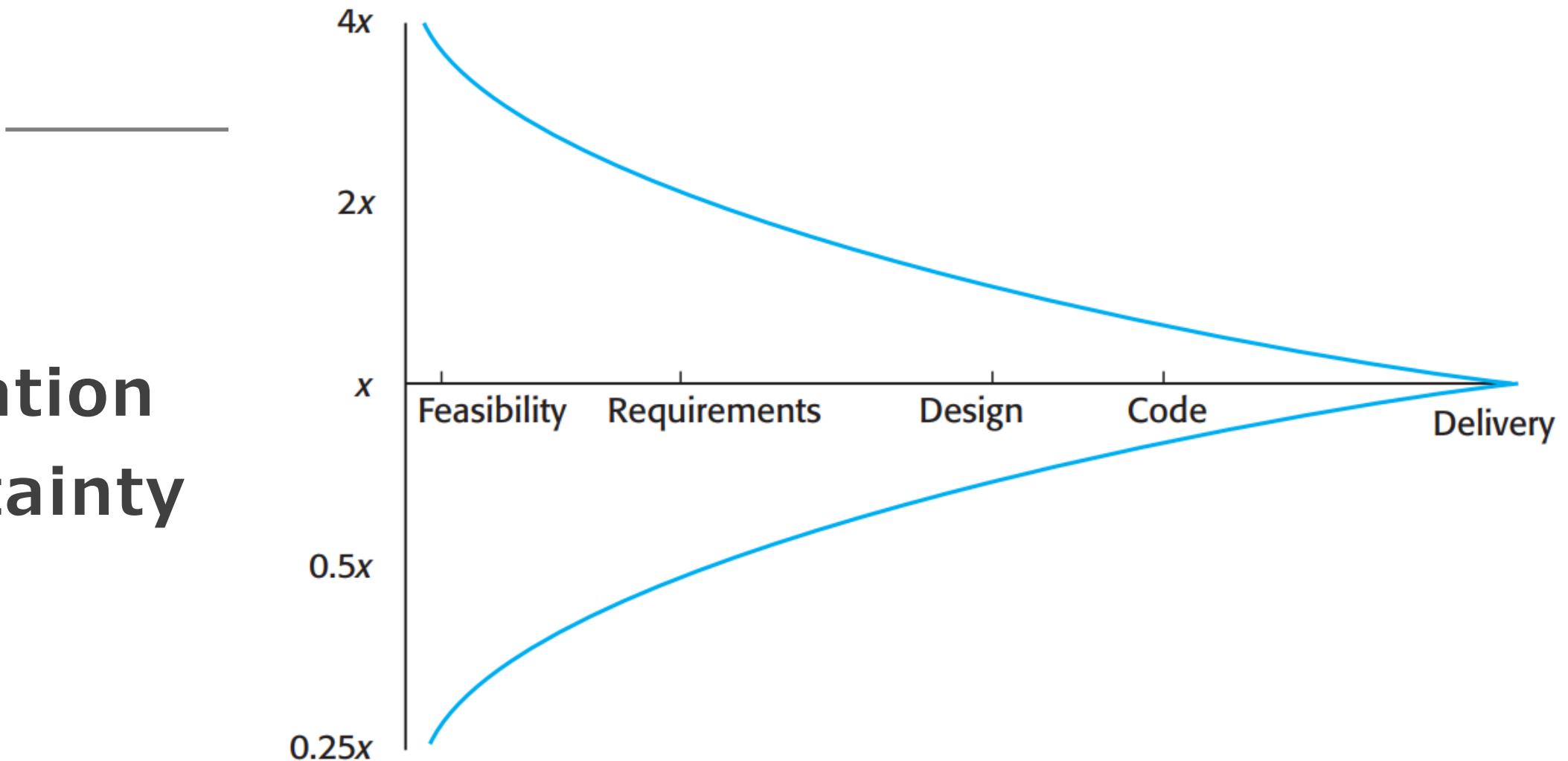
- ✓ Incomplete knowledge about:
  - ❖ *Project scope and changes*
  - ❖ *Prospective resources and staffing*
  - ❖ *Technical and organizational environment*
  - ❖ *Infrastructure*
  - ❖ *Feasibility of functional requirements*
- ✓ *Comparability of projects in case of new or changing technologies, staff, methodologies*
- ✓ *Learning curve problem*
- ✓ *Different expectations towards project manager*

# Challenges for estimation

- ✓ Estimation is too low
  - ✓ *Scope and tasks(WBS) incomplete/unknown*
- ✓ Estimation is too high
  - ✓ *Political / human reasons*
  - ✓ *Learning curve*
- ✓ New technologies can make new parameters necessary

# Challenges for estimation

Estimation  
uncertainty



# **Estimating Effort**

# Cost Estimation

- ✓ Effort costs (the dominant factor in most projects)
  - ✓ – salaries of engineers involved in the project
  - ✓ – costs of building, heating, lighting
  - ✓ – costs of networking and communications
  - ✓ – costs of shared facilities (e.g library, staff restaurant, etc.)
  - ✓ – costs of pensions, health insurance, etc.
- ✓ Other costs
  - ✓ – *Hardware and software costs*
  - ✓ – *Travel and training costs*

# Cost Estimation

- ✓ Staff categories (based on experience, qualification and skills), for example:
  - ✓ teamlead, junior business analyst, senior business analyst, junior programmer, senior programmer, subject matter expert
- ✓ Cost rate: Cost per person per day
  - ✓ 2 alternatives for cost rate:
    - ✓ *Single cost rate for all types (no differentiation necessary)*
    - ✓ *Assign different cost rates to different categories*
- ✓ Personnel cost = person days x cost rate

## Timepris for selvstendige

Våre Talenter sprer seg over de aller fleste fagdisipliner innen avhenger av visse faktorer som for eksempel utdanningsnivå. Som tommelfingerregel er gjennomsnittlig timepris for en selvstendig utvikler per spesiell

**Senior frontend utvikler:** 1100-1300 NOK

**Junior frontend utvikler:** 800-1000 NOK

**Fullstack utvikler:** 1000-1200 NOK

**Backend utvikler:** 1000-1200 NOK

**App-utvikler:** 800-1000 NOK

**DevOps:** 1300-1450 NOK

**Tech Lead/Scrum:** 1300-1450 NOK

Men, ikke alle prisnivåer er hugget i stein. Vi diskuterer oss fre

We use cookies to improve your experience. By proceeding, you accept

# **Estimating effort- Basic principles**

---

Select an estimation model (or build one first)

Evaluate known information: project scope, resources, software process (for example documentation requirements), system components

Feed this information as parametric input data into the model

Model converts the input into an estimate about the effort

# Basic of an Estimation model



**Examples:**

Data Input

Size & Project Data

System Model

Software Process

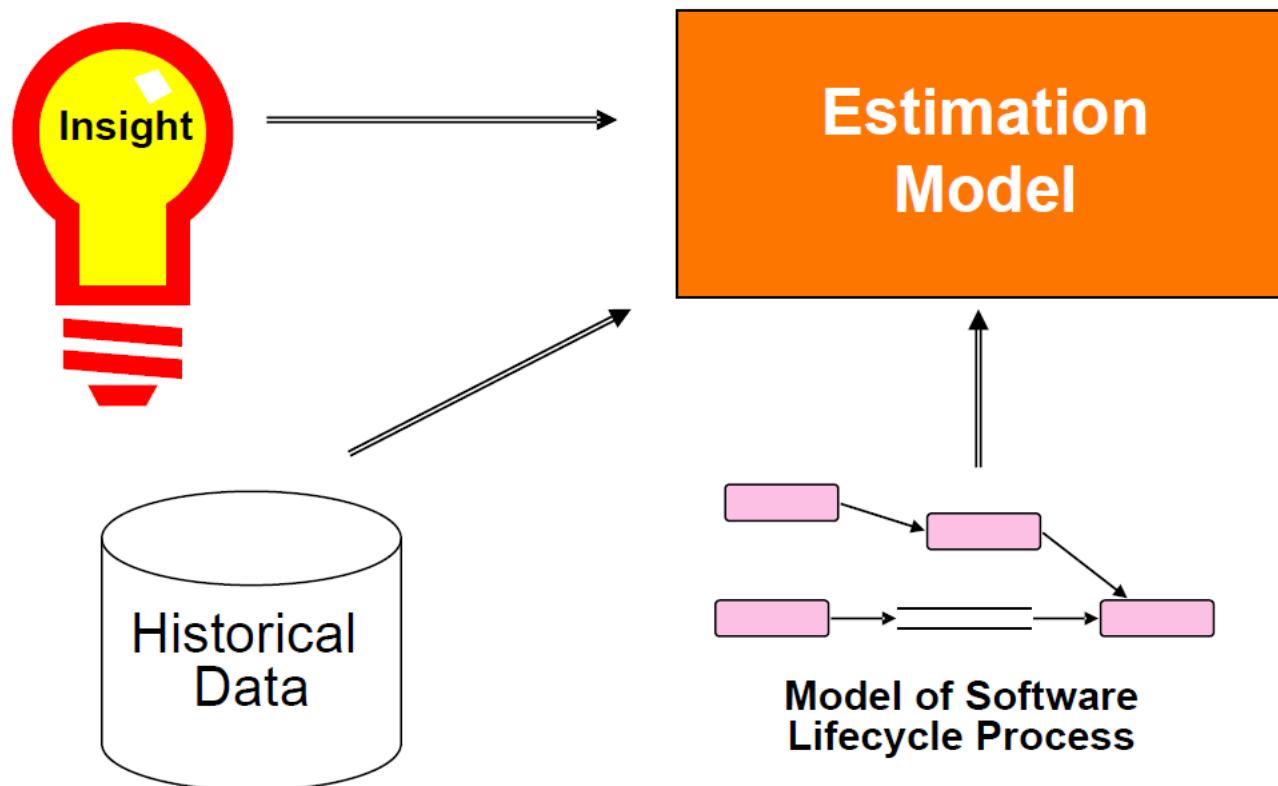
Estimate

Effort & Schedule

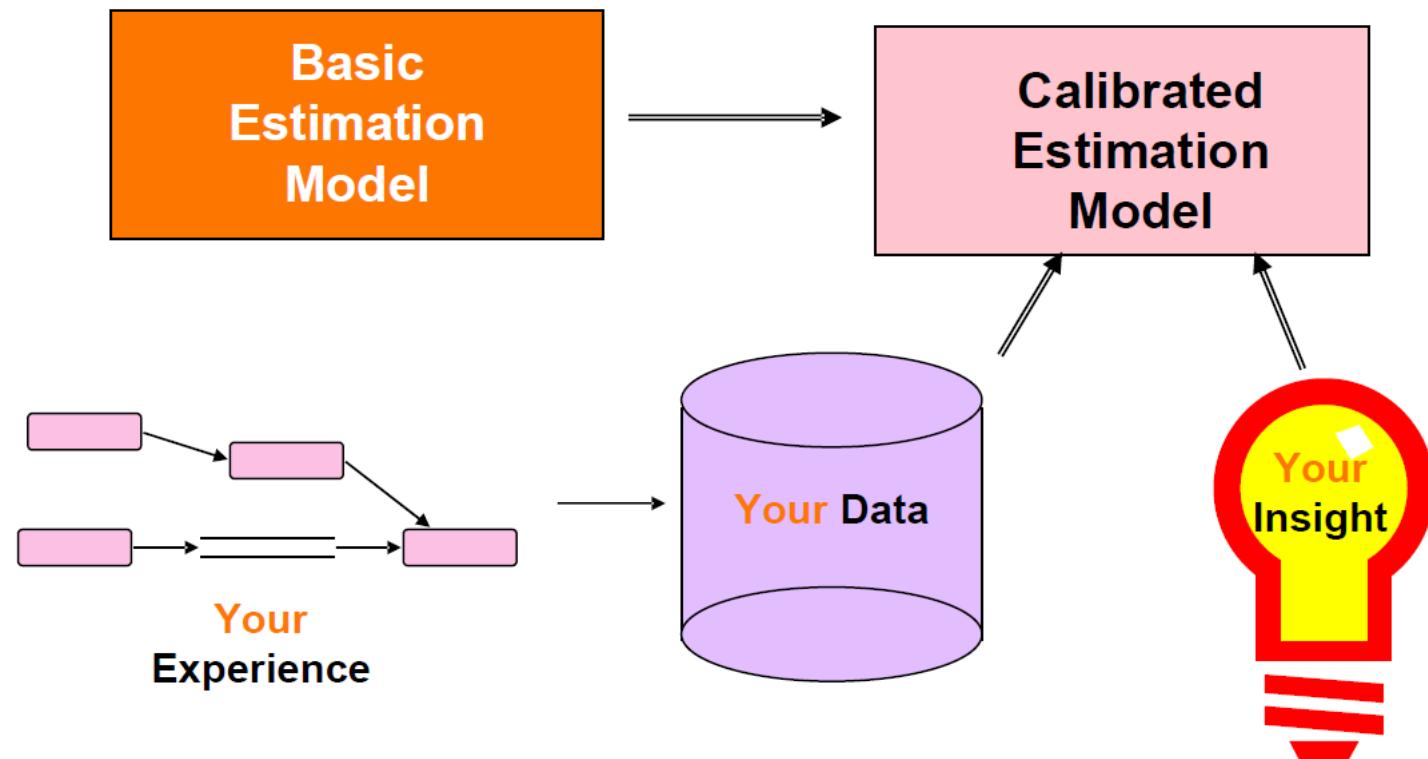
Performance

Cycle Time.

# How to build an estimation model?



# Calibrating the model



# **Top-Down and Bottom-Up Estimation**

---

Two common approaches for estimations

## **Top-Down Approach**

- Estimate effort for the whole project
- Breakdown to different project phases and work products

## **Bottom-Up Approach**

- Start with effort estimates for tasks on the lowest possible level
- Aggregate the estimates until top activities are reached

# Top-Down and Bottom-Up Estimation

## Top-Down Approach

- Normally used in the planning phase when **little information** is available how to solve the problem
- Based on **experiences** from similar projects
- Not appropriate for project controlling (**too high-level**)
- **Risk add-ons** usual as result tends to be too low

## Bottom-Up Approach

- Normally used after activities are **broken down to task level** and estimates for the tasks are available
- Result can be used for project controlling (**detailed level**)
- **Smaller risk add-ons** (tends to be too high)
- Often a **mixed approach** with recurring estimation cycles is used.

# Estimation Techniques

---

- ✓ Expert judgement
- ✓ Estimation by analogy
- ✓ Parkinson's Law
- ✓ Pricing to win
- ✓ Lines of code
- ✓ Function point analysis
- ✓ Algorithmic cost modelling
- ✓ COCOMO

# Expert judgement

= Guess from experienced people

---

- Mostly used **top-down** for the whole project, but also for some parts of a bottom-up approach
- Relatively **cheap** estimation method. Can be accurate if experts have direct experience of similar systems
- No better than the **participants**
- Result **justification** difficult
- Very inaccurate if there are no experts!

# Estimation by analogy

(do you work on the same project before?)

---

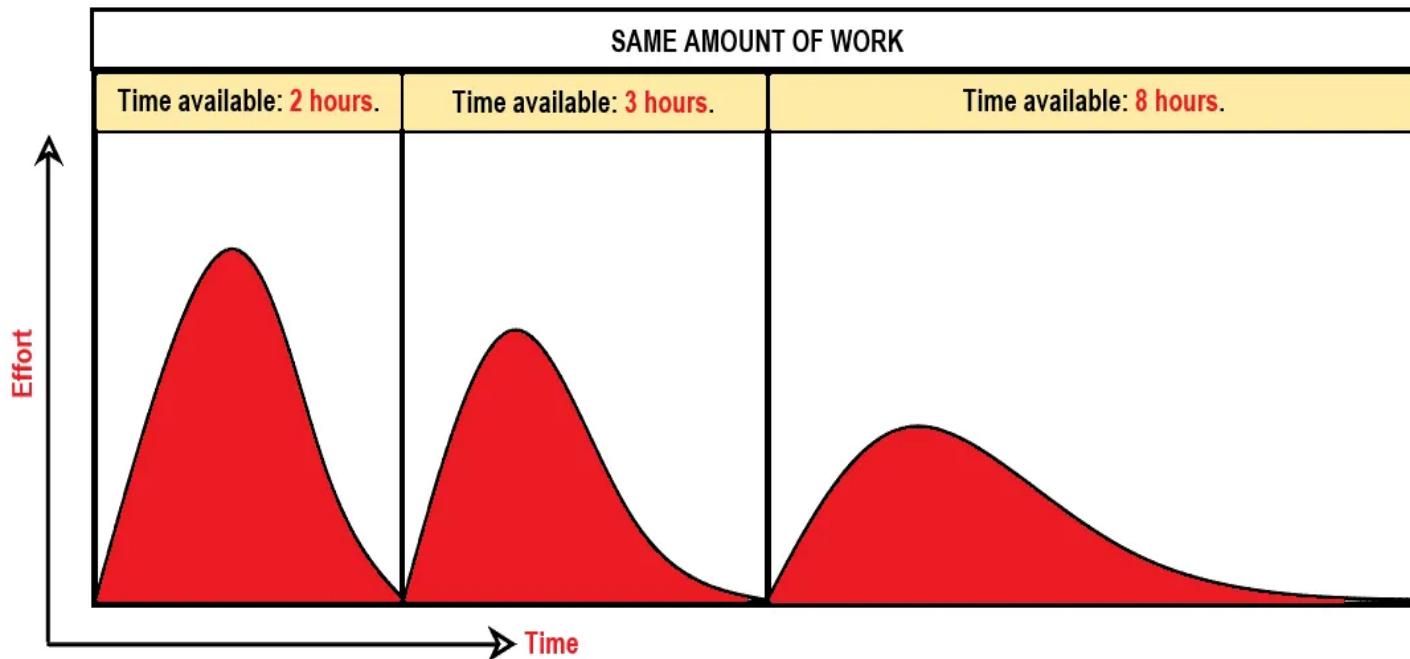
- The cost of a project is computed by comparing the project to a similar project in the same application domain
- Accurate if project data available
- Impossible if no comparable project has been tackled. Needs systematically maintained cost database

# Parkinson's Law

*“work expands to fill the time allotted for its completion”*

---

- The project costs whatever resources (people, money, time) are available
- No overspend
- System is usually unfinished



# Pricing to win

*The project costs whatever the customer has to spend on it*

---

- You get the contract
- The probability that the customer gets the system he or she wants is small. Costs do not accurately reflect the work required

# Line of Code (LOC)

Traditional way for estimating application size (FORTRAN and assembler -> line-oriented languages)

<b>Function</b>	<b>Estimated LOC</b>
User interface and control facilities (UICF)	2,300
Two-dimensional geometric analysis (2DGA)	5,300
Three-dimensional geometric analysis (3DGA)	6,800
Database management (DBM)	3,350
Computer graphics display facilities (CGDF)	4,950
Peripheral control function (PCF)	2,100
Design analysis modules (DAM)	8,400
<i>Estimated lines of code</i>	<i>33,200</i>

# Line of Code (LOC)

Traditional way for estimating application size (FORTRAN and assembler -> line-oriented languages)

Advantage: Easy to do

Disadvantages:

- No standard definition for Line of Code (logical versus physical)
- Of no help given a written project scope or functional design
- You get what you measure: If the number of lines of code is the primary measure of productivity, programmers ignore opportunities of reuse
- Multi-language environments: Hard to compare mixed language projects with single language projects

The use of lines of code metrics for productivity should be regarded as professional malpractice (Caspers Jones).

# Function Point Estimation

---

Based on FP metric for the size of a product

Based on the number of inputs (Inp), outputs (Out), inquiries (Inq), master files (Maf), interfaces (Inf)

Step 1: Classify each component of the product (Inp, Out, Inq, Maf, Inf) as simple, average average, or complex

- Assign the appropriate number of function points
- The sum of function pointers for each component gives UFP (unadjusted function points)

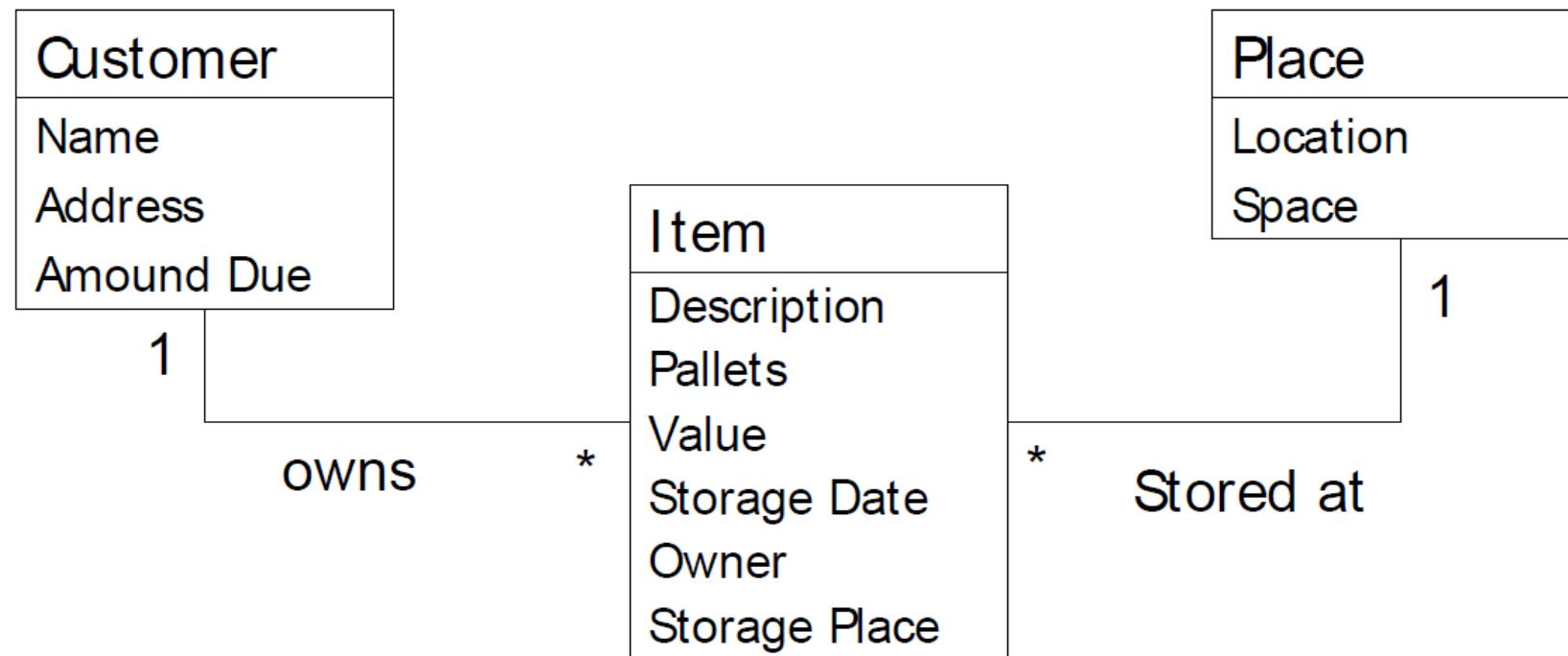
# Function Point Estimation

---

- Step 2: Compute the technical complexity factor (TCF)
  - Assign a value from 0 ("not present") to 5 ("strong influence throughout") to each of 14 factors such as transaction rates, portability
  - Add the 14 numbers: This gives the total degree of influence (DI)
    - $TCF = 0.65 + 0.01 \times DI$
    - *The technical complexity factor (TCF) lies between 0.65 and 1.35*
- Step 3 The number of function function points (FP) is:
  - $FP = UFP \times TCF$

# Function Point Estimation (example)

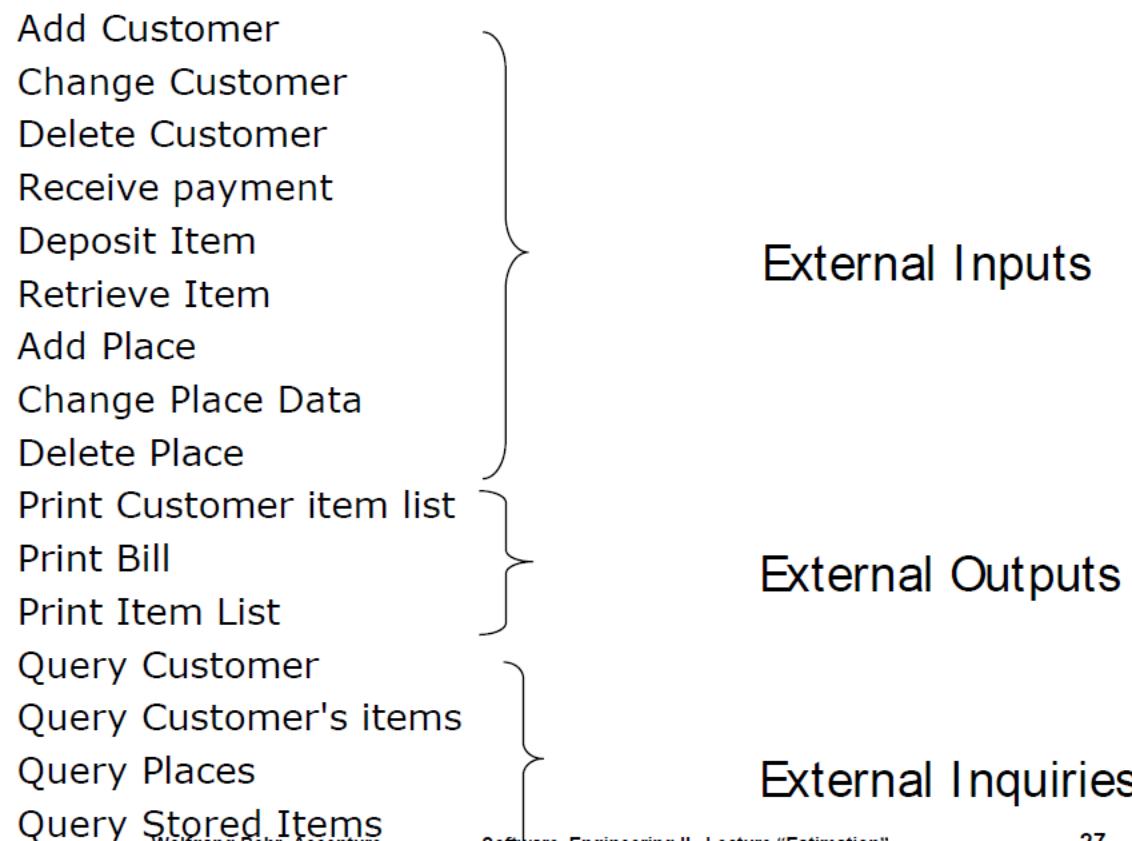
---



# Function Point Estimation (example)

---

## Mapping Functions to Transaction Types



# Function Point Estimation (example)

## Calculate the Unadjusted Function Points

Function Type	Number	Weight Factors			=	<input type="text"/>
		simple	average	complex		
External Input (EI)	<input type="text"/>	x 3	4	6	=	<input type="text"/>
External Output (EO)	<input type="text"/>	x 4	5	7	=	<input type="text"/>
External Queries (EQ)	<input type="text"/>	x 3	4	6	=	<input type="text"/>
Internal Datasets (ILF)	<input type="text"/>	x 7	10	15	=	<input type="text"/>
Interfaces (EIF)	<input type="text"/>	x 5	7	10	=	<input type="text"/>
Unadjusted Function Points (UFP) =						<input type="text"/>

The unadjusted function points are adjusted with general system complexity (GSC) factors

# Function Point Estimation (example)

After the GSC factors are determined, compute the

$$\text{Value Added Factor (VAF): } \text{VAF} = 0.65 + 0.01 * \sum_{i=1}^{14} \text{GSC}_i \quad \text{GSC}_i = 0, 1, \dots, 5$$

Function Points (FP) = Unadjusted Function Points (UFP) \* VAF

PF (Performance factor) = Number of function points that can be completed per day

Effort = FP / PF

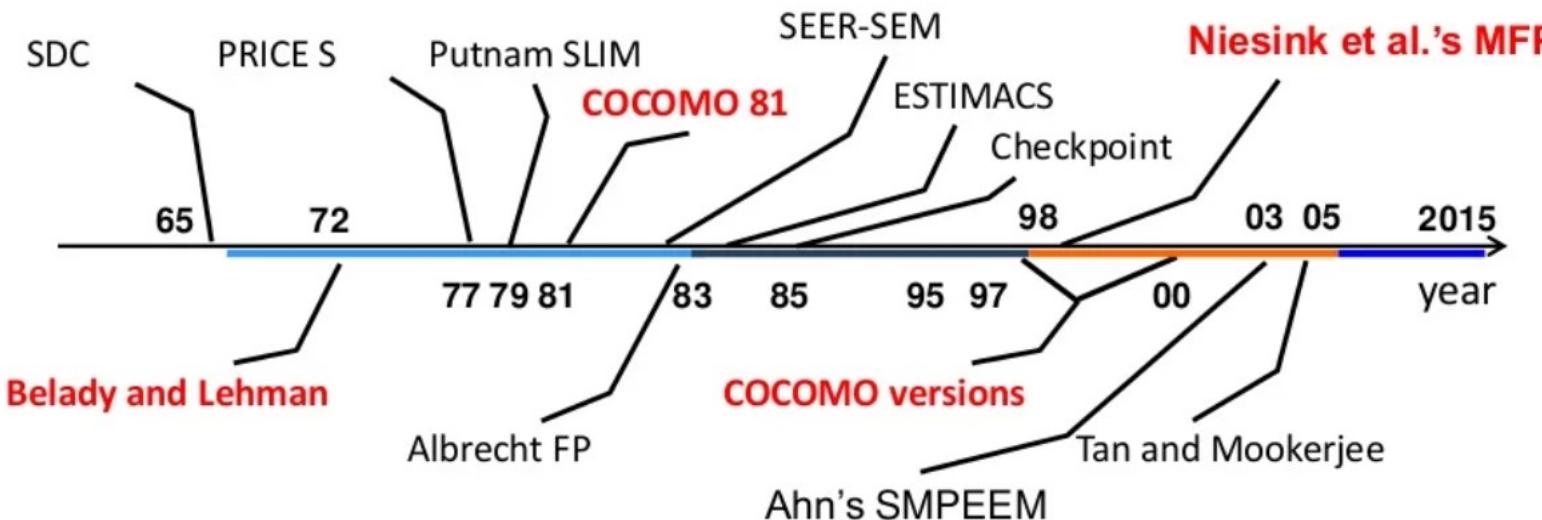
The unadjusted function points are adjusted with general system complexity (GSC) factors

# Function Point Estimation (challenges)

- Complete description of functions necessary
  - Often not the case in early project stages -> especially in iterative software processes
- Internal functions (algorithms) rather underestimated, as model is based on user oriented requirements and functions
- Only complexity of specification is estimated
  - Implementation is often more relevant for estimation
- High uncertainty in calculating function points:
  - Weight factors are usually deducted from past experiences (environment, used technology and tools may be out-of-date in the current project)
- Not suitable for project controlling.

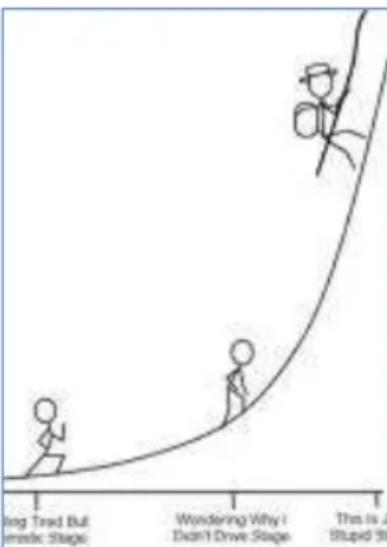
# Algorithmic cost modelling

- Most models before 2005 are basing on some form of parametric formulas
- After 2005, more advanced prediction and classification approaches



- 65-85: the search for right parametric forms
- 85-95: advances in size and complexity metrics
- 95-05: proliferation of software development styles
- 05-15: advances in prediction and classification models

# Belady and Lehman (1972)



$$\text{Effort} = p + K^{c-d}$$

- *Effort*: total maintenance effort
- $p$ : productive effort, including analysis, design, code testing
- $d$ : degree of maintenance team familiarity with the software
- $c$ : complexity caused by lack of structured design and document
- $K$ : empirical constant, depends on the environment

# The COCOMO model

- Effort (person month) is a function of size (LOC)
- Exists in three stages
  - Basic - Gives a 'ball-park' estimate based on product attributes
  - Intermediate
  - Modifies basic estimate using project and process attributes
  - Advanced - Estimates project phases and parts separately

$$\text{Effort} = b \times \text{Size}^c$$

- Organic: small teams develops software in known environment ( $b=2.4, c=1.05$ )
- Embedded: inflexible and constrained environment ( $b=3.6, c=1.20$ )
- Semidetached: varying levels of team experience working on larger projects ( $b=3.0, c=1.12$ )

$$\text{Effort} = b \times \text{Size}^c \times \text{EAF}$$

- Intermediate model
- $b, c$ : calibrated factors
- EAF: effort adjustment factor

# COCOMO suite of models (1981-2007)

## Advantages

- Repeatable estimations
- Easy to modify input
- Easy to customize and refine formula

## Disadvantages

- Subjective inputs
- Unable to deal with exceptional conditions
- Mainly designed for waterfall
- Needs historical data for calibration

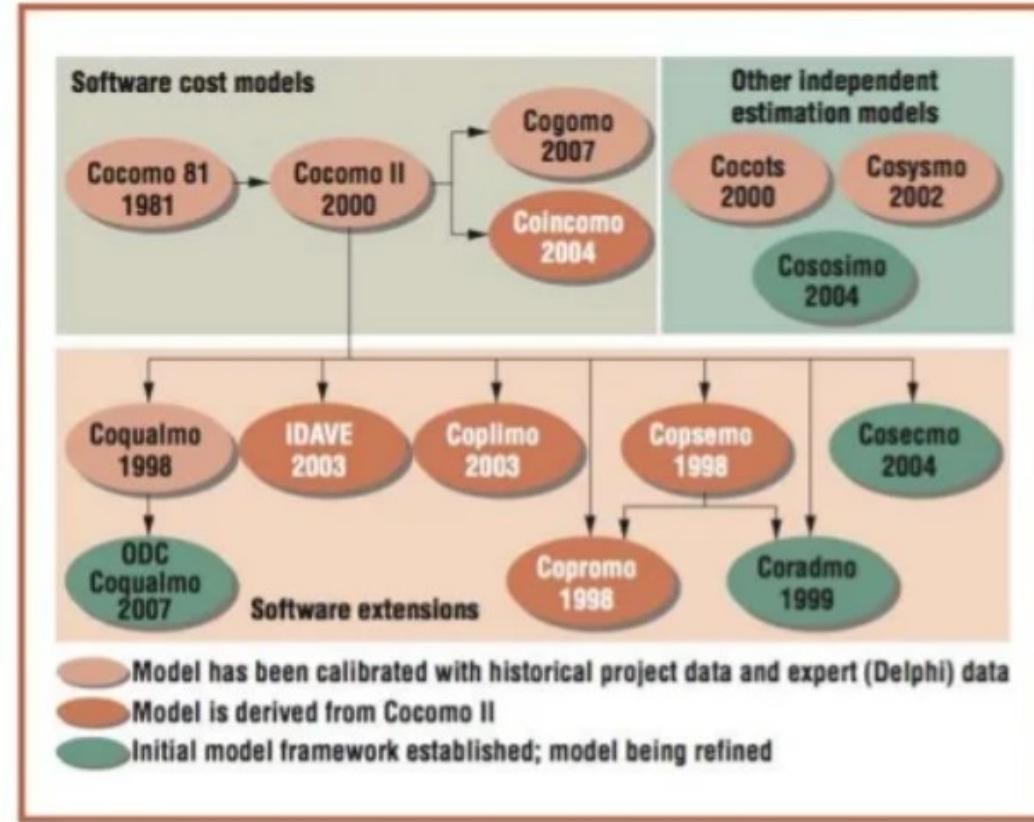


Figure from Boehm et al. 2008



# **Machine Learning-based Estimating Effort**

# Types of ML techniques

---

Case-Based Reasoning (CBR)

Decision Trees (DT)

Bayesian Networks (BN)

Support Vector Regression (SVR)

Genetic Algorithms (GA)

Genetic Programming (GP)

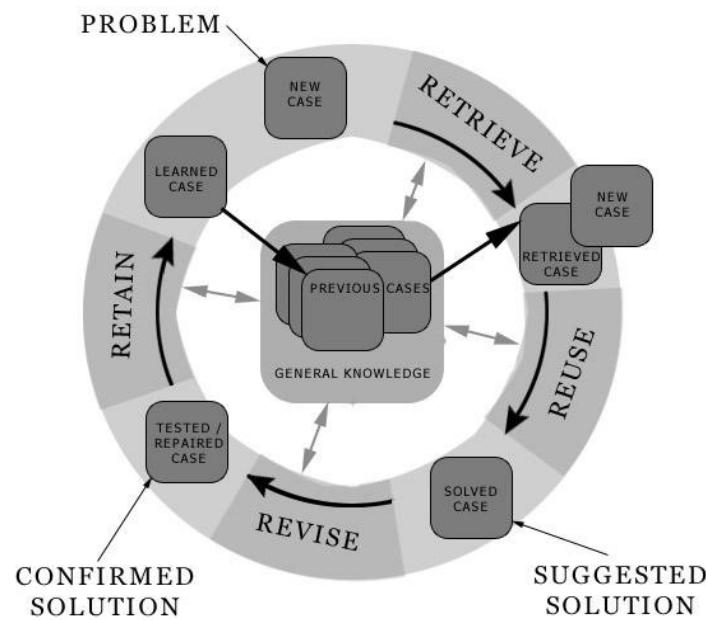
Association Rules (AR)

Artificial Neural Networks (ANN)

J. Wen, S. Li, Z. Lin, Y. Hu, and C. Huang, "Systematic literature review of machine learning based software development effort estimation models," *Information and Software Technology*, vol. 54, no. 1, pp. 41–59, Jan. 2012

# Case-Based Reasoning (CBR)

- Finding a similar past case, and reusing it in the new problem situation
- CBR is the most widely used method in the software estimation practice



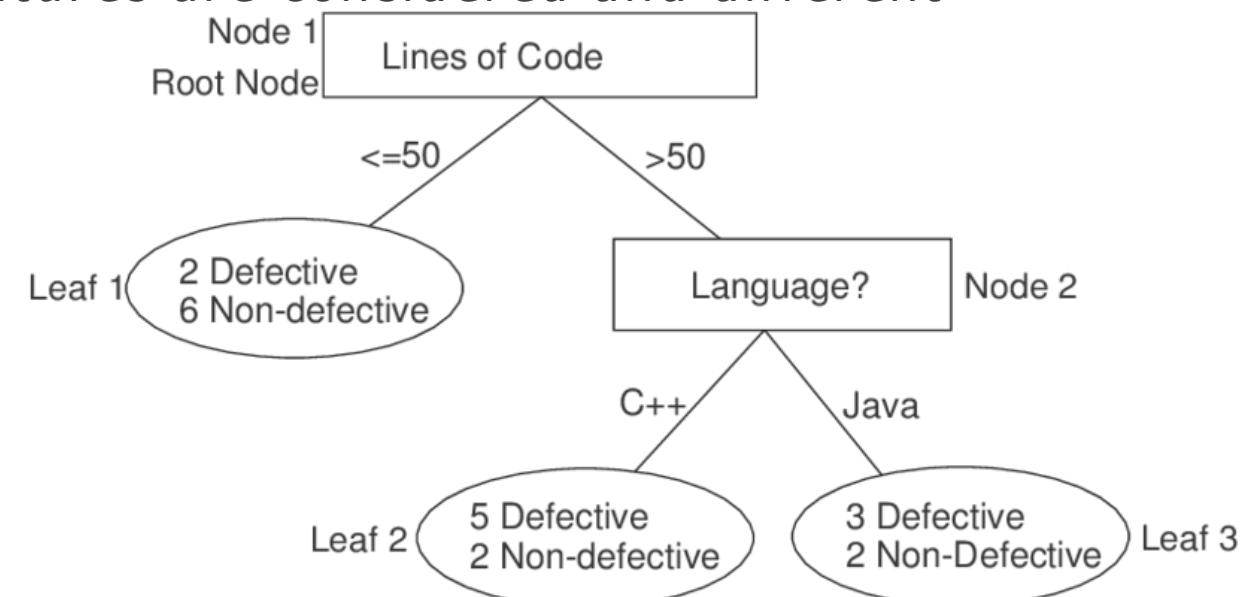
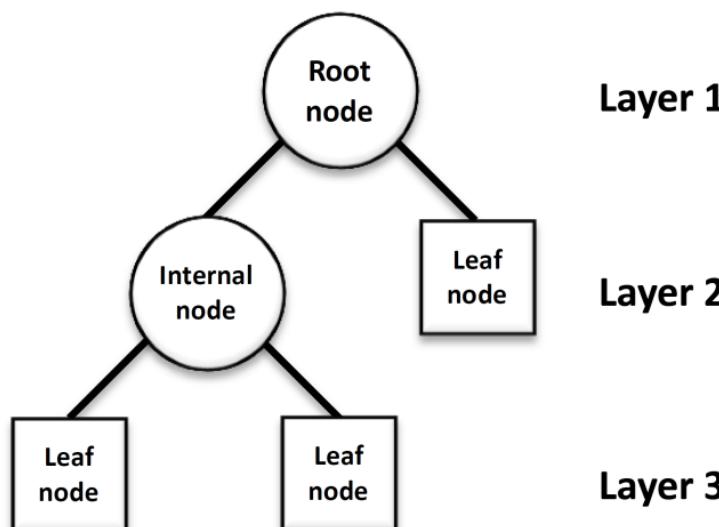
## Similarity measure:

- Manhattan distance
- Euclidean distance
- Minkowski distance
- grey relational coefficient
- Gaussian distance
- Mahalanobis distance

ID	Feature	Full name or explanation
1	TeamExp	Team experience in years
2	ManagerExp	Project manager's experience in years
3	YearEnd	Year of completion
4	Length	The length of project
5	Transactions	Number of transaction processed
6	Entities	Number of entities
7	PointsAdjust	Adjusted function points
8	PointsNonAjust	Unadjusted function points
9	Envergure	Complex measure derived from other factors
10	Effort	Measured in person-hours

# Decision Trees (DT) / Classification and regression trees (CART)

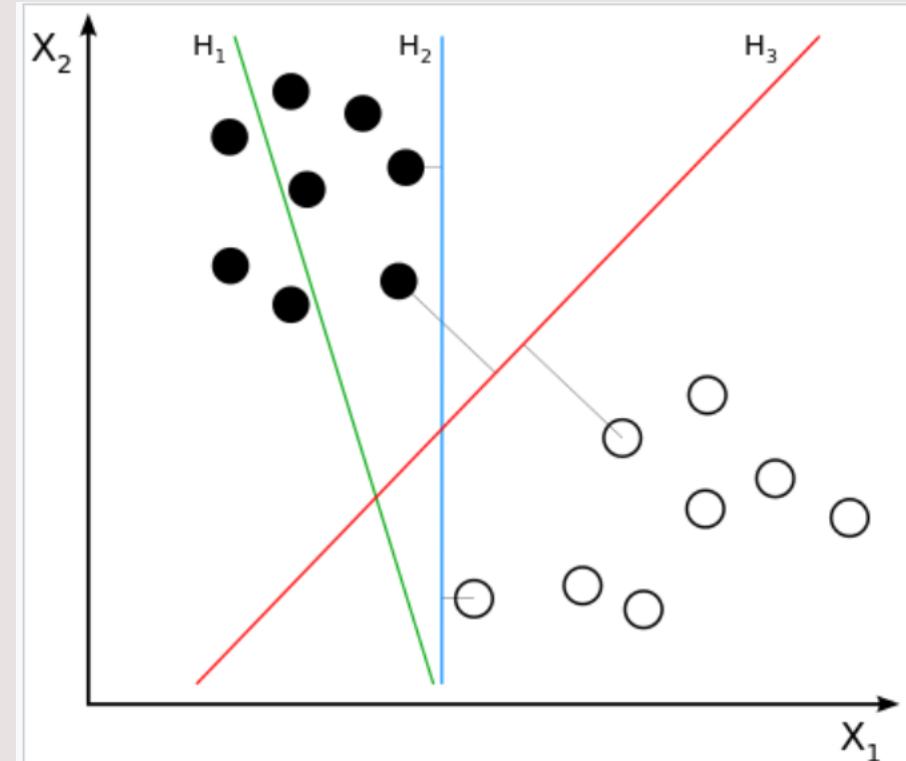
- relatively easy to understand and are also very effective
- DT splits the original dataset to sub datasets which are more homogeneous.
- Recursive Binary Splitting: *all the features are considered and different split points are tried*



When to stop splitting?

# Support Vector Machine (SVM)

- Builds a model that assigns new case to one category or the other (binary classifier)
- Map training cases to points in space to maximise the width of the gap between the two categories.
- A new case is then mapped into that same space and predicted to belong to a category based on which side of the gap they fall.



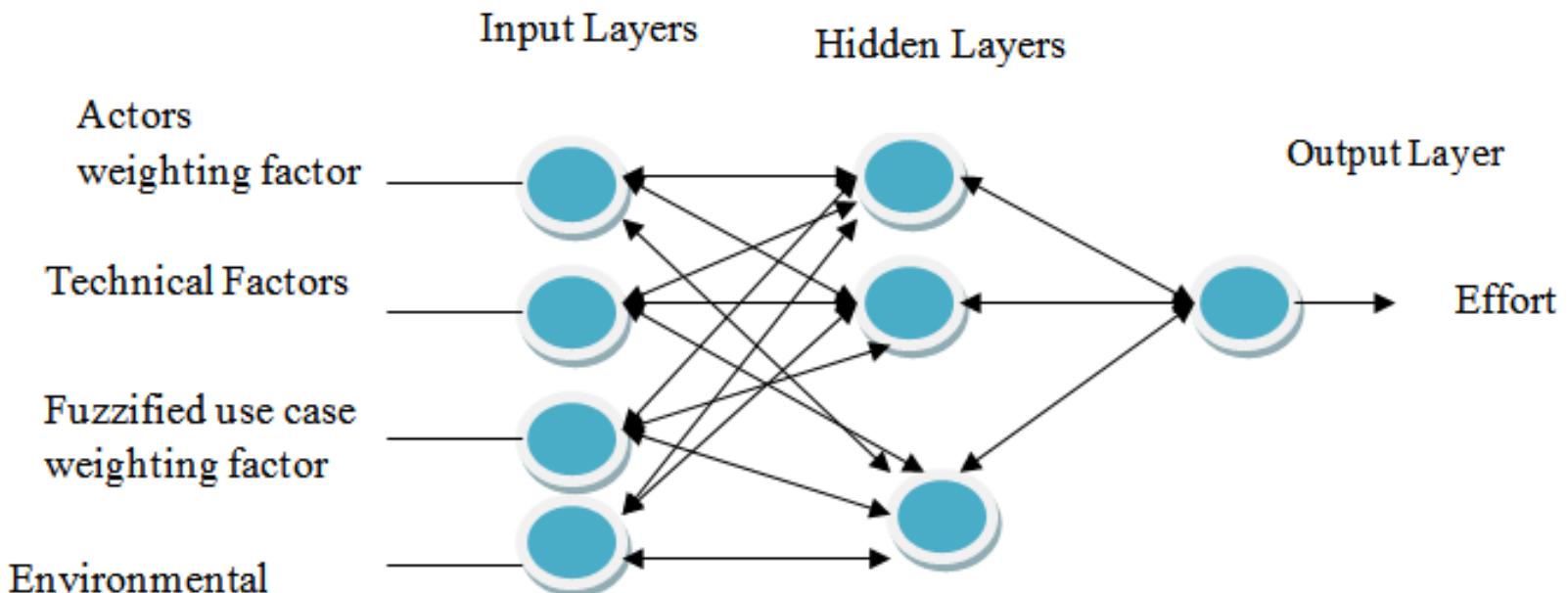
$H_1$  does not separate the classes. □

$H_2$  does, but only with a small margin.

$H_3$  separates them with the maximal margin.

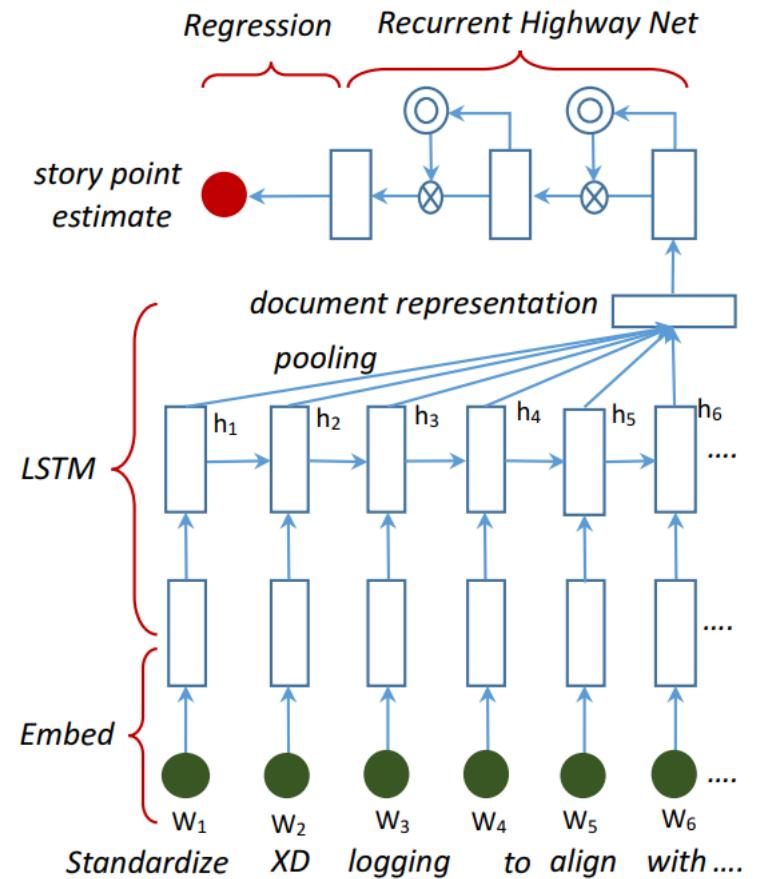
# Artificial Neural Network (ANN)

- Input layers: size of the software in function points or Lines Of Code (LOC), and other effort drivers such as complexity of the software, database size, experience etc.
- Each node has its own linear regression model, composed of input and output variables



# Deep Learning

- Model for Agile projects, predict efforts for the next Iteration
- Adopt Long Short Term Memory / Recurrent Neural Network
- Input: SP: story points, TD length: the number of words in the title and description of an issue, LOC: line of code



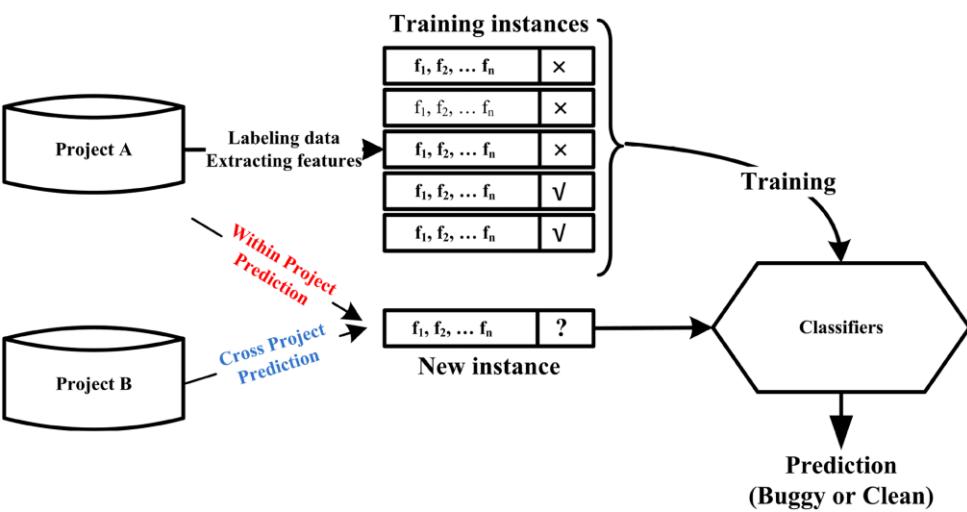


# **Defect Prediction**

# Importance of defect prediction

---

- Software defect: flaw or imperfection in software work or software process
- A defect is also referred as a fault or a bug
- Every software project has bugs, which we hope to reduce.
- Defect prediction is essential in the field of software quality and software reliability
- Focus on predicting those defects that affect project and product performance



Urgent Severe	Urgent Not Severe
Not Urgent Severe	Not Urgent Not Severe

# Pareto rule (20-80 rule)

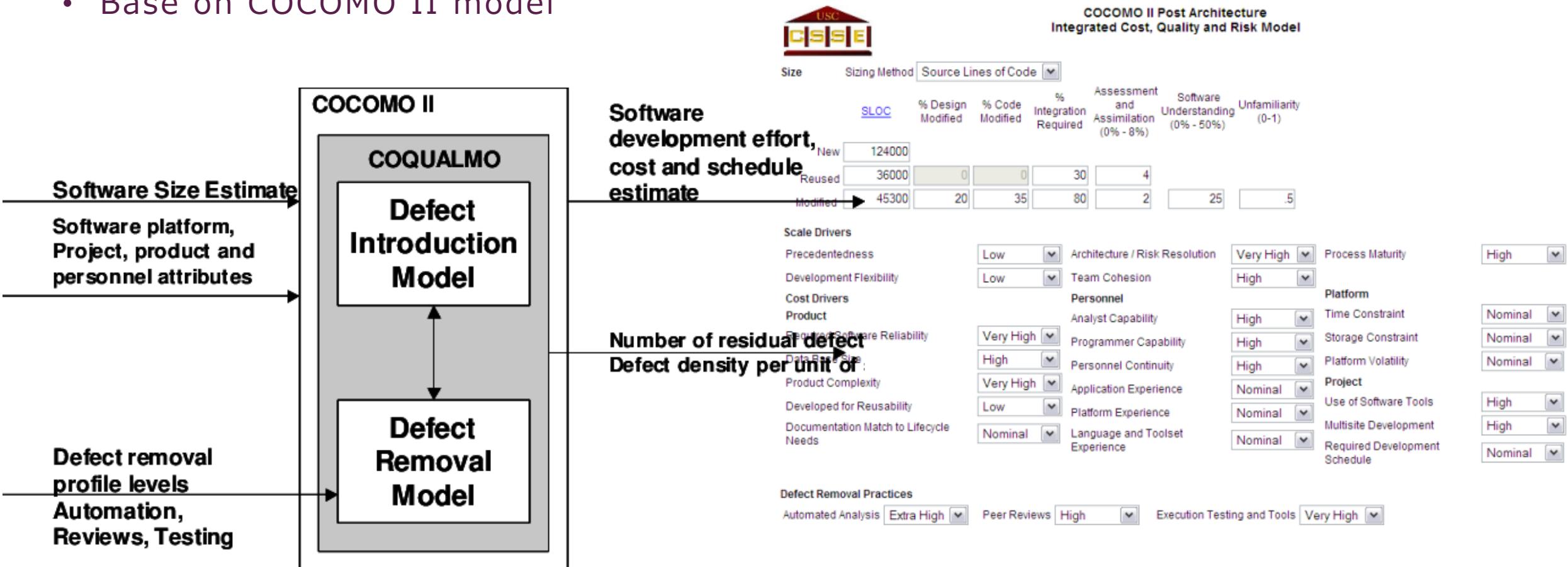
---

Most defects (80%) exist in few modules (20%)

Defect data is usually imbalanced

# Early days prediction models - COQUALMO

- predicting number of residual defects/KSLOC (Thousands of Source Lines of Code) or defects/FP (Function Point) in a software product
- Base on COCOMO II model



# Algorithmic approaches - Basic principles

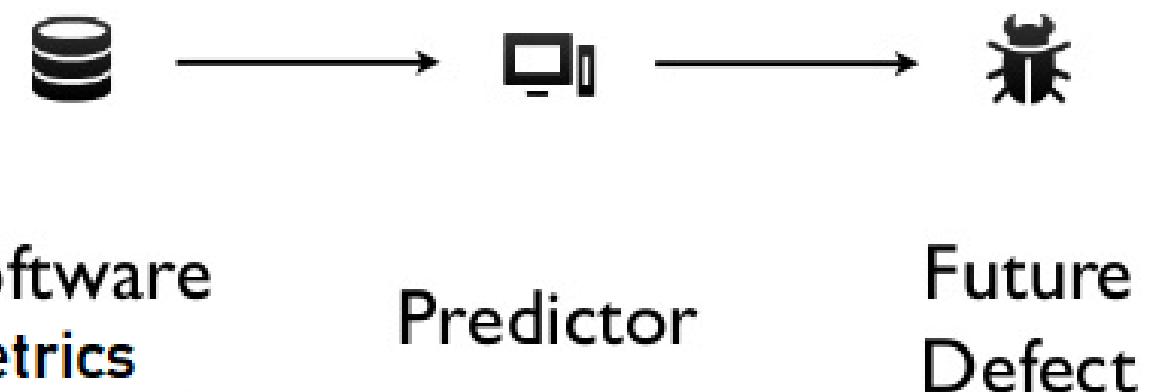
---

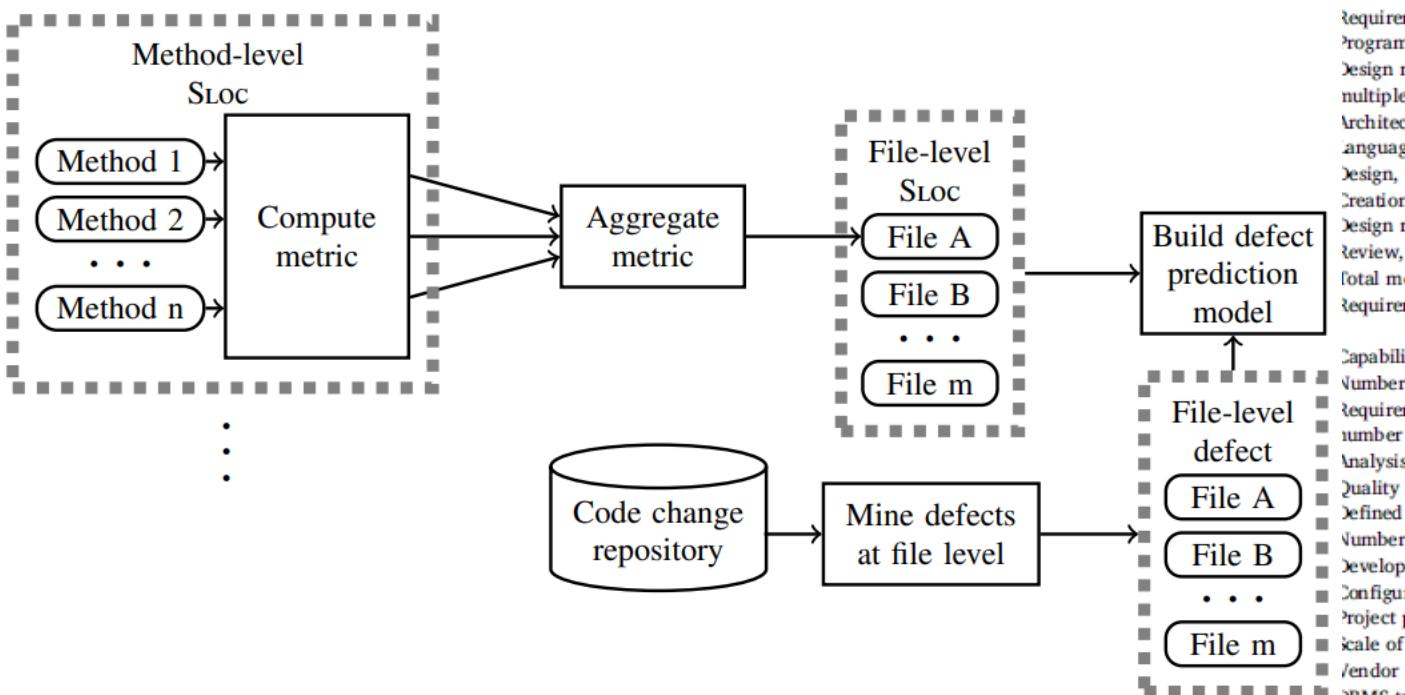
Software defect prediction – mainly base on historical data

When to use: (1) use for planning (2) in-process monitoring of defect discovery numbers

What to predict?

- Number of defect
- Defect density
- Defect proneness

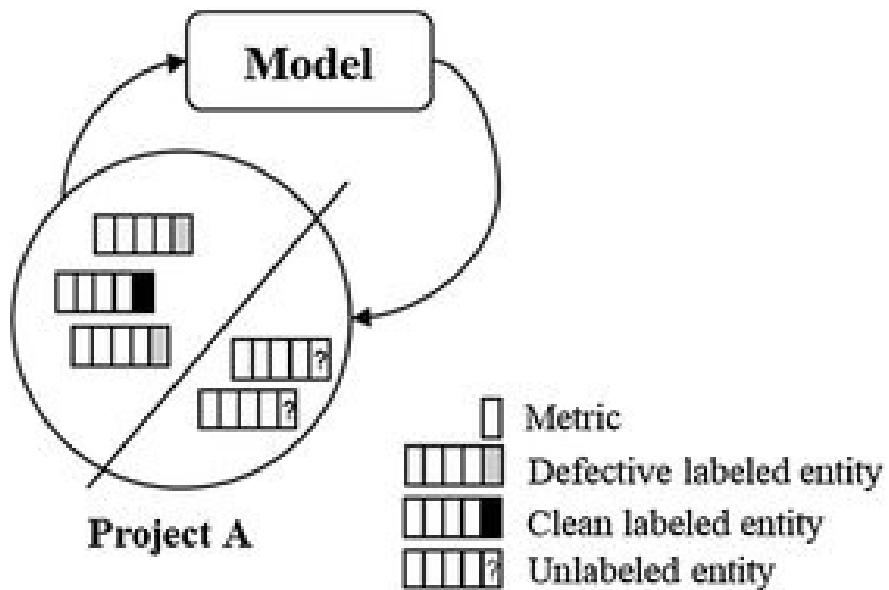




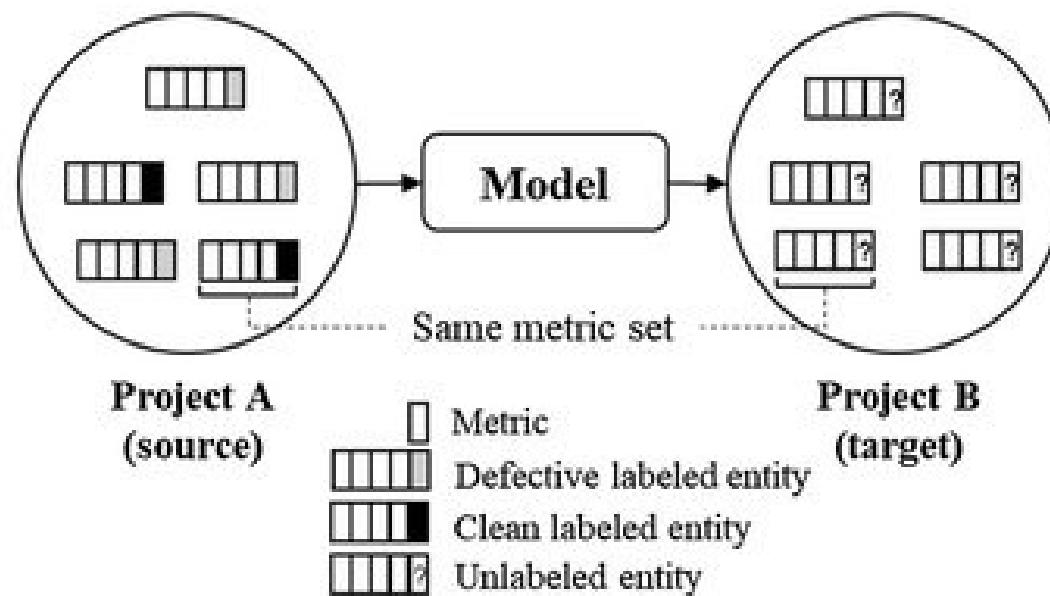
R. Özakıncı and A. Tarhan, "Early software defect prediction: A systematic map and review," *Journal of Systems and Software*, vol. 144, pp. 216–239, Oct. 2018, doi: [10.1016/j.jss.2018.06.025](https://doi.org/10.1016/j.jss.2018.06.025).

# With-in vs. Cross-project prediction

---



(a) Within-Project Defect Prediction (WPDP)



(b) Cross-Project Defect Prediction (CPDP)

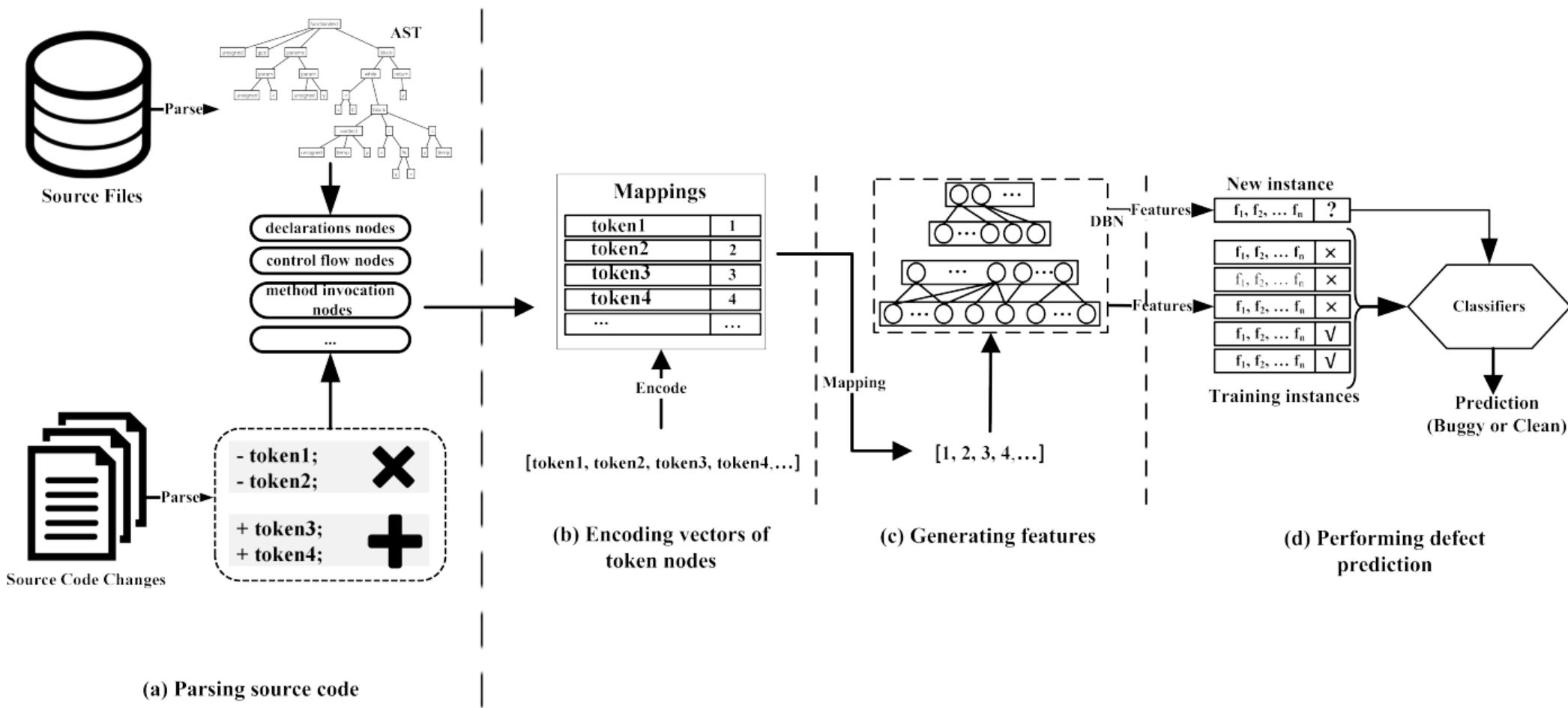
due to the diversity in development processes, a defect prediction model is often not transferable and requires to be rebuilt when the target project changes

# Critiques on defect prediction

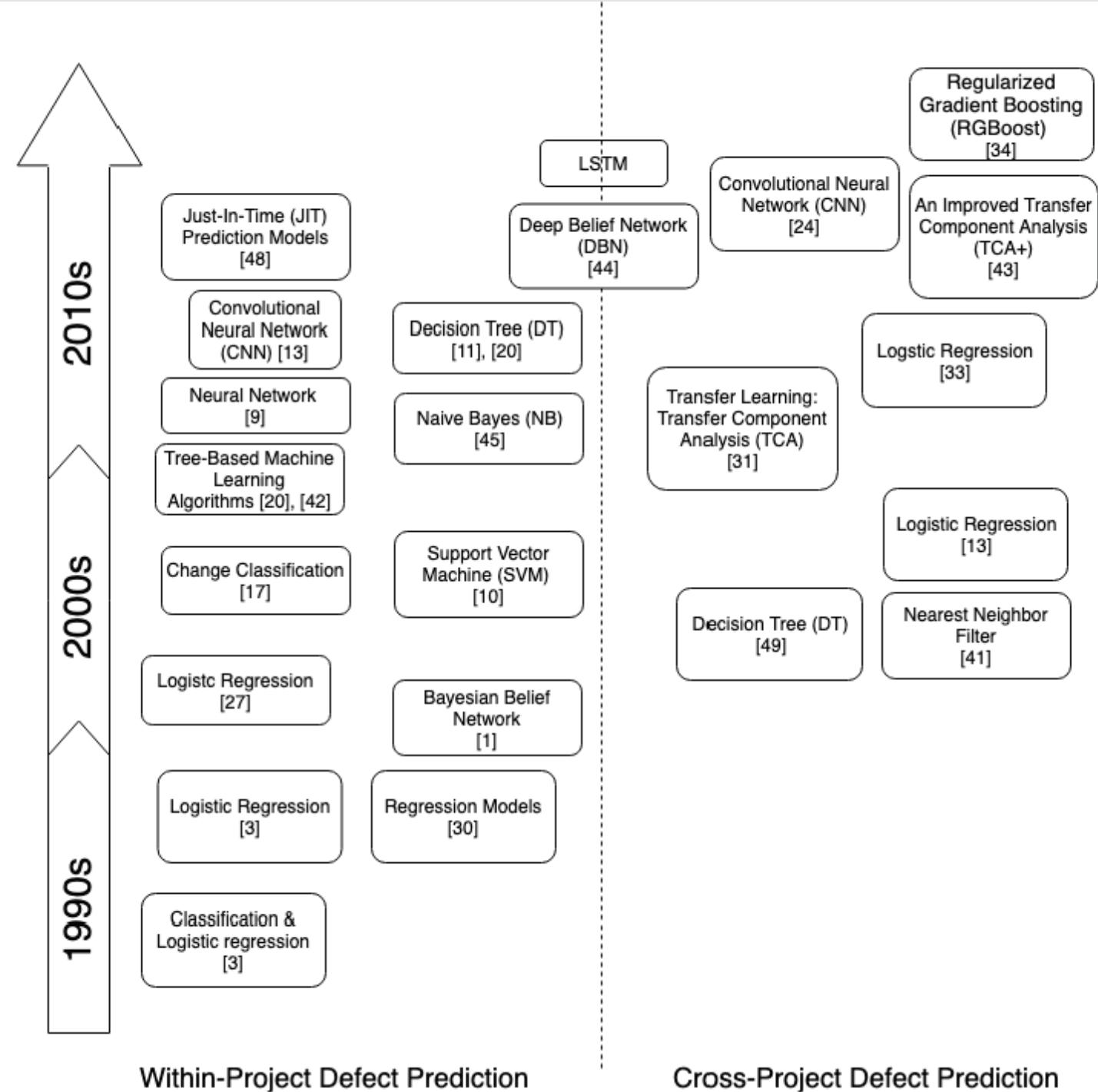
---

- the unknown relationship between defects and failures
- problems with the “multivariate” statistical approach
- problems of using size and complexity metrics as sole “predictors” of defects
- problems in statistical methodology and data quality
- false claims about software decomposition and the “Goldilock’s Conjecture”

# A typical ML model for defect prediction



# History of ML/AI approaches in defect prediction



S. Omri and C. Sinz, "Deep Learning for Software Defect Prediction: A Survey," in *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, New York, NY, USA: Association for Computing Machinery, 2020, pp. 209–214. Accessed: Mar. 19, 2022. [Online]

# Just-in-time defect prediction

## Fixing Commit Message

Revert "Make VisibleRefFilter.Filter reuse the refs passed from JGit."

This reverts commit [b032a529f83892dfbdb375c47a90d89756dd8ab](#). This commit introduced an issue where tags were not replicated under certain circumstances.

Bug: [Issue 2500](#)

Bug: [Issue 1748](#)

Change-Id: [I9c902b99c7f656c7002cf3eab9e525f22a22fb85](#)

## Defective Commit: b032a529f83892dfbdb375c47a90d89756dd8ab

2 gerrit-server/src/main/java/com/google/gerrit/server/git/VisibleRefFilter.java

```
103 103 if (!deferredTags.isEmpty() && (!result.isEmpty() || filterTagsSeparately)) {  
104 104     TagMatcher tags = tagCache.get(projectName).matcher(  
105 105         tagCache,  
106 106         db,  
107 107     -     filterTagsSeparately ? filter(db.getAllRefs()).values() : result.values();  
108 108     +     filterTagsSeparately ? filter(refs).values() : result.values();  
109 109         for (Ref tag : deferredTags) {  
110 110             if (tags.isReachable(tag)) {  
111 111                 result.put(tag.getName(), tag);  
112 112             }  
113 113         }  
114 114     }  
115 115 }
```

## Fixing Commit: 6db280663f836096c30a9626e7170f4a36d8cc1f

2 gerrit-server/src/main/java/com/google/gerrit/server/git/VisibleRefFilter.java

```
112 112 if (!deferredTags.isEmpty() && (!result.isEmpty() || filterTagsSeparately)) {  
113 113     TagMatcher tags = tagCache.get(projectName).matcher(  
114 114         tagCache,  
115 115         db,  
116 116     -     filterTagsSeparately ? filter(refs).values() : result.values();  
117 117     +     filterTagsSeparately ? filter(db.getAllRefs()).values() : result.values();  
118 118         for (Ref tag : deferredTags) {  
119 119             if (tags.isReachable(tag)) {  
120 120                 result.put(tag.getName(), tag);  
121 121             }  
122 122         }  
123 123     }  
124 124 }
```

# Technical Debt Management in Visma

Mili Orucevic  
Chief Software Quality Engineer



nguages

h, Swift,

ossible

la, Dart,

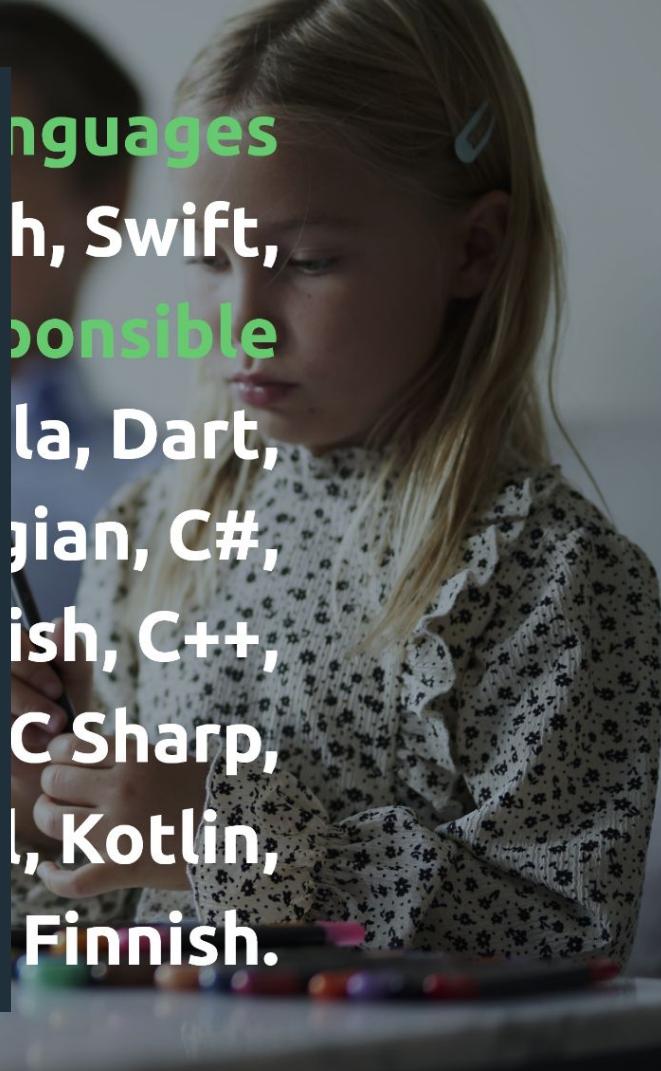
gian, C#,

ish, C++,

C Sharp,

, Kotlin,

Finnish.



## Agenda

- Technical Debt in General
- How are we working with Technical Debt in Visma?
- Dive into some examples
  - Assessment
  - Dashboards
  - Indexes
  - Example of large Technical Debt
- Research: Why Technical Debt Matter?



# Mili Orucevic

## Chief Software Quality Engineer

<https://www.linkedin.com/in/milio>



# Lehman's Law of Software Evolution

## Law of Continuing Change

"A system must be continually adapted or it becomes progressively less satisfactory"

## Law of Increasing Complexity

"As a system evolves, its complexity increases unless work is done to maintain or reduce it."

# Technical Debt definition

"Shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite. Objects make the cost of this transaction tolerable. The **danger occurs when the debt is not repaid**. Every minute spent on **not-quite-right code** counts as **interest on that debt**. Entire engineering organizations can be brought to a stand-still under the debt load of an unconsolidated implementation, object- oriented or otherwise"

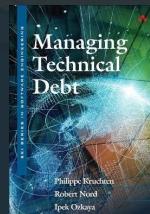
1992 Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)

**Ward Cunningham**

"In software-intensive systems, technical debt consists of design or implementation constructs that are **expedient in the short term** but that set up a technical context that **can make a future change more costly or impossible**. Technical debt is a contingent liability whose impact is limited to internal system qualities—primarily, but not only, maintainability and evolvability."

Kruchten, Nord, Ozkaya (p.5)

## Managing Technical Debt



Software systems are prone to the **build up of cruft** - deficiencies in internal quality that **make it harder than it would ideally be to modify and extend** the system further.

Technical Debt is a metaphor, coined by Ward Cunningham, that frames how to think about dealing with this cruft, thinking of it like a financial debt. The extra effort that it takes to add new features is the interest paid on the debt.

Blogpost, 21.05.2019:  
<https://martinfowler.com/bliki/TechnicalDebt.html>

**Martin Fowler**

# Technical Debt Landscape

Business

New Features  
Additional Functionality

Evolvability

Product Roadmap

↔↔↔ Mostly Invisible →→

## Architecture

Architecture smells  
Pattern Violations  
Structural Complexity

↔↔↔ Visible →→

## Code

Code Complexity  
Code Smells  
Coding Style Violations  
Low Internal Quality

Defects

Low External Quality

## Production Infrastructure

Build, Test and Deploy Issues

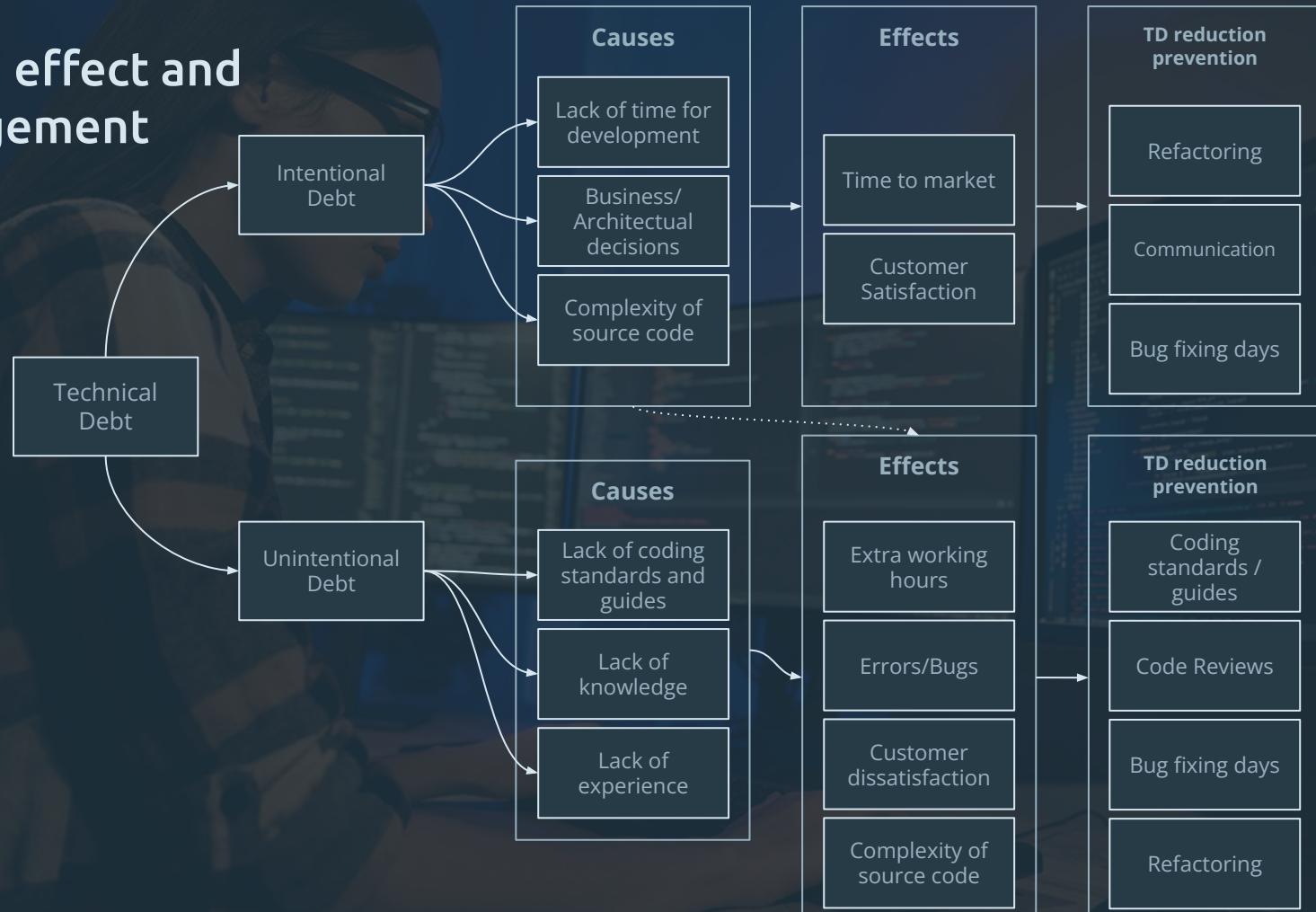
Maintainability

End Product

End Users

Internal Team Work

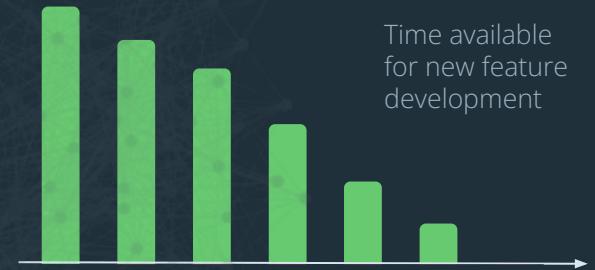
# Cause, effect and management



# Visualizing Technical Debt

Possible symptoms of Technical Debt

- Decline in number of releases
- Increase in delivery lead time
- Less and less time for developing new features
- Might treat symptoms instead of real issues



Time available  
for new feature  
development

No one is here

Won't be here for  
long



# Measuring Technical Debt

## Static Code Analysis

- 18626 days or 51 years of technical debt,  
for 1106 projects
  - ~16 days effort per project
- First 10 days of March, 51 days of effort of  
technical debt was introduced
  - 7 min per project per day

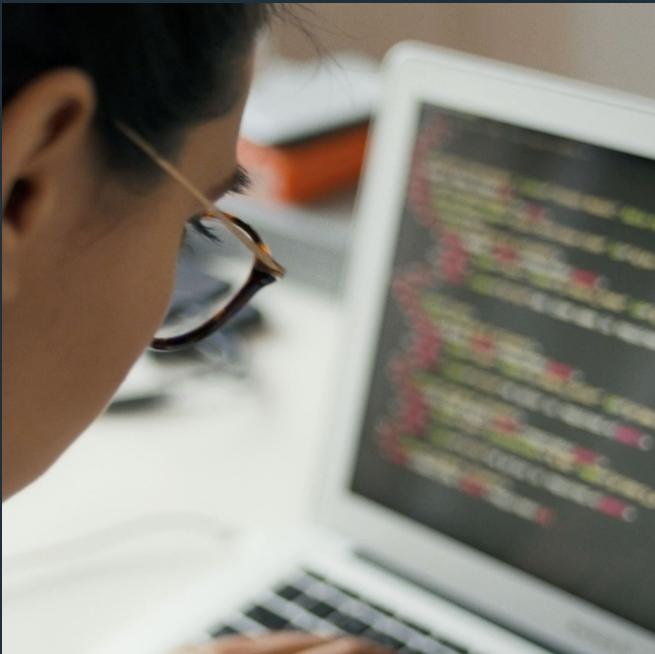
**Should not be the only way to  
measure technical debt**

## Other examples outside Visma

- Project: 15 year old code based
- Technical debt accumulated: 4000 years



How are we creating  
awareness and work with  
**Technical Debt** in Visma?





# Visma - a SaaS company

**14 000**  
Engaged employees

With more than **6 500**  
developers

Since 2014 more than  
**140 companies**  
have joined Visma

**1 135 000**  
Customers

We are **where you are**  
Strong local presence with  
more than **150**  
locations

Running through Visma's  
systems in September '21:

**10,3 million payslips**  
**23,2 million invoices**

# What did we want to solve?

Raise the **awareness** of technical debt in teams

Create **visibility** of technical debt

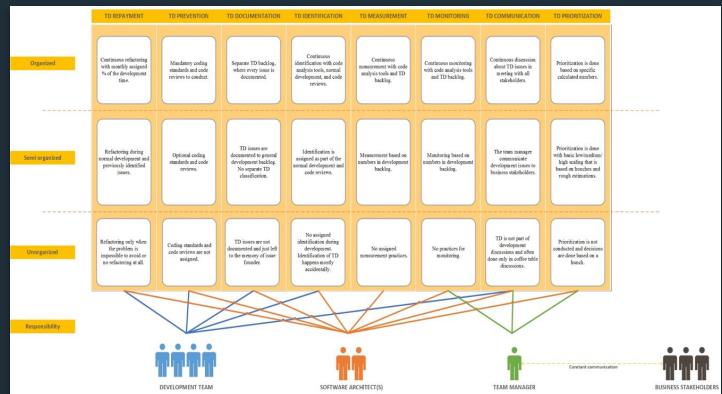
Common **process** for dealing with technical debt

Keep a technical debt **register** for teams

Ensure that technical debt is **planned** and **repaid** where reasonable



# Technical Debt Maturity Assessment



Technical Debt Maturity Model		Prevention	
Primary responsible roles: Architects, developers, and anyone else writing code			
<b>TD Repayment</b>		Unorganized (level 0) <input checked="" type="checkbox"/> The team has not agreed on coding standards and does not perform code review. Semi-organized (level 1) <input type="checkbox"/> Following the team's coding standards and performing code review is optional or not yet. Organized (level 2) <input type="checkbox"/> The team follows coding standards and performs code review as an non-normative change.	
<b>TD Identification</b>		Unorganized (level 0) The team does not identify technical debt during the development process, except when it happens accidentally. Semi-organized (level 1) <input type="checkbox"/> The team identifies technical debt during development and code review. Organized (level 2) <input type="checkbox"/> The team identifies technical debt using static code analysis tools (e.g. Refactor, SonarQube, Coverity). <input type="checkbox"/> The team identifies technical debt as part of the pre-mortem process like incident review and post-mortem management.	
<b>TD Communication</b>		Unorganized (level 0) No communication in technical tool (e.g. Jira, Confluence) with qualitative and quantitative numbers in backlog. Semi-organized (level 1) <input checked="" type="checkbox"/> Product/Service Owner communicates development issues to business stakeholders. Communication must describe customer value. Organized (level 2) <input type="checkbox"/> Prioritization is done with basic low/medium/high scaling that is based on backlog and rough estimations.	
<b>TD Prioritization</b>		Unorganized (level 0) TD is not part of development discussions and often done only in coffee table discussions. Semi-organized (level 1) <input type="checkbox"/> Development team, software architect(s), team manager; product/service owner Organized (level 2) <input type="checkbox"/> Prioritization is not conducted and decisions are done based on a hunch.	
Technical Debt Self-assessment		TD Measurement	
<b>TDM activity / TDM levels</b>		Unorganized (level 0) <input checked="" type="checkbox"/> Continuous refactoring with monthly assigned of the development backlog.	
<b>TD Monitoring</b>		Unorganized (level 0) The team does not monitor technical debt during development and code review. Semi-organized (level 1) <input type="checkbox"/> Technical debt is monitored over time in JIRA by looking at the number of technical debt issues. Organized (level 2) <input type="checkbox"/> Different types of technical debt metrics is measured over time by looking at the sum of all severity scores and metrics in SonarQube.	
<b>TD Analysis</b>		Unorganized (level 0) Technical debt is not analyzed. Semi-organized (level 1) <input type="checkbox"/> Technical debt is analyzed and summarized using the Priority field in JIRA. Organized (level 2) <input type="checkbox"/> Technical debt is analyzed and a severity score is calculated and documented in JIRA.	
<b>TD Planning</b>		Unorganized (level 0) The team rarely talks about technical debt internally (in the team) or externally (with stakeholders). Semi-organized (level 1) <input type="checkbox"/> The team frequently talks about technical debt internally (in the team). Organized (level 2) <input type="checkbox"/> The team frequently communicate technical debt risks to stakeholders. Focusing on the business and/or customer value of paying down the technical debt with the highest priority. The team also communicates technical debt risks to stakeholders. Focusing on the business and/or customer value of paying down the technical debt with the highest priority. The team also communicates technical debt risks to stakeholders. Focusing on the business and/or customer value of paying down the technical debt with the highest priority.	
<b>TD Repayment</b>		Unorganized (level 0) The team only pays down technical debt when it becomes blocking issue. Semi-organized (level 1) <input type="checkbox"/> The team performs sufficient refactoring as part of the normal backlog prioritization and development process. Organized (level 2) <input type="checkbox"/> Some of the team's capacity is leveled meant for paying down technical debt (20% or more) & the industry standard and highly recommended. Supereorganized (level 4) <input type="checkbox"/> The team has instituted a maximum cap on technical debt, and will dedicate 20% of their capacity to paying down their technical debt whenever the agreed cap is exceeded.	

Inspired by research done at the Lappeenranta University of Technology by Jesse Yli-Huumo.

# Technical Debt Maturity Assessment

## Latest version

Category	Requirements
Prevention	<input type="checkbox"/> The team follows coding standards and performs code review for all non-trivial changes <input type="checkbox"/> The intention of implementing any workaround or corner-cutting is discussed within the team before any code is written; team decides together if the workaround will be implemented or more time will be allocated to avoid it > More Info
Identification	<input type="checkbox"/> The team identifies technical debt during implementation and code reviews <input type="checkbox"/> The team identifies technical debt using static/dynamic code analysis tools (e.g. ReSharper, SonarQube, Coverity, Snyk) and adds it in the backlog. > More Info <input type="checkbox"/> The team identifies technical debt as part of operational processes like incident reviews and problem management <input type="checkbox"/> A process to identify outdated dependencies is implemented (e.g Dependabot, dotnet-outdated)
Documentation	<input type="checkbox"/> Technical debt is documented in the same backlog as everything else and the NFR label is used <input type="checkbox"/> The team documents tech debt about to be introduced (intentional) technical debt > More Info <input type="checkbox"/> Our top 3 TD issues are logged and documented in a way that are understandable for stakeholders and decision makers as well. Business impact, if relevant, is part of the documentation. > More Info
Analysis	<input type="checkbox"/> Technical debt is analyzed and a severity score is calculated and documented in JIRA > Why Severity Scores? <input type="checkbox"/> The most important technical debt for our product is reflected in the top 3 TD issues logged in Jira
Monitoring	<input type="checkbox"/> All types of technical debt is visible in Jira by looking at the sum of severity scores. Here are suggestions for TD JIRA Dashboard / TD Dashboard > Click here to expand...
Communication	<input type="checkbox"/> The team continuously communicates technical debt risks to stakeholders and decision makers, focusing on the business and/or customer value of paying down the technical debt with the highest severity score. (Using the ArchTech Maturity Index and a list of technical debt issues in JIRA sorted by the severity score are two ways of doing this.) > Why is it important to communicate and be transparent about technical debt?
Planning	<input type="checkbox"/> The team includes technical debt decisions and planning into their periodic processes. > Intention and More Info <input type="checkbox"/> The team decides on what technical debt should be paid down next based on the severity scores
Repayment	<input type="checkbox"/> Some of the team's capacity is always reserved for paying down technical debt (20% or more is the industry standard and highly recommended)

# Technical Debt Maturity Assessment

Category	Description
<b>Prevention</b>	How is introduction of unintentional technical debt prevented?
<b>Identification</b>	How, and where is technical debt identified?
<b>Documentation</b>	How is technical debt documented and labeled?
<b>Analysis</b>	How is technical debt analyzed with emphasis on risk likelihood, impact and severity?
<b>Monitoring</b>	How is technical debt monitored over time, f.ex. trends, top TD issues
<b>Communication</b>	How is technical debt communicated outside the team, f.ex. towards stakeholders?
<b>Planning</b>	How is technical debt included and prioritized in the improvement plans?
<b>Repayment</b>	How is technical debt repaid?

# Technical Debt Maturity Assessment

Category	Description
<b>Prevention</b>	How is introduction of unintentional technical debt prevented?
<b>Identification</b>	How, and where is technical debt identified?
<b>Documentation</b>	How is technical debt documented and labeled?
<b>Analysis</b>	How is technical debt analyzed with emphasis on risk likelihood, impact and severity?
<b>Monitoring</b>	How is technical debt monitored over time, f.ex. trends, top TD issues
<b>Communication</b>	How is technical debt communicated outside the team, f.ex. towards stakeholders?
<b>Planning</b>	How is technical debt included and prioritized in the improvement plans?
<b>Repayment</b>	How is technical debt repaid?

# Technical Debt Maturity Assessment

Category	Description
<b>Prevention</b>	How is introduction of unintentional technical debt prevented?
<b>Identification</b>	How, and where is technical debt identified?
<b>Documentation</b>	How is technical debt documented and labeled?
<b>Analysis</b>	How is technical debt analyzed with emphasis on risk likelihood, impact and severity?
<b>Monitoring</b>	How is technical debt monitored over time, f.ex. trends, top TD issues
<b>Communication</b>	How is technical debt communicated outside the team, f.ex. towards stakeholders?
<b>Planning</b>	How is technical debt included and prioritized in the improvement plans?
<b>Repayment</b>	How is technical debt repaid?

# Technical Debt Maturity Assessment

Category	Description
<b>Prevention</b>	How is introduction of unintentional technical debt prevented?
<b>Identification</b>	How, and where is technical debt identified?
<b>Documentation</b>	How is technical debt documented and labeled?
<b>Analysis</b>	How is technical debt analyzed with emphasis on risk likelihood, impact and severity?
<b>Monitoring</b>	How is technical debt monitored over time, f.ex. trends, top TD issues
<b>Communication</b>	How is technical debt communicated outside the team, f.ex. towards stakeholders?
<b>Planning</b>	How is technical debt included and prioritized in the improvement plans?
<b>Repayment</b>	How is technical debt repaid?

# Technical Debt Maturity Assessment

Category	Description
<b>Prevention</b>	How is introduction of unintentional technical debt prevented?
<b>Identification</b>	How, and where is technical debt identified?
<b>Documentation</b>	How is technical debt documented and labeled?
<b>Analysis</b>	How is technical debt analyzed with emphasis on risk likelihood, impact and severity?
<b>Monitoring</b>	How is technical debt monitored over time, f.ex. trends, top TD issues
<b>Communication</b>	How is technical debt communicated outside the team, f.ex. towards stakeholders?
<b>Planning</b>	How is technical debt included and prioritized in the improvement plans?
<b>Repayment</b>	How is technical debt repaid?

# Technical Debt Maturity Assessment

Category	Description
<b>Prevention</b>	How is introduction of unintentional technical debt prevented?
<b>Identification</b>	How, and where is technical debt identified?
<b>Documentation</b>	How is technical debt documented and labeled?
<b>Analysis</b>	How is technical debt analyzed with emphasis on risk likelihood, impact and severity?
<b>Monitoring</b>	How is technical debt monitored over time, f.ex. trends, top TD issues
<b>Communication</b>	How is technical debt communicated outside the team, f.ex. towards stakeholders?
<b>Planning</b>	How is technical debt included and prioritized in the improvement plans?
<b>Repayment</b>	How is technical debt repaid?

# Technical Debt Maturity Assessment

Category	Description
<b>Prevention</b>	How is introduction of unintentional technical debt prevented?
<b>Identification</b>	How, and where is technical debt identified?
<b>Documentation</b>	How is technical debt documented and labeled?
<b>Analysis</b>	How is technical debt analyzed with emphasis on risk likelihood, impact and severity?
<b>Monitoring</b>	How is technical debt monitored over time, f.ex. trends, top TD issues
<b>Communication</b>	How is technical debt communicated outside the team, f.ex. towards stakeholders?
<b>Planning</b>	How is technical debt included and prioritized in the improvement plans?
<b>Repayment</b>	How is technical debt repaid?

# Technical Debt Maturity Assessment

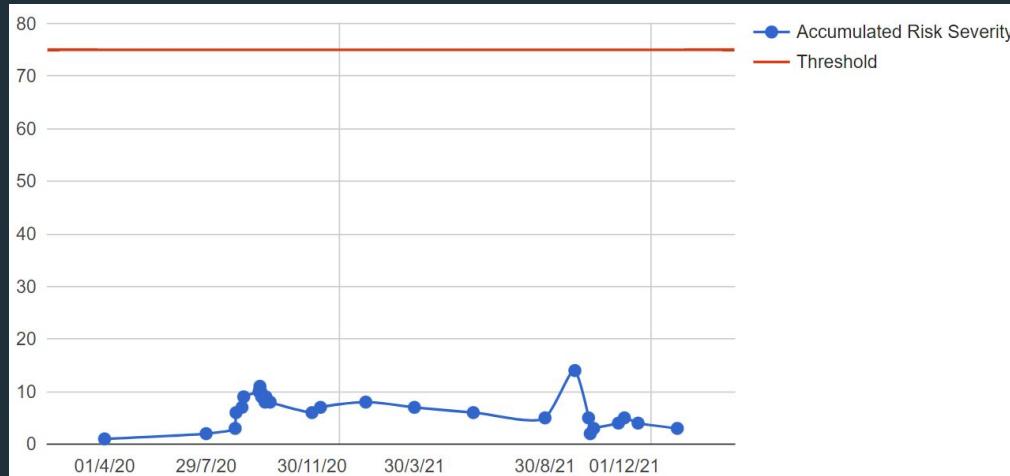
Category	Description
<b>Prevention</b>	How is introduction of unintentional technical debt prevented?
<b>Identification</b>	How, and where is technical debt identified?
<b>Documentation</b>	How is technical debt documented and labeled?
<b>Analysis</b>	How is technical debt analyzed with emphasis on risk likelihood, impact and severity?
<b>Monitoring</b>	How is technical debt monitored over time, f.ex. trends, top TD issues
<b>Communication</b>	How is technical debt communicated outside the team, f.ex. towards stakeholders?
<b>Planning</b>	How is technical debt included and prioritized in the improvement plans?
<b>Repayment</b>	How is technical debt repaid?

# Example assessment

[Link to assessment](#)



# Dashboard - team A

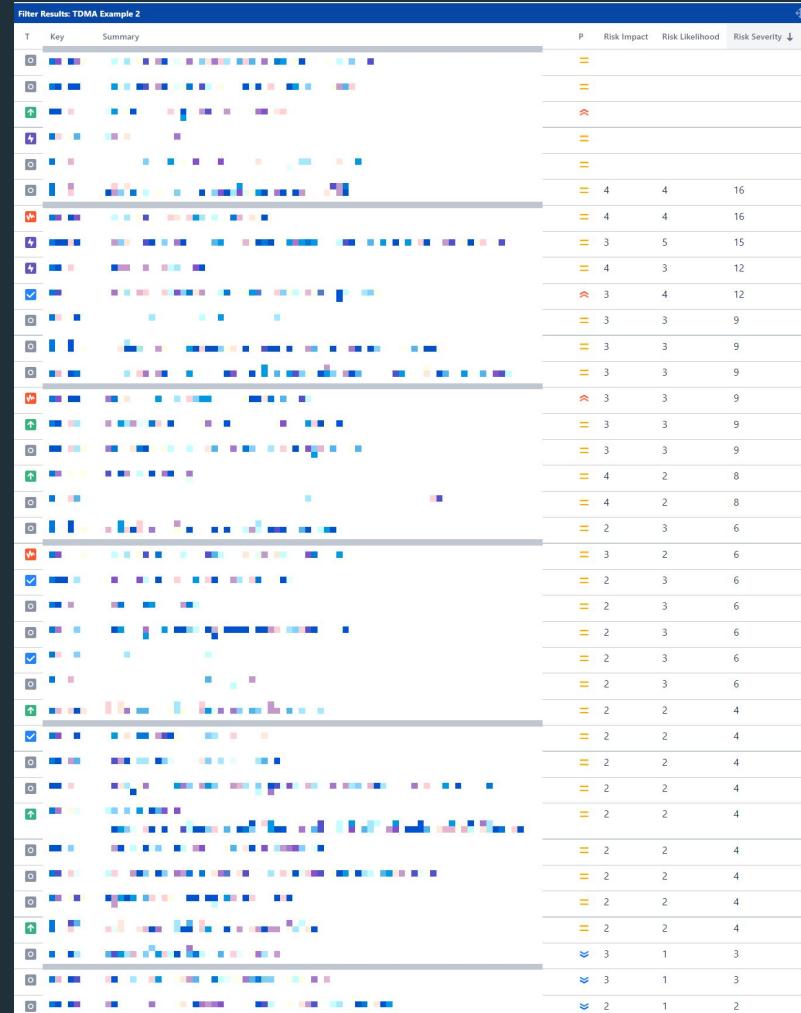


Filter Results: TDMA Example

T	P	Risk Impact	Risk Likelihood	Risk Severity
✓	=	3	3	9
✓	=	3	1	3
○	=	1	1	1
○	=	1	1	1
○	=	1	1	1
✓	=	1	1	1
○	▼	1	1	1
○	=	1	1	1
✓	=	1	1	1
✓	=	1	1	1
↑	=	1	1	1
✓	=	1	1	1

1-12 of 12

# Dashboard - team B



# Examples from SonarQube

The screenshot displays the SonarQube interface with three main sections: Quality Gate Status, Measures, and Conditions.

**Quality Gate Status:**

- Passed:** All conditions passed.
- Failed:** 1 condition failed. On New Code: 1 New Critical Issues is greater than 0.
- Failed:** 1 conditions failed. On New Code: 11 Bugs, 4 Vulnerabilities, 0 Security Hotspots.

**MEASURES:**

- New Code: Since November 10, 2020
- Overall Code

**Conditions:**

**Conditions on New Code**

Conditions on New Code apply to all branches and to Pull Requests.

Metric	Operator	Value
Coverage	is less than	60.0%
Duplicated Lines (%)	is greater than	3.0%

**Conditions on Overall Code**

Conditions on Overall Code apply to branches only.

Metric	Operator	Value
Coverage	is less than	80.0%
Duplicated Lines (%)	is greater than	2.0%
Maintainability Rating	is worse than	A
Reliability Rating	is worse than	A
Security Rating	is worse than	A
Unit Test Errors	is greater than	0
Unit Test Failures	is greater than	0

**Measures Summary:**

- Debt: 13d
- Code Smells: 340
- Maintainability: A
- Coverage: 77.7% (Coverage on 17k Lines to cover)
- Unit Tests: 994
- Duplications: 2.0% (Duplications on 41k Lines)
- Duplicated Blocks: 53

# Examples from SonarQube

The screenshot displays the SonarQube interface for a C# project. On the left, a sidebar lists several code smells:

- 'System.Exception' should not be thrown by user code. (Code Smell)
- 'System.Exception' should not be thrown by user code. (Code Smell)
- Refactor this code to not nest more than 3 control flow statements. (Code Smell)
- 'System.Exception' should not be thrown by user code. (Code Smell)
- Remove the unused local variable 'request'. (Code Smell)
- Fix this implementation of 'IDisposable' to conform to the dispose pattern. (Code Smell +2)

A message at the bottom of the sidebar states: "Unused local variables should be removed".

On the right, the code editor shows a snippet of C# code with annotations:

```
67  
68  
69  
70  
71  
72  
73  
    }  
  
    private async Task<[REDACTED]> [REDACTED]([REDACTED])  
    {  
        [REDACTED]  
        var request = new GetSecretValueRequest { SecretId = secretName };  
  
        Remove the unused local variable 'request'. Why is this an issue?  
        last year ▾ L73 9%  
        Code Smell Major Open [REDACTED] 5min effort Comment  
        unused ▾  
  
    }  
74  
75  
76  
77  
78  
79  
80  
81  
82  
83  
84  
85  
86  
87  
88  
89  
90  
91  
92
```

The code editor has a color-coded navigation bar below the code lines, with colors corresponding to different file types or regions.

At the bottom of the page, there is a "Noncompliant Code Example" section containing the following C# code:

```
public int NumberOfMinutes(int hours)  
{  
    int seconds = 0; // seconds is never used  
    return hours * 60;  
}
```

Below this example, two footer messages are displayed:

- General exceptions should never be thrown
- Utility classes should not

In the bottom right corner, the VISMA logo is visible.

# “Not our fault”

## Example of a bigger technical debt

- Unintentional
- External factors
- Time

“Move from AngularJS to new version of Angular”

~ 1583 hours = 66 days

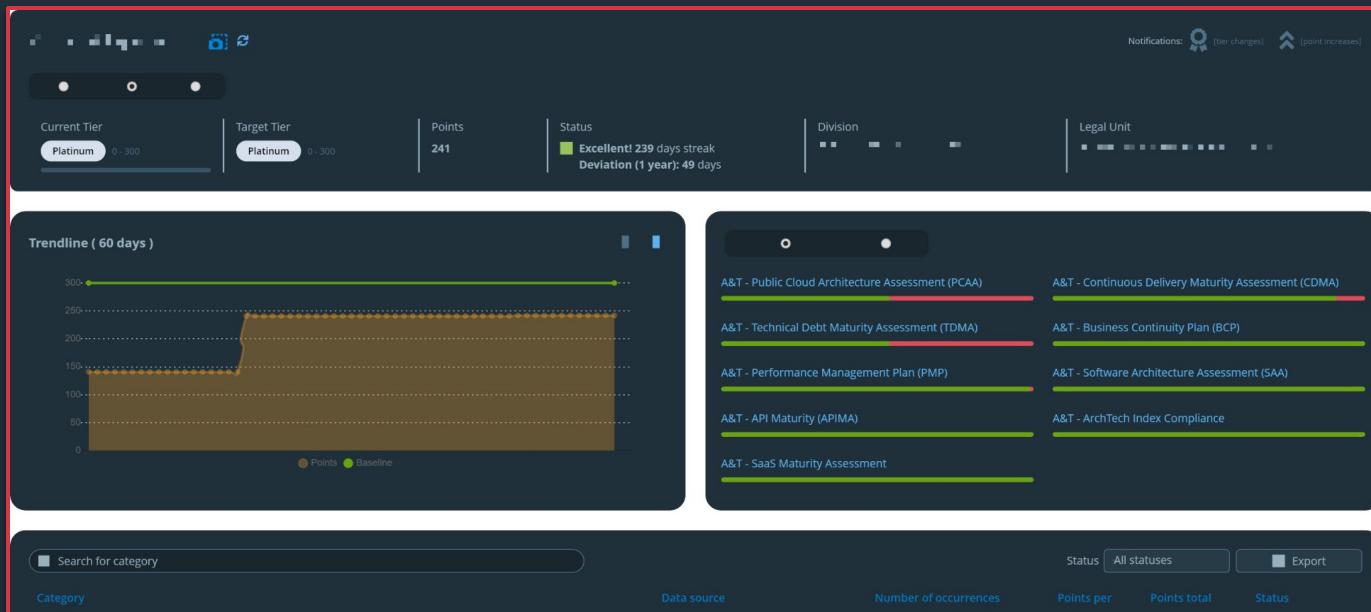
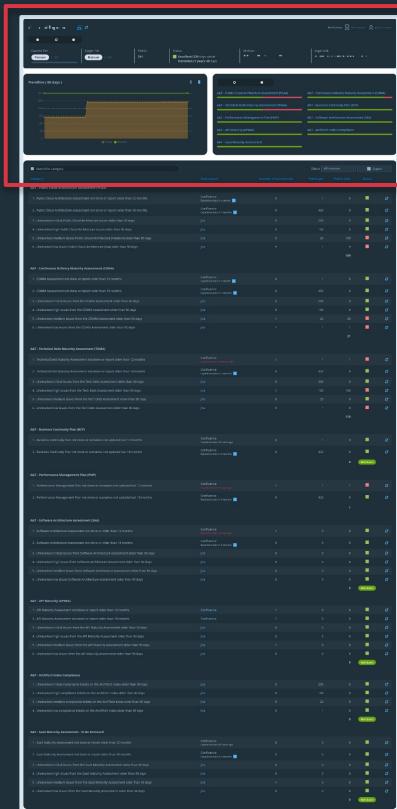


# Automatically track deviations

Visma Index



# Visma Index



# Visma Index



# Why Technical Debt Matter?

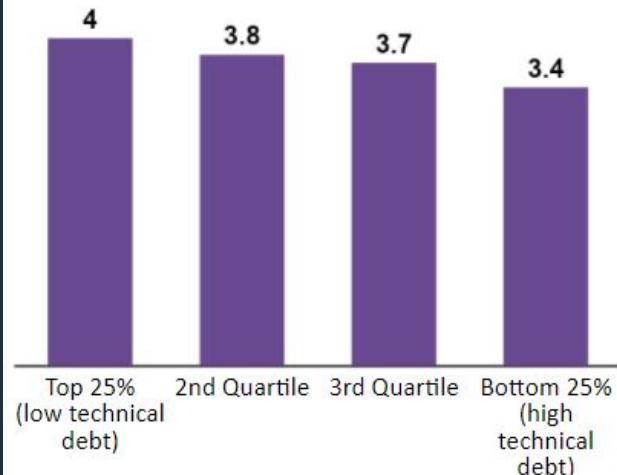
Lessons learned from own research



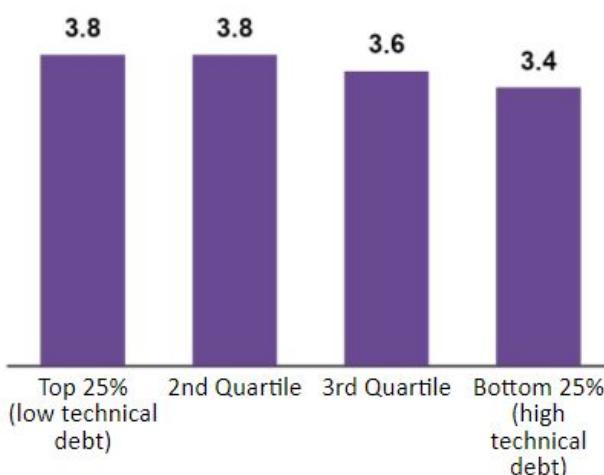
```
140         target="_blank"
141         rel="noopener noreferrer"
142         href={trackUrl}
143       >
144       Instagram
145     </a>
146   </li>
147 </ul>
148 </div>
149   );
150 }
151
152 renderWhatShowLinks() {
153   return (
154     <div className={styles.showLinks}>
155       <h4 className={styles.showLinksTitle}>
156         {this.renderWhatShowLinksTitle()}
157       </h4>
158       {this.renderWhatShowLinksList()}
159     </div>
160   );
161 }
162
163 renderWhatShowItem(title, url) {
164   return (
165     <li className={styles.footer}>
166       <a href={trackUrl(url)}
167         target="_blank"
168         rel="noopener noreferrer"
169       >
170         {title}
171       </a>
172     </li>
173   );
174 }
175
176 renderFooterSub() {
177   return (
178     <div className={styles.footerSub}>
179       <a href="/" title="Home - Unsplash">
180         <img
181           type="logo"
182           className={styles.footerSubLogo}
183         />
184       </a>
185       <span className={styles.footerSlogan}>
186         <img alt="Unsplash logo" />
187       </span>
188     </div>
189   );
190 }
191
192 render() {
193   return (
194     <footer className={styles.footerGlobal}>
195       <div className="container">
196         {this.renderFooterMain()}
197         {this.renderFooterSub()}
198       </div>
199     </footer>
200   );
201 }
```

# Technical debt matters!

Customer-centricity vs. Technical Debt (quartiles)



Innovation vs. Technical Debt (quartiles)

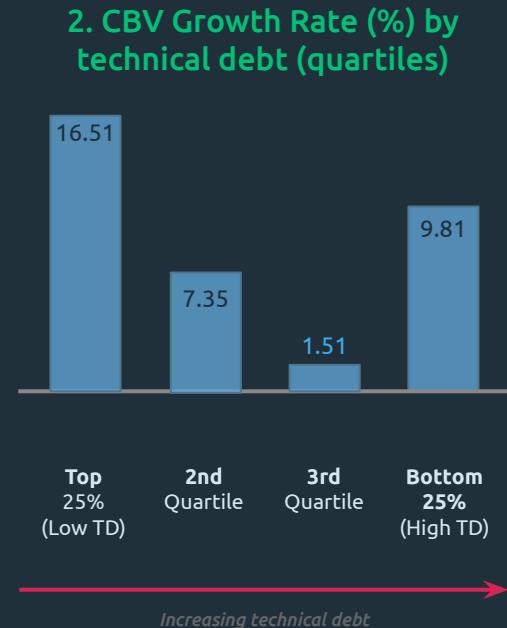
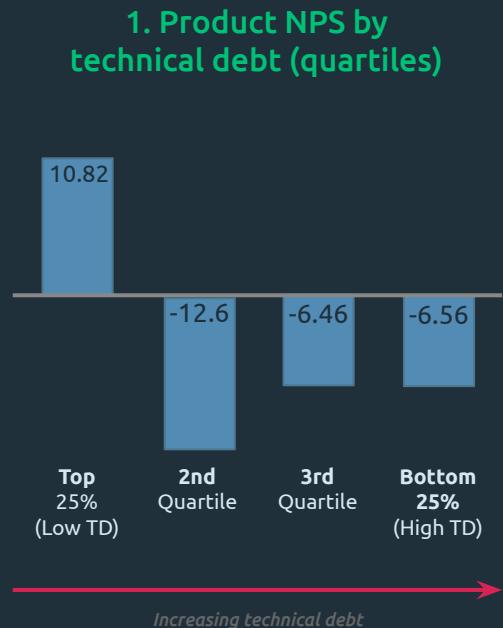


Products with **more technical debt** are developed by teams with **lower customer-centricity and innovation scores**.

Presumably, it's difficult to have time and energy for being innovative and customer-centric when you have work with and around old technology and practices.

# Technical debt matters!

The reduced innovation and customer-centricity in products with high technical debt drives down customer satisfaction and revenue growth.



1. In general, products with less technical debt are more successful
  - Higher product NPS
  - Higher product growth rate

2. Product lines in the bottom 25% of technical debt have higher growth rate than expected. These product lines tend to have lower CBV, and could be smaller products that have accumulated technical debt through rapid growth.



Entrepreneurial  
Responsible  
Dedicated  
Inclusive

---

Make progress happen



# **TDT 4242 Advanced Software Engineering**

**- Spring 2022**

# About Jingyue Li (Bill)

- Jingyue Li (Bill)
  - Master (Computer science) in China
  - Architect: IBM China Ltd.
    - Banking solutions
  - PhD and Post-Doc (Software engineering) at IDI, NTNU
  - Principal researcher: DNV Research & Innovation
    - Software engineering
    - Dependability of critical software systems
  - Teaching experience:
    - Software architecture, IT projects, software security and data privacy, advanced software engineering, empirical software engineering
    - Started teaching this course from 2017
  - [Jingyue.li@ntnu.no](mailto:Jingyue.li@ntnu.no)



# About Daniela

- Co-Responsible teacher: Daniela Cruzes
  - Master (Computer science) in Brazil
  - Dr. Ing (Software engineering) in Brazil
  - Post-Doc in USA and at IDI
  - Tester: CPqD.
  - Billing solutions for Telecom.
  - Software Security Officer: FARA.
  - Transport Intelligent Systems
  - Senior Research Scientist: SINTEF
  - Professor of Software Engineering:
    - NTNU
  - Lead Security Researcher : VISMA
- 
- [Email: dcruzes@ntnu.no](mailto:dcruzes@ntnu.no)



# About Anh Nguyen-Duc



- Occupation & Experience
  - Assoc. Prof. at NTNU, Assoc. Prof. At USN
  - Entrepreneurs (Muml, VVN, etc), Startup mentors (VVNOR, Klikkit, Mitanbud, Comartek AS, etc)
- Research interest:
  - Software startups & software ecosystem
  - Secure software development, Static analysis
  - AI for SE, SE for AI, AI ethics
- Education
  - Master (Software Engineering) in Sweden & Germany
  - PhD and Post-Doc (Software engineering) at IDI, NTNU
- Teaching experience:
  - Customer driven project (TDT4290), Software engineering (TDT4140), IT project (IT2901), Web development & HCI (WEB1000), Software process (SYS1000), Project management (PRO1000, MIS405), Green IT (MSM4103), Product Process and People (MS4102)
- Contact: [anhn@ntnu.no](mailto:anhn@ntnu.no) or [angu@usn.no](mailto:angu@usn.no)

# Outline

- Knowledge to learn to be a good software engineer
  - What advanced SE knowledge to be covered in this course?
  - How is this course organized?

# Goals of the software methodology – Ensure value creation

- Ensure economic gain
- Ensure personal gain
- Ensure organizational (business) gain
- Ensure societal gain
- Phases of value achievement
  - Plan for value achievement. Business case and value realization plan
  - Perform change
  - Implement change and realize the value

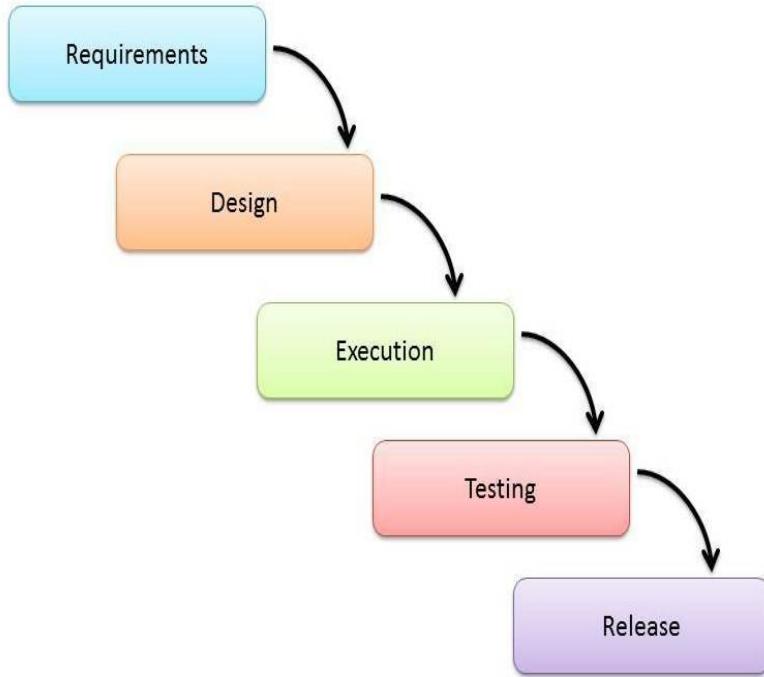
# Categories of software based on functions

- Application software (web, mobile, IoT, control SW)
- System software (OS, DB, ...)
- Computer programming tools (compiler, MATLAB, etc.)
- ...

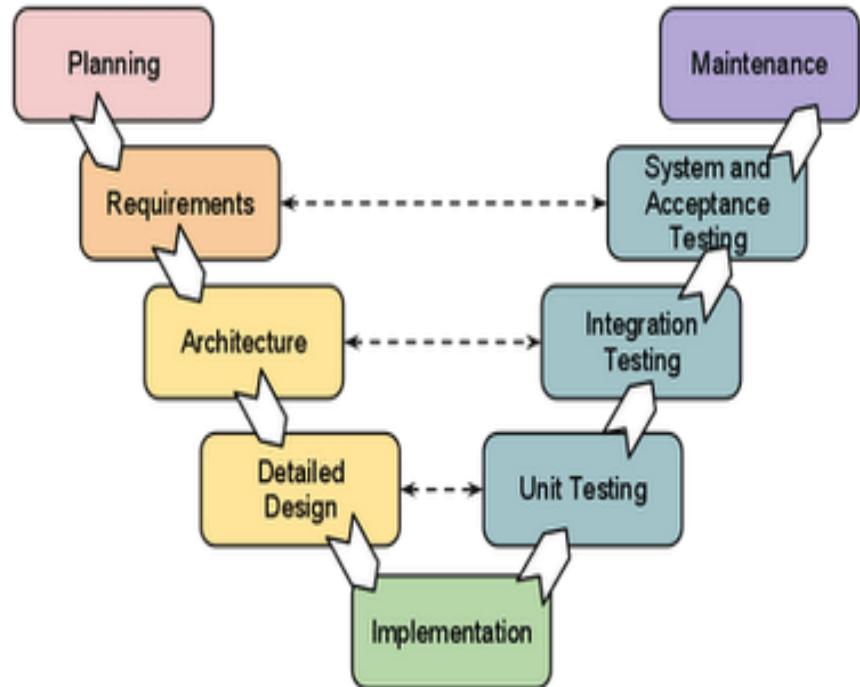
# Project types

- In-house development:
  - The same company makes requirements, develops, and uses the system
- Custom development:
  - One company is customer, and
    - Another company analyzes requirements and develops system
    - Another company analyzes requirements, yet another develops
  - May include outsourcing to subcontractors
- Mass-market software
  - Project to buy or rent package (SaaS)
  - Project to develop an adaptation of package (e.g., ERP-system)
- Reuse of resources in digital or software ecosystems

# Typical software development lifecycle models\*



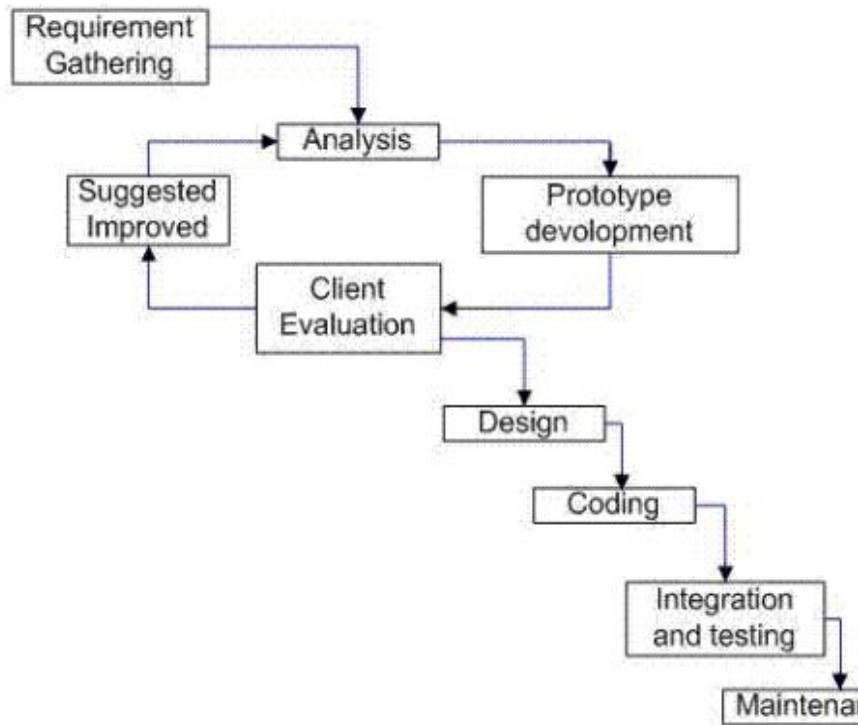
Waterfall model



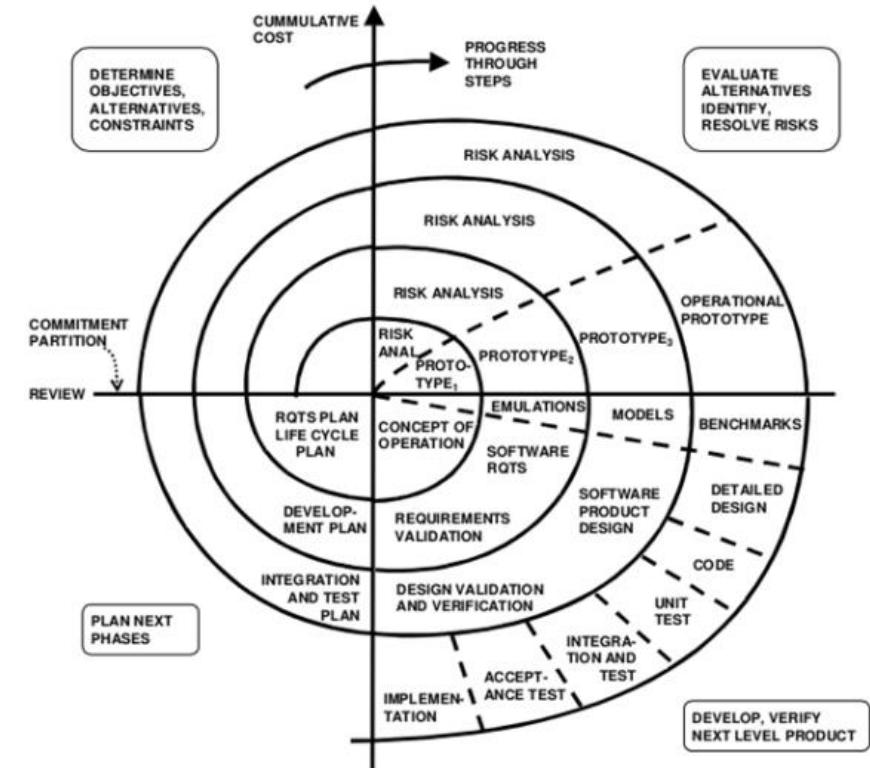
V-model

\* <https://melsatar.blog/2012/03/15/software-development-life-cycle-models-and-methodologies/>

# Typical software development lifecycle models (cont')\*



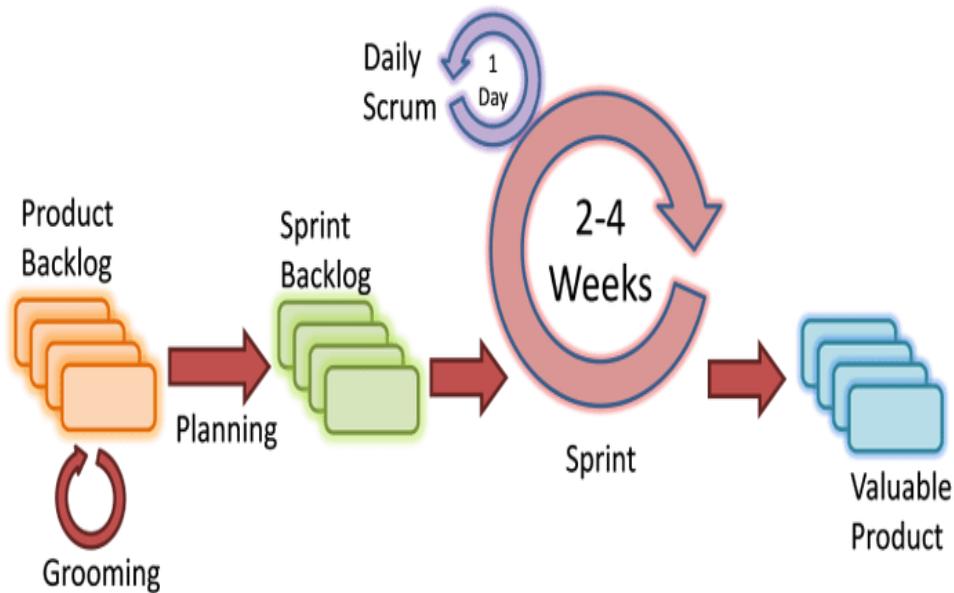
Evolutionary Prototyping Model



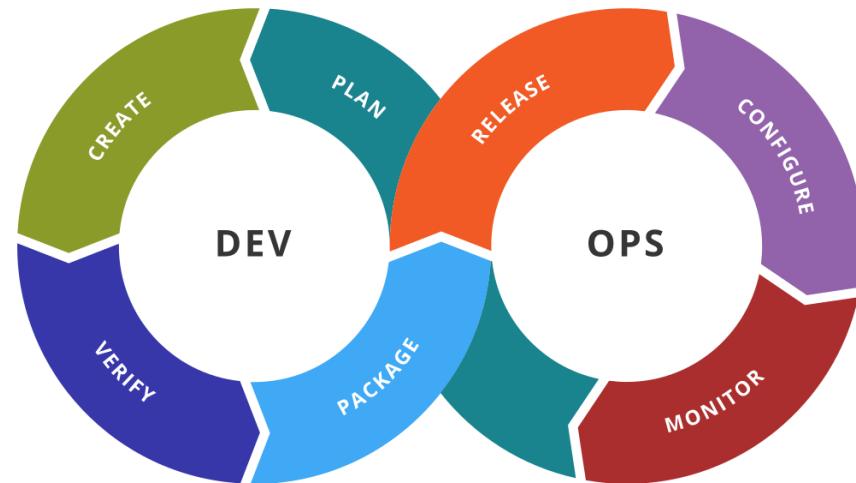
Spiral Model

\* <https://melsatar.blog/2012/03/15/software-development-life-cycle-models-and-methodologies/>

# Typical software development lifecycle models (cont’’)\*



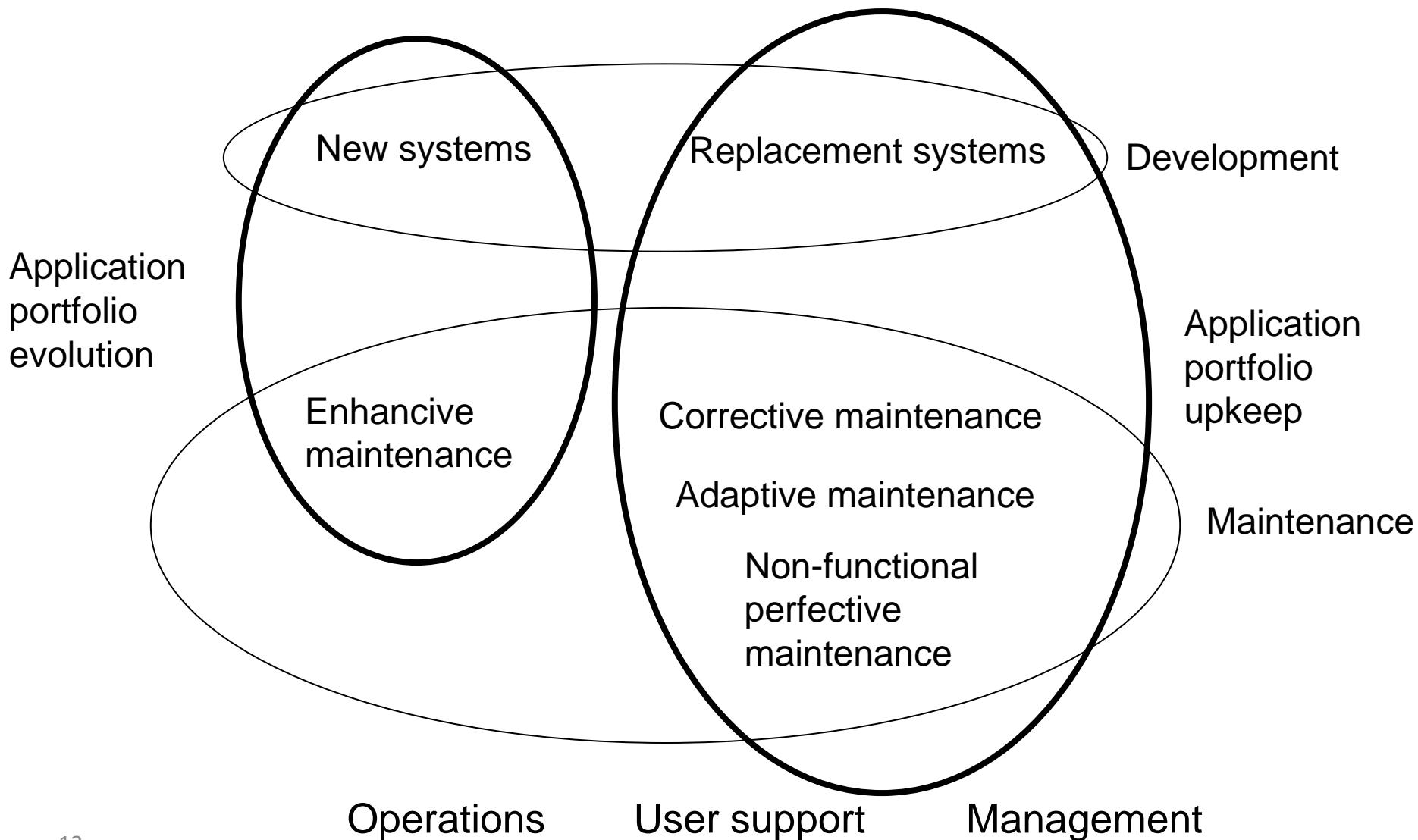
Agile Model



DevOps

\* <https://melsatar.blog/2012/03/15/software-development-life-cycle-models-and-methodologies/>

# Tasks within development and maintenance of software systems



# **Software Engineering Body of Knowledge (SWEBOK)\* covered**

- Software requirements
- Software design
- Software construction
- Software testing
- Software maintenance
- Software configuration management
- Software engineering management
- Software engineering process

\*<https://www.computer.org/web/swebok>

# Software Engineering Body of Knowledge (SWEBOK) covered (Cont')

- Software engineering models and methods
- Software quality
- Software engineering professional practice
- Software engineering economics
- Computing foundations
- Mathematical foundations
- Engineering foundations

# Related courses

- TDT4140 Software Engineering
- TDT4250 Advanced Software Design
- TDT4290 Customer-driven project

# Outline

- Knowledge to learn to be a good software engineer
  - What advanced SE knowledge to be covered in this course?
  - How is this course organized?

# High-level course modules

- High-quality requirements
- High-quality and effective testing
- High-quality and maintainable code
- Advanced topics
  - DevOps
  - Cost and defect estimation
  - Software startup
  - Software eco-systems

# Tentative lecture plan

## High-quality requirements

Week	Date	Lecture content	Speaker
3	17.01.22	Requirement engineering 1	Anh Nguyen Duc
4	24.01.22	Requirement engineering 2	Anh Nguyen Duc

# Challenges in Requirements Engineering

- Importance of getting requirements right:  
1/3 budget to correct errors originates from requirements

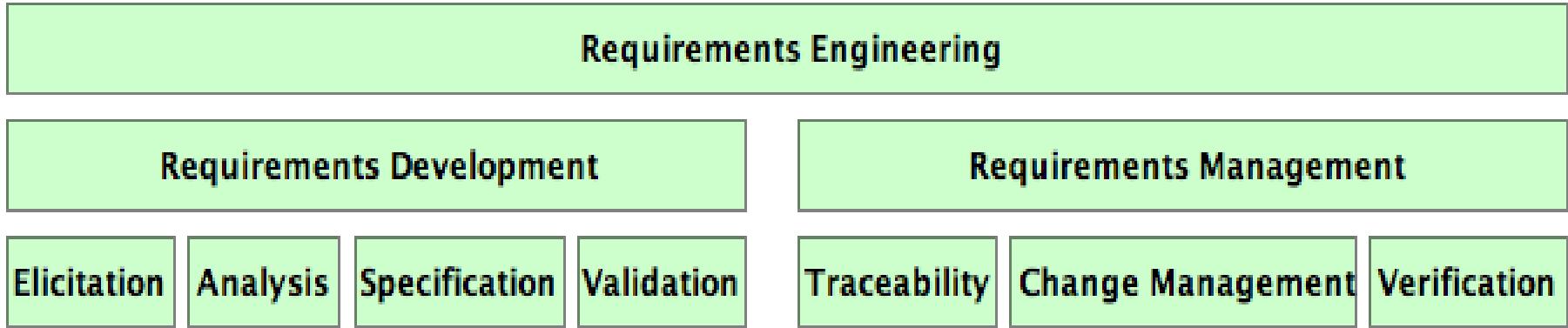
Phase in which fixed	Relative Cost
Requirements	1
Design	3 – 6
Coding	10
Development Testing	15 – 40
Acceptance Testing	30 – 70
Operations	40 – 1000

# Challenges in Requirements Engineering (Cont')

- Why projects are canceled:

Factors	Percentage of Responses
Incomplete Requirements	13.1%
Lack of User Involvement	12.4%
Lack of Resources	10.6%
Unrealistic Expectations	9.9%
Lack of Executive Support	9.3%
Changing Requirements	8.7%
Lack of Planning	8.1%
Didn't need it any longer	7.5%
Lack of IT Management	4.3%
Technology Illiteracy	9.9%

# Requirements Engineering process



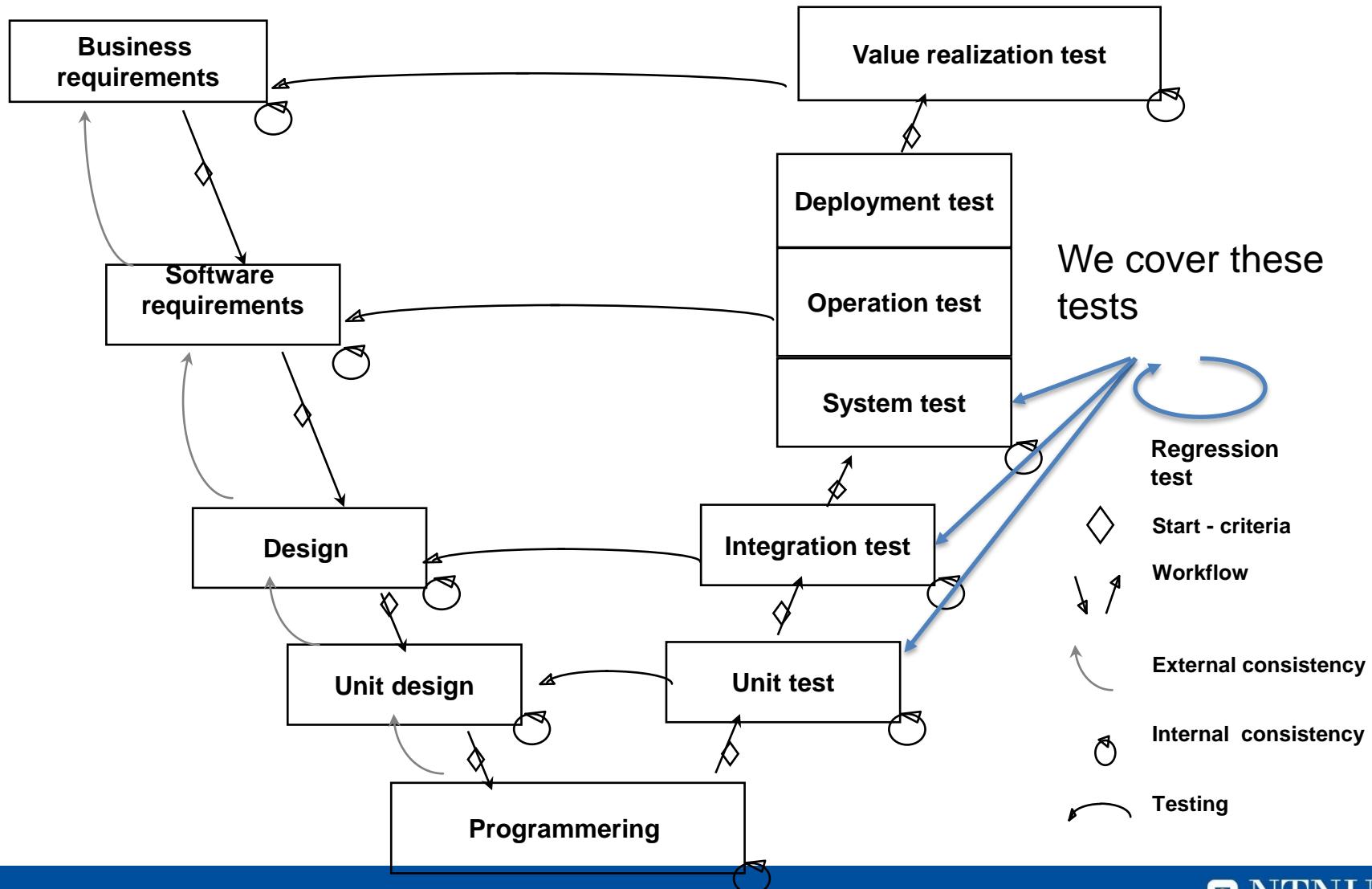
- Use cases
- User stories
- Goal-oriented RE approaches

# Characteristics of good software requirements specification\*

- Complete
- Unambiguous
- Consistent
- Correct
- Verifiable
- Traceable
- Ranked for importance and/or stability
- Modifiable

\* IEEE Std 830-1998

# Requirements matched by testing – the V-model



# IEC 61508 – Test requirements

Technique/Measure *		Ref.	SIL 1	SIL 2	SIL 3	SIL 4
1	Probabilistic testing	C.5.1	---	R	R	R
2	Dynamic analysis and testing	B.6.5 Table B.2	R	HR	HR	HR
3	Data recording and analysis	C.5.2	HR	HR	HR	HR
4	Functional and black box testing	B.5.1 B.5.2 Table B.3	HR	HR	HR	HR
5	Performance testing	Table B.6	R	R	HR	HR
6	Model based testing	C.5.27	R	R	HR	HR
7	Interface testing	C.5.3	R	R	HR	HR
8	Test management and automation tools	C.4.7	R	HR	HR	HR
9	Forward traceability between the software design specification and the module and integration test specifications	C.2.11	R	R	HR	HR
10	<b>Formal verification</b>	<b>C.5.12</b>	---	---	R	R

# Tentative lecture plan

## High-quality and effective testing

Week	Date	Lecture content	Speaker
6	07.02.22	Control flow and data flow testing	Jingyue Li
7	14.02.22	Functional testing with Black Box	Jingyue Li
8	21.02.22	System and acceptance testing and Regression testing	Jingyue Li
9	28.02.22	Testing in Practice	Daniela Cruzes

# Tentative lecture plan

## High-quality and maintainable code

Week	Date	Lecture content	Speaker
10	07.03.22	Code review and code refactoring	Jingyue Li
11	14.03.22	Technical Debt process in Industry	Guest lecture from Visma

# Code review, analysis, and refactoring



## Refactoring Code



```
if (NotificationClient == null)
{
    NotificationClient = new b1.desktop.NotificationClient();
    //NotificationClient.Insert();
}
else
{
    NotificationClient.LastRequest = DateTime.Now;
    NotificationClient.RequestCount = NotificationClient.RequestCount + 1;
    //NotificationClient.Update();
}

if (NotificationClient.Deny == false)
{
    NotificationRequest NotificationRequest = new b1.desktop.NotificationRequest();
    NotificationRequest.ClientId = ClientId;
    NotificationRequest.Request.UserId = UserHolder;
    NotificationRequest.
```

- Making code better, may not be faster
- Better structured, better built
- More readable, more understandable
- Easy to work with
- Easy to add new features
- Easy to spot and fix bugs
- Keeping code in control
- Improving existing code

# Tentative lecture plan

## Advanced topics

Week	Date	Lecture content	Speaker
5	31.01.22	DevOps and Git Lab	Nora Thomas from Vipps.no
12	21.03.22	Cost and defect estimation	Anh Nguyen Duc
13	28.03.22	Software startup Software eco-systems	Anh Nguyen Duc

# DevOps

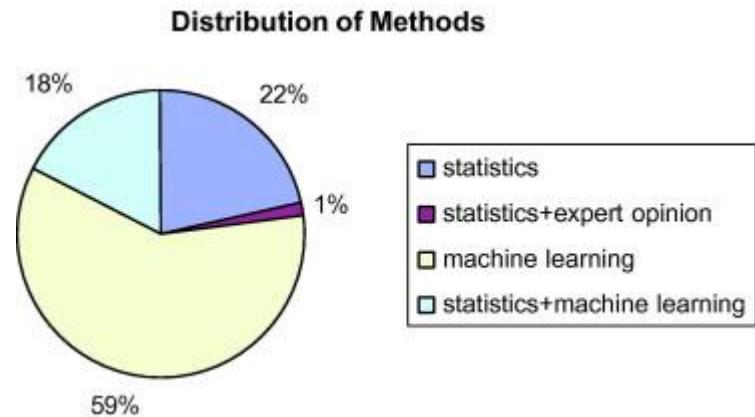
- DevOps is the practice of operations and development engineers participating together in the entire service lifecycle, from design through the development process to production support.
- Integrated development and operations
- Supporting quick releases

# Cost and defect estimation - Cost

- Why?
  - 55% total number of project runs out of budgets
  - Almost every software project experiences some sort of project delays
  - Estimation is needed for better planning and preventive actions
- Purpose of software estimation
- How
  - Expert judgement: Delphi method, focus group, planning poker, etc
  - Regression-based method: COCOMO models, Putnam model, and function points
  - Simulation-based
  - Machine learning based approach: Bayesian Network, Deep Learning, etc
- Where to apply:
  - Waterfall vs. Agile
  - Within-company vs. cross-company
  - Private vs open source dataset

# Cost and defect estimation - Defect

- Defects vs. fault, error, breach, failure
- Software defect can be very costly!
- Defect prediction is a cost-effective to deal with defect
  - Informed static analysis
  - Bug triaging
  - Faulty vs. non-faulty classification
- Approaches
- Metrics:
  - Project-level metrics: project type, budget, duration
  - Team level metrics: engineer's competence
  - Code level metrics: LOC, CK metric suite, code smell, number of functions, number of interfaces, number of class, code complexity, etc
  - Product level metrics: number of component, cohesion, etc
  - Collaboration level metrics: people-to-people interaction, people-to-file interaction
- Advanced defect prediction methods
  - Regression-based approaches
  - Support Vector Machine
  - Machine Learning (supervised/ unsupervised)
  - Decision Trees
  - Naïve Bayes
  - Deep Learning



# Software startup

- Airbnb, Uber, Spotify, Skype, Kahoot, AppSumo, Vipps, Mitandbud, etc are used to be startups
- Startup companies are unique:
  - Little or no operating history
  - Limited resources
  - Multiple influences
  - Dynamic technologies and markets
- A temporary organizational state with special way of working, engineering practices and activities can be done different than in established contexts

# Software startup

- Understanding failure in startups
- Reasoning about software startup engineering
  - Problem-solution fit and Product-market fit
  - Startup way of working
  - Technical debt
  - Influences of multiple stakeholders
  - Competence and financial situations
- Components of software startups
  - Organization
  - Competence
  - Product
  - Way of working
  - Startup ecosystem

# Software startup

- Processes, framework, practices for software startup engineering
  - Lean approach to developing product and businesses
  - Minimum Viable Product
  - Requirement Engineering
  - Minimum Viable User Experience
  - Design thinking
  - Customer journey
  - Growth hacking
- Continuous experimentation in startups
  - Goal-driven hypotheses
  - Build-measure-learn loops
  - Hypothesis driven engineering

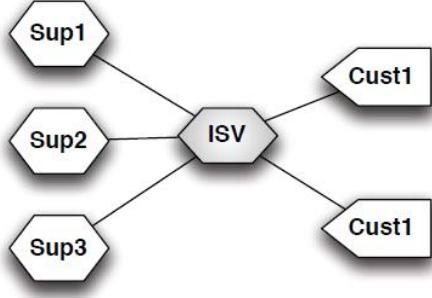
# Software ecosystem

- “*a set of businesses functioning as a unit and interacting with a shared market for software and services, together with the relationships among them*” Messerschmitt and Szyperski (2003)
- Key elements:
  - A central hub: an owner of the platform
  - A platform: Usually technological platform
  - Niche players: who generate value from the platform
  - Example: app stores, open source projects, etc

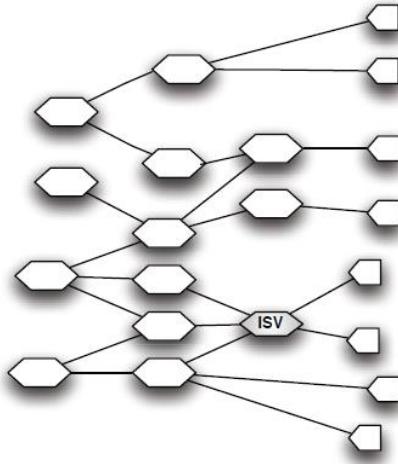


# Software ecosystem

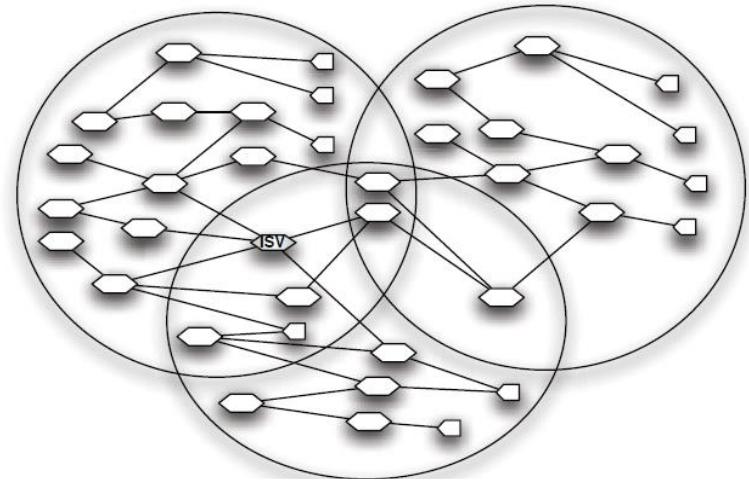
- Scope of ecosystem



A. A hub  
between  
suppliers  
and customers



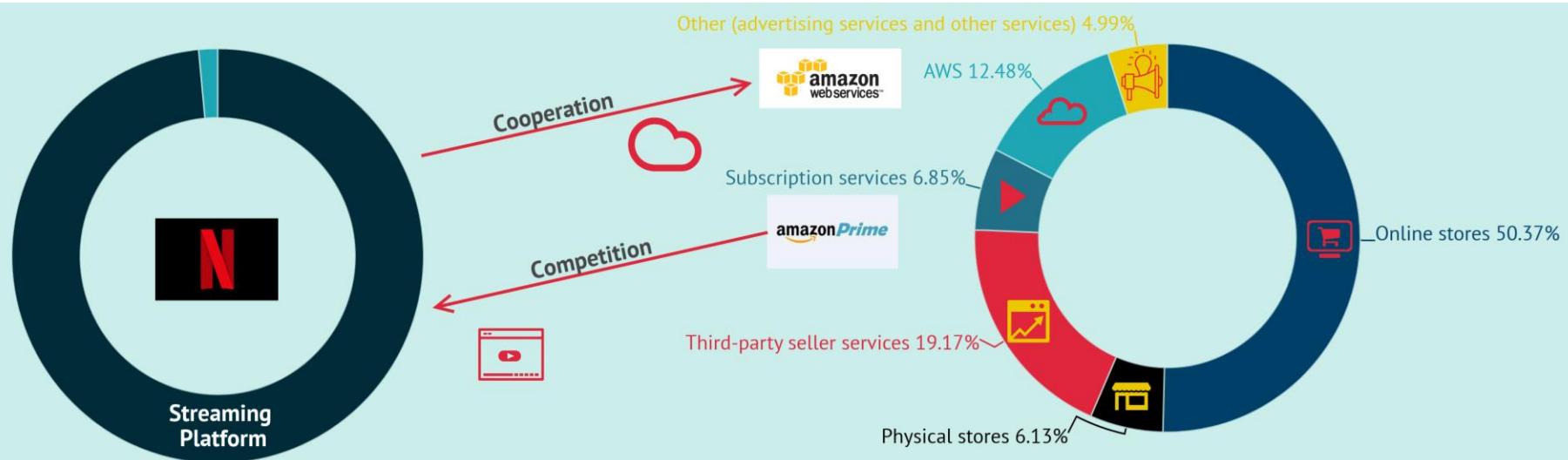
B. The hierarchy  
of hubs



C. Dynamic relationships  
among hubs, suppliers  
and customers

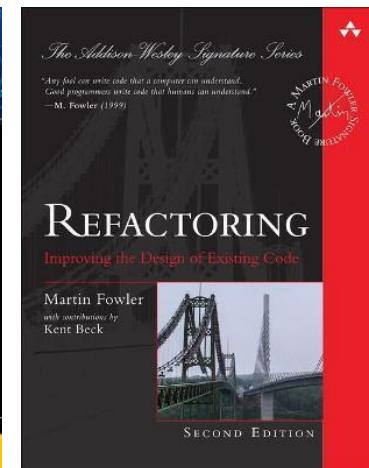
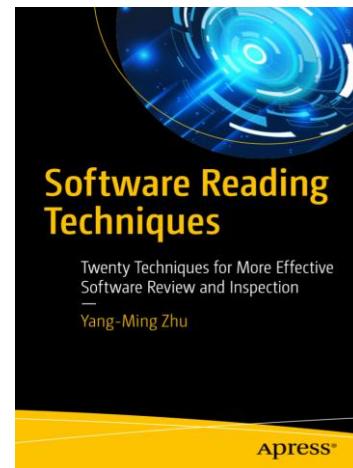
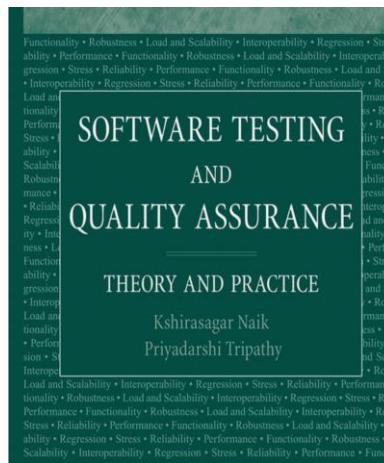
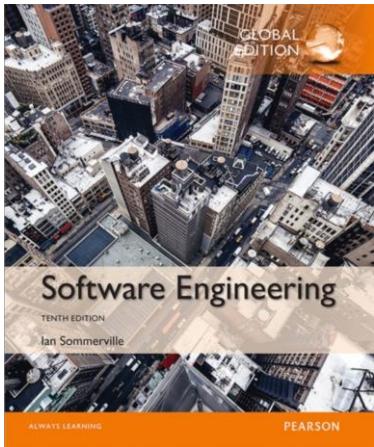
# Software ecosystem

- Scope of ecosystem
- Healthiness of ecosystem
- Collaboration vs. Competition
- Sustainability



# Curriculum

- Journal and conference articles (will be uploaded to blackboard)
- Some chapters from books (accessible in NTNU library)



(1<sup>st</sup>/2<sup>nd</sup> edition)

# Teaching goals

- **Apply** goal-oriented requirement engineering method and requirement boilerplate to identify and model requirements;
- **Identify** and correct typical requirements quality issues;
- **Apply** different testing and code review, analysis, and refactoring strategies;
- **Explain** the industrial state of the practice methods of verifying and validating high-assurance software-intensive system;

# Teaching goals (Cont')

- Understand the state of the practice agile methods, such as DevOps, and apply corresponding tools;
- Explain the issues and solutions of distributed and collaborative software development and maintenance;
- Explain key issues and solutions to managing and operating large and complex software-intensive systems.

# Evaluation and grading

- Exercises: 50%
  - Exercise 1: 10%
  - Exercise 2: 25%
  - Exercise 3: 15%
- Exam (3rd June): 50%
- You have to pass both!
  - You have to hand in **all** the exercises **and** get a passing grade to be allowed to take the exam
  - If you fail the exam, you will fail the course

Exercises	Exam	Final grade
Fail	Fail	Fail
Fail	Pass	Fail
Pass	Fail	Fail
Pass	Pass	Combine exercise and exam grades

# Exercise 1

- The purpose is to let students practice
  - Software requirement elicitation methods
  - Evaluating software requirements quality
- You need to
  - Based on an existing medium-sized web application, collect and understand new requirements you get from a peer group
  - Document the new requirements in high quality

# Exercise 2

- The purpose is to let students practice
  - DevOps tools
  - Extending existing code
  - Software testing methods
- We will ask you extending the existing system and test the existing code
- You need to
  - Apply DevOps tools
  - Develop new code
  - Run tests manually and write test scripts

# Exercise 3

- The purpose is to let students practice
  - Code review
  - Code analysis
  - Code refactoring
- You need to
  - Do manual code review of the existing code
  - Refactor the code based on your review and analysis results

# Exercise tasks and schedule

The exercise introduction lectures are digital via Zoom on Tuesday 08:15 AM - 10:00 AM.

The exercise introduction lecture schedule is in the following Table.

Exercises schedule				
ID	Weeks	Introduction lecture	Start	Deliverable
1	3-7	18 Jan. 08:15 - 10:00	18 Jan.	<ul style="list-style-type: none"><li>Final report of exercise 1. 14 February at 23:59 (Monday), Week 7</li></ul>
2	7-12	15 Feb. 08:15 - 10:00	15 Feb.	<ul style="list-style-type: none"><li>Final report of exercise 2. 21 March at 23:59 (Monday), Week 12</li></ul>
3	12-14	22 March 08:15 - 10:00	22 March	<ul style="list-style-type: none"><li>Final report of exercises 3. 4 April at 23:59 (Monday), Week 14</li></ul>

In other weeks without exercise introduction lectures, we will have TAs online via the same Zoom link to answer questions.

# Exercise groups

- 1-2 students in each group
- To form a group
  - Send an e-mail with a list of the names and e-mails of the group members to [tdt4242@idi.ntnu.no](mailto:tdt4242@idi.ntnu.no)
- If you don't have a group
  - Send an e-mail to [tdt4242@idi.ntnu.no](mailto:tdt4242@idi.ntnu.no)
  - You will be assigned to a group
- Deadline: 1 Feb.

# Deadline for exercises

All exercises will have a deadline for delivery.

This deadline may only be exceeded after agreement with the

- course responsible
- teaching assistant (email: [tdt4242@idi.ntnu.no](mailto:tdt4242@idi.ntnu.no))

If no such agreement exists, we will deduct 20% points on the grade for any obligatory exercise for each week it is delayed.

# Reference group

- A group of 2-3 students that have a special duty to provide feedback about the course during the semester
- We'll have 2-3 meetings where we can discuss content and form of lectures and assignments
- The group should be formed during the first 3 weeks, so please nominate yourself (or others)
- More information (In Norwegian) →  
<http://www.ntnu.no/utdanningskvalitet/roller.html#Studentene>

# About you

I need to know a little more about you to adapt my teaching focuses and exercises!

# Empirical research on Software Ecosystem (SeCo) and Software Startup

Anh Nguyen Duc

# Agenda

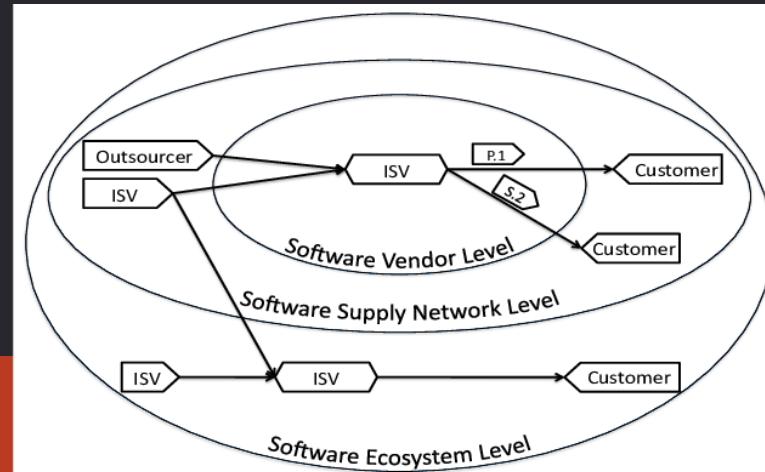
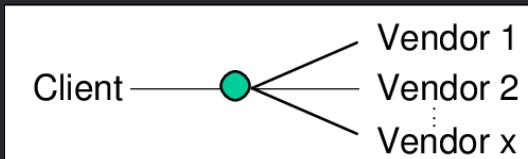
---

- Software Ecosystem (SECO)
- Software Startups

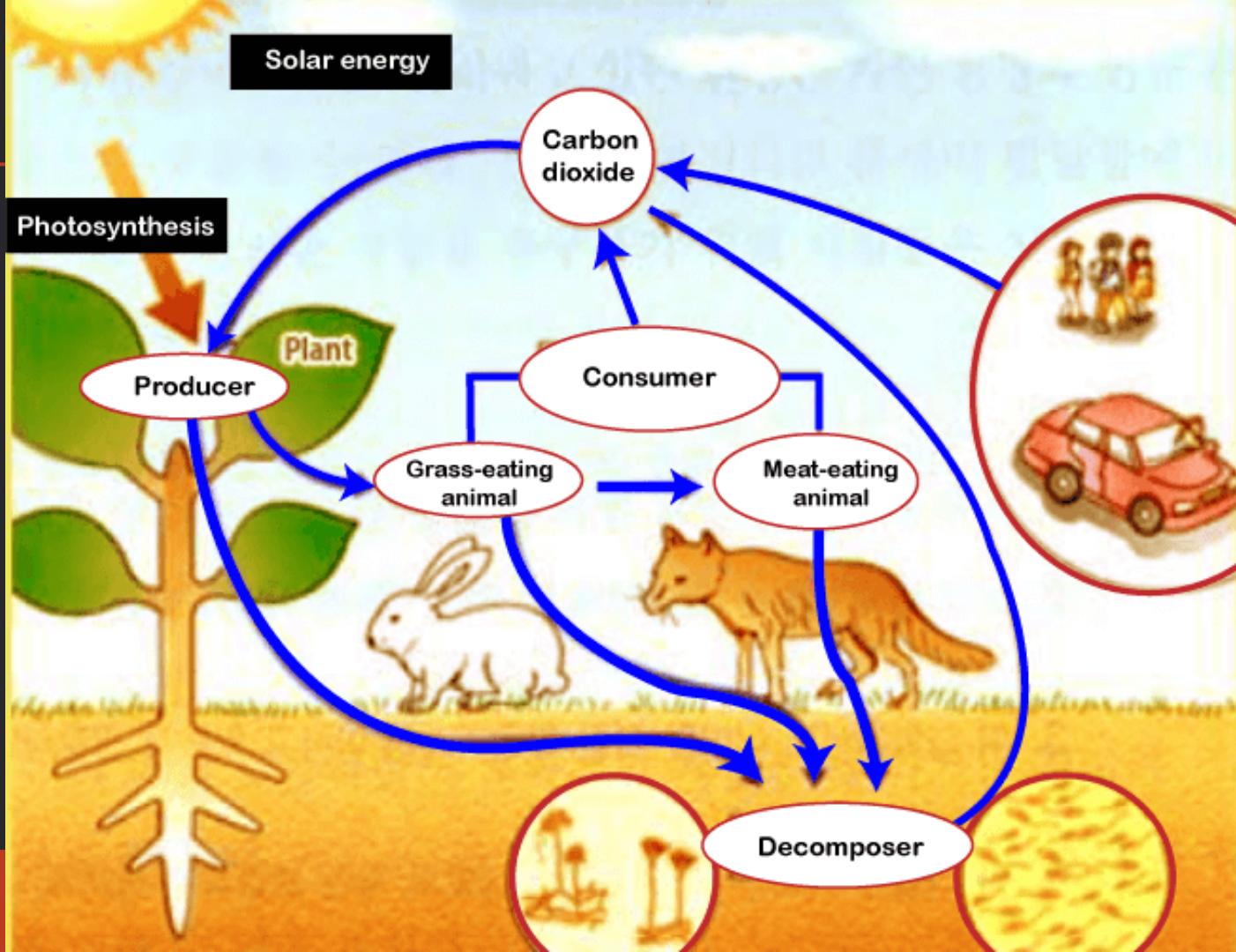
# Software Ecosystem

# Introduction

- Software ecosystems is an emerging trend within the software industry
- The shift from closed organizations and processes towards open structures
- Ecosystem metaphor is used to understand relationships among stakeholders



# Biological ecosystem



# **Definitions of Software Ecosystem (SeCo)**

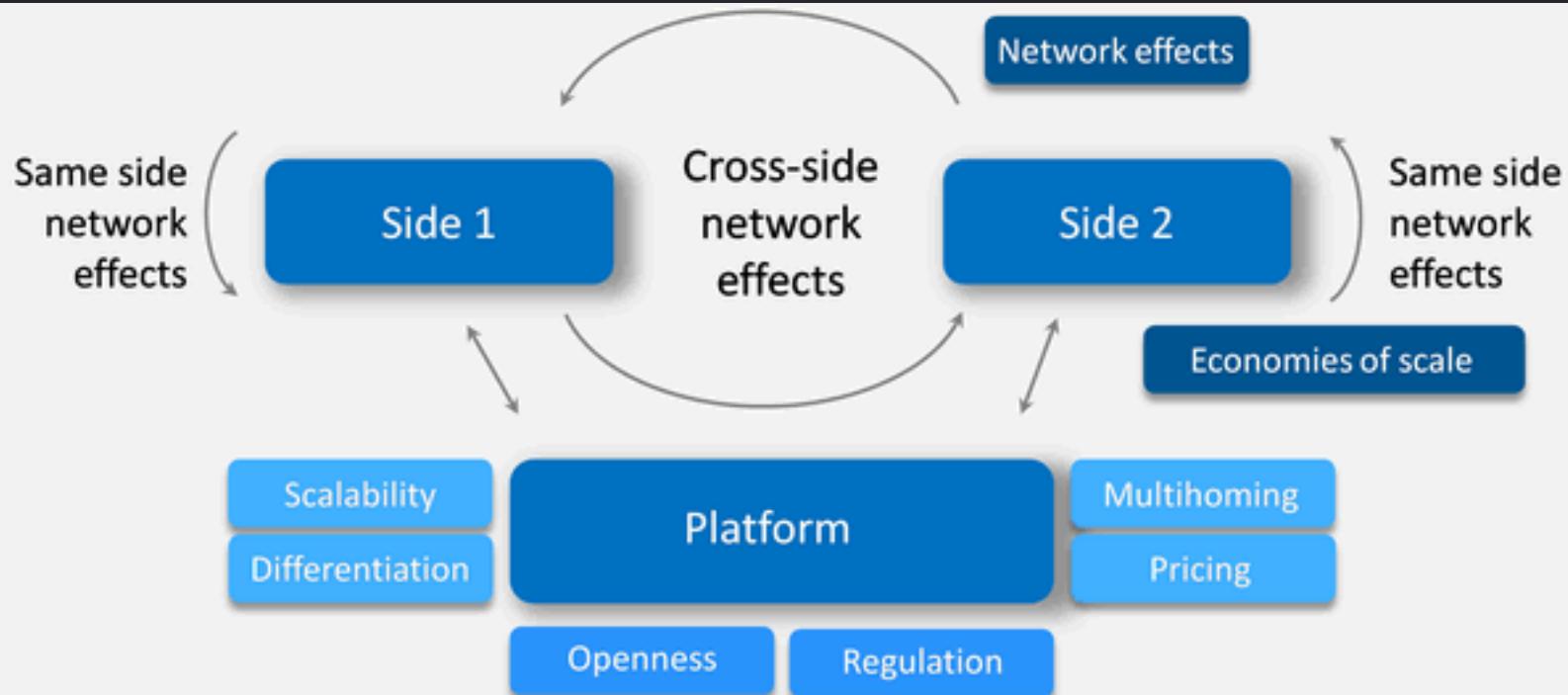
- SeCo as an informal network of independent units that have a positive influence on the economic success of a software product and benefit from it (Kittlaus and Clough)
- SeCo as consisting of the set of software solutions that enable, support, and automate the activities and transactions by the actors in the associated social or business ecosystems and the organizations that provide these solutions (Bosch)
- a set of actors functioning as a unit and interacting with a shared market for software and services, together with the relationships among them (Jansen)
- Three shared concepts stand out in these definitions:
  - (1) actors, organizations and businesses,
  - (2) networks and social or business ecosystems, and
  - (3) software.

# Menti.com

Examples of Software  
Ecosystems you know?

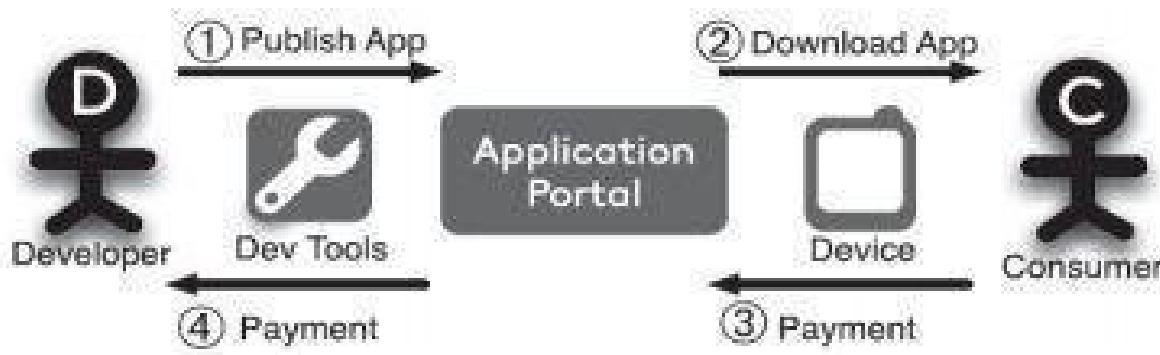


# Foundation – two-sided market theory



# Software Ecosystem (SECO)

- App stores: Apple, Google, Microsoft, etc



# Software Ecosystem (SECO)

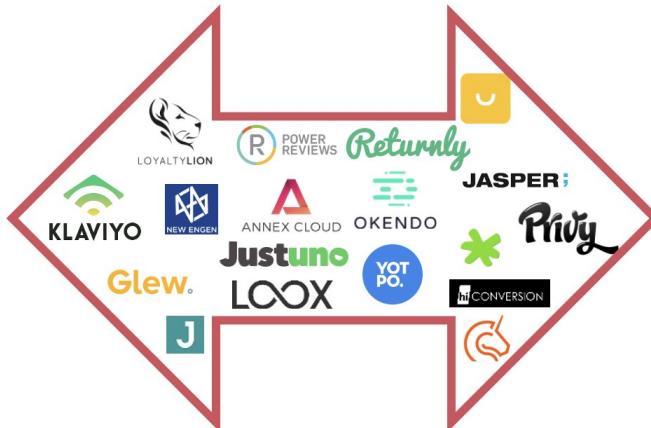
- Service providers on top of Shopify ecosystem

Integrated Back Office

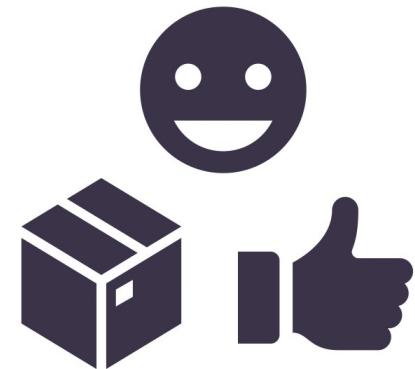


Shipping	Inventory	Payments
Analytics	Order Management	Promotions

Front Office Software Vendor



Customer



# Other examples of Software Ecosystems



- **Standards** - XML, BPM, OSGi, J2EE, Corba, SEPA, etc.
- **Products** - OpenOffice, Microsoft Word, SAP BusinessOne, Grand Theft Auto, etc.
- **Hardware** - Playstation 3, HTC Diamond, PDAs, BMW 5 series, etc.
- **Platforms** - .Net, Facebook, Android, OS X, etc.

# Types of SeCo

Example: The Apple iOS software ecosystem is based on a software platform and coordinated by a privately owned entity with a commercial extension market to which participants can submit extensions after making a payment .

- Core technology
  - software platform,
  - software service platform,
  - software standard
- Coordinated by
  - Community
  - Private organization
- Extension market
  - no extension market,
  - a list of extensions,
  - an extension market,
  - a commercial extension market,
  - multiple extension markets
- Accessibility
  - for free,
  - after a screening,
  - after making a payment

# Characteristics of SeCos

---

- **Complexity** - In a platform ecosystem with numerous actors, complexity must be controlled somehow to reduce risk of gridlocks, unpredictable ripple effects and co-innovation problems
- **Productivity** - A network's ability to consistently transform technology and other raw materials of innovation into lower costs and new products. Simple to measure: return on invested capital.
- **Robustness** - Should be capable of surviving disruptions such as unforeseen technological change.
- **Niche creation** - the ecosystem's capacity to increase meaningful diversity through the creation of valuable new functions or niches.

# A healthy Software Ecosystem

---

- ❑ A stable and sustainable platform and mechanism to govern the platform
- ❑ Developers, software firms making benefits from collective innovation
- ❑ Consumers receive high quality services/ products, opportunities for value co-creation

# A healthy Software Ecosystem

---

- ❑ Sustainability is the key!
- ❑ How to design and maintain a sustainable software ecosystem ?



# A healthy Software Ecosystem

---

- Sustainability is the key!
  
- A viable business model
- A well-organized inter-organizational interaction model
- A paradigm supporting Collaboration
- A mechanism to allow Coopetition



# The 3C in SeCo

---

COOPERATION



COLLABORATION

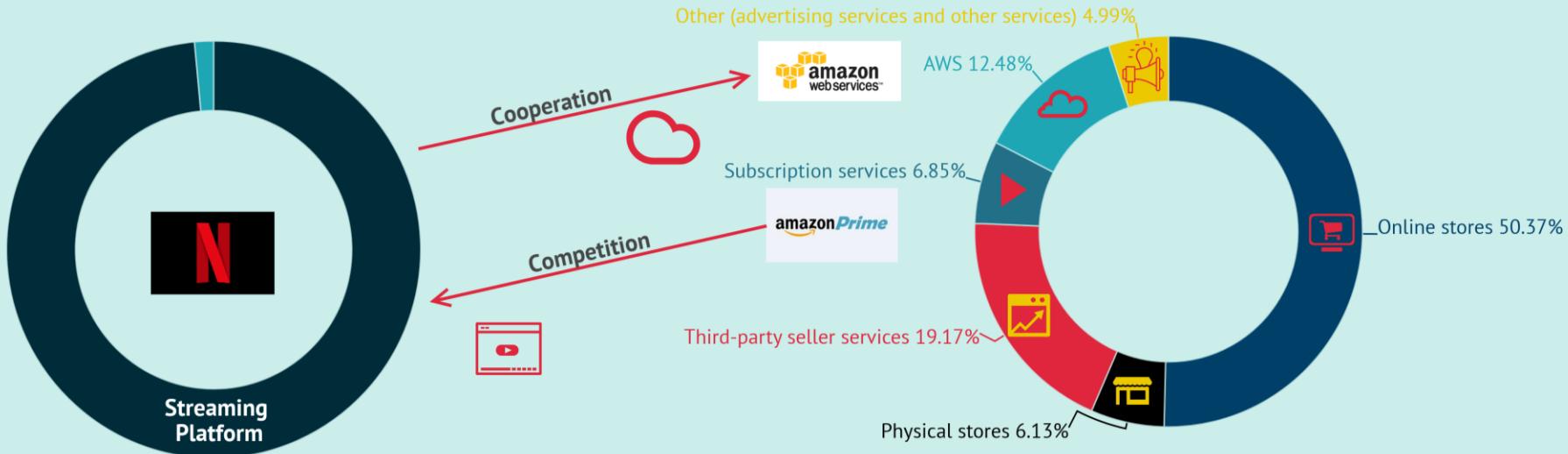


COOPETITION

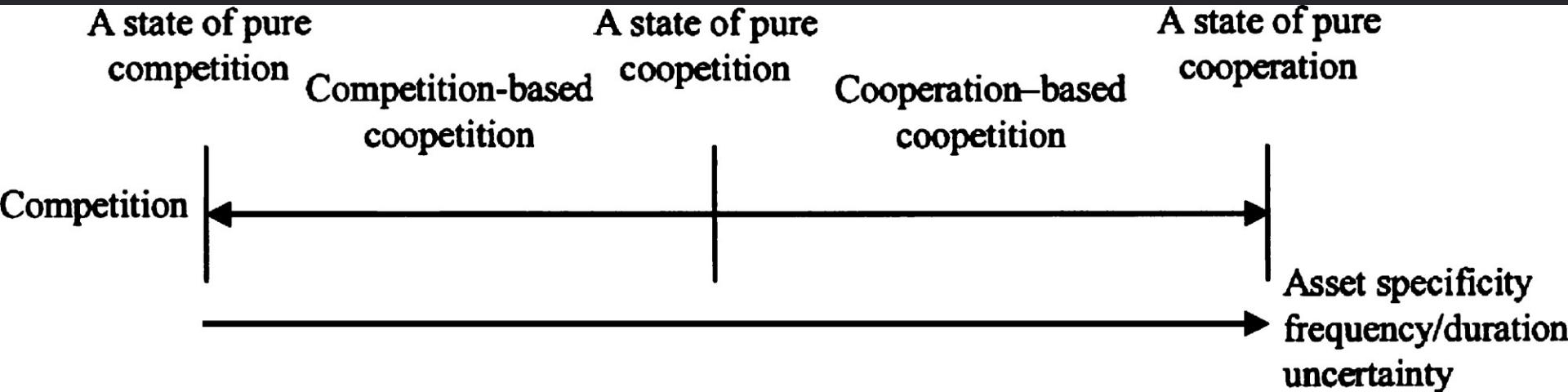


# Coopetition in a nutshell

Netflix uses AWS cloud services while compete with Amazon Prime

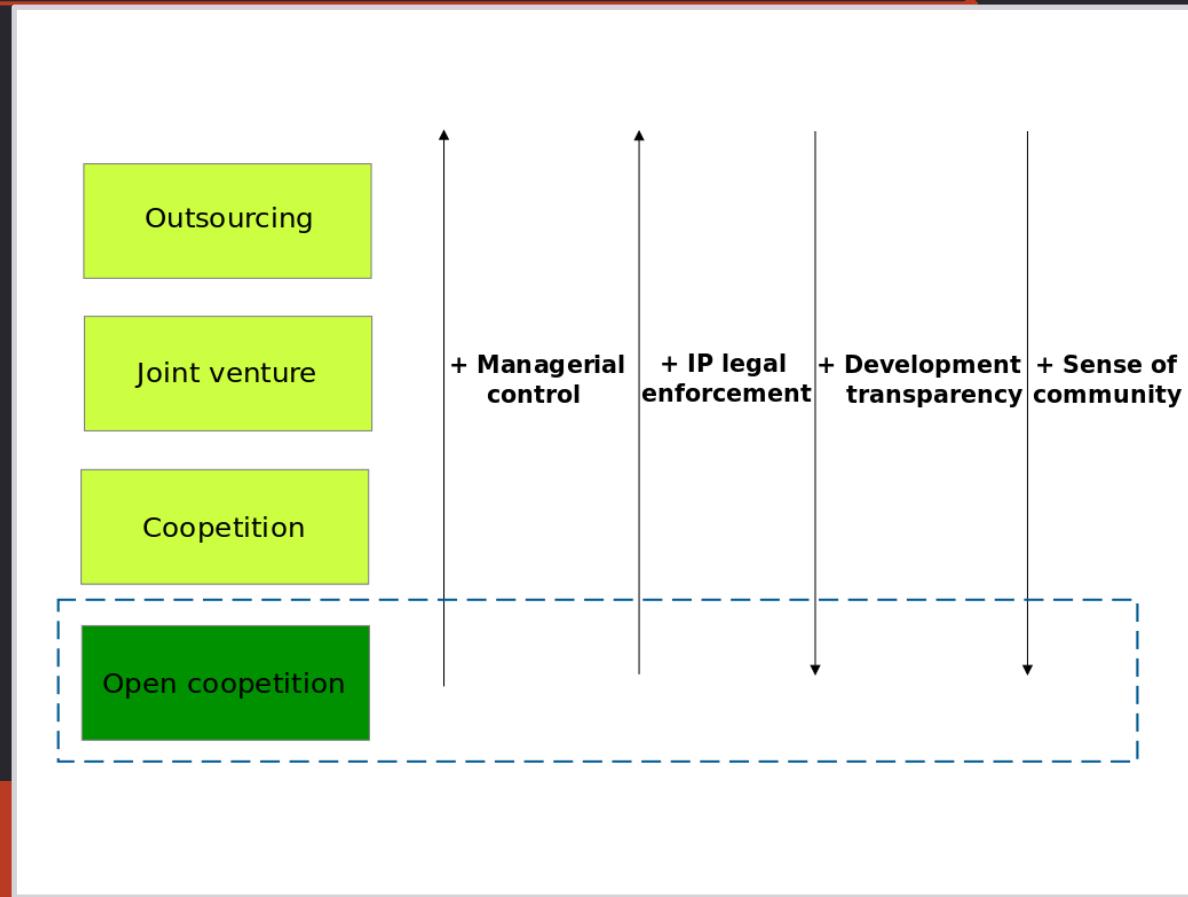


# A model of coopetition



# Open coopetition

Coopetition +  
Open Source



*How do commercial  
companies maintain both  
collaboration and  
competition in an open  
source software  
ecosystem?*

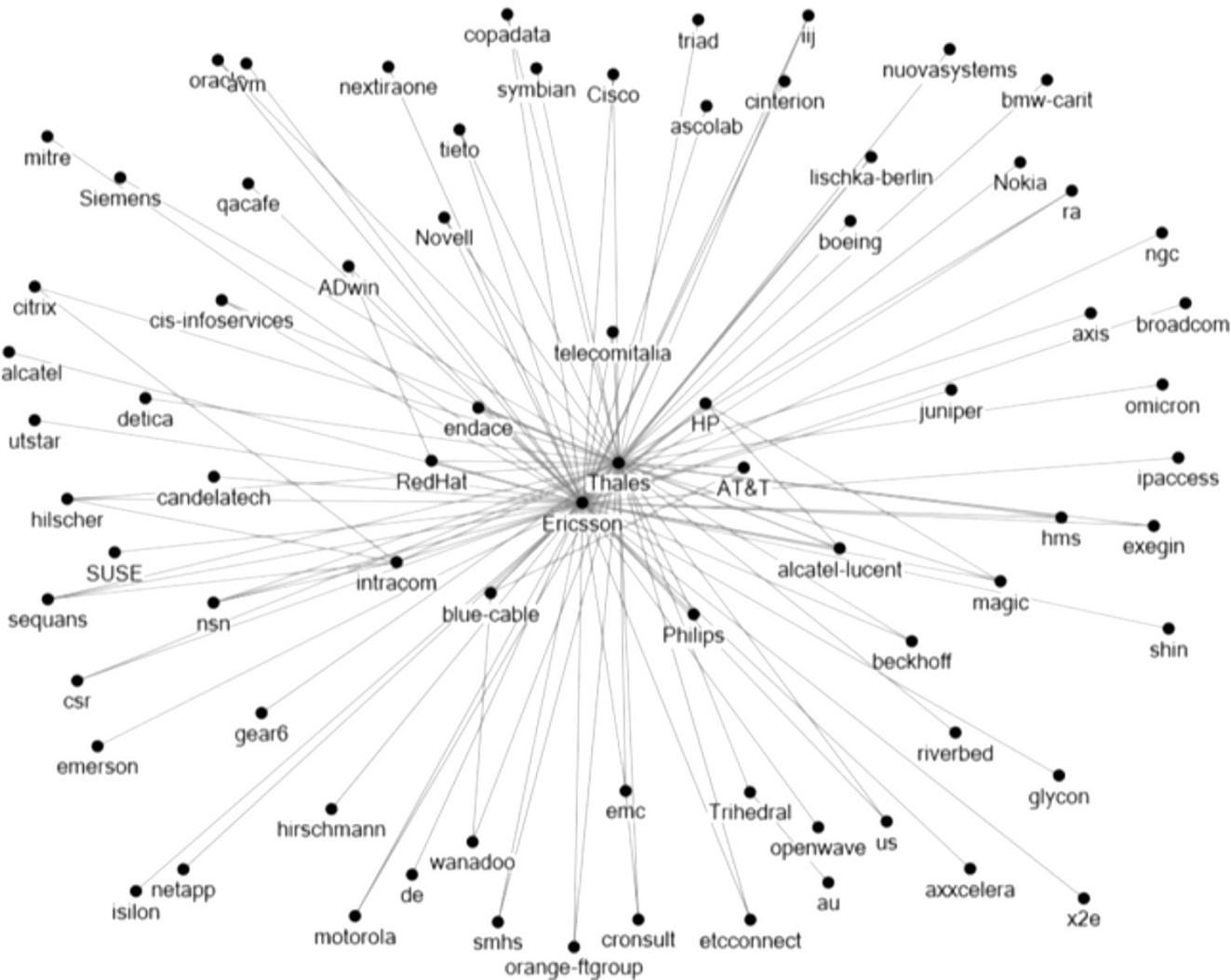


# Our cases

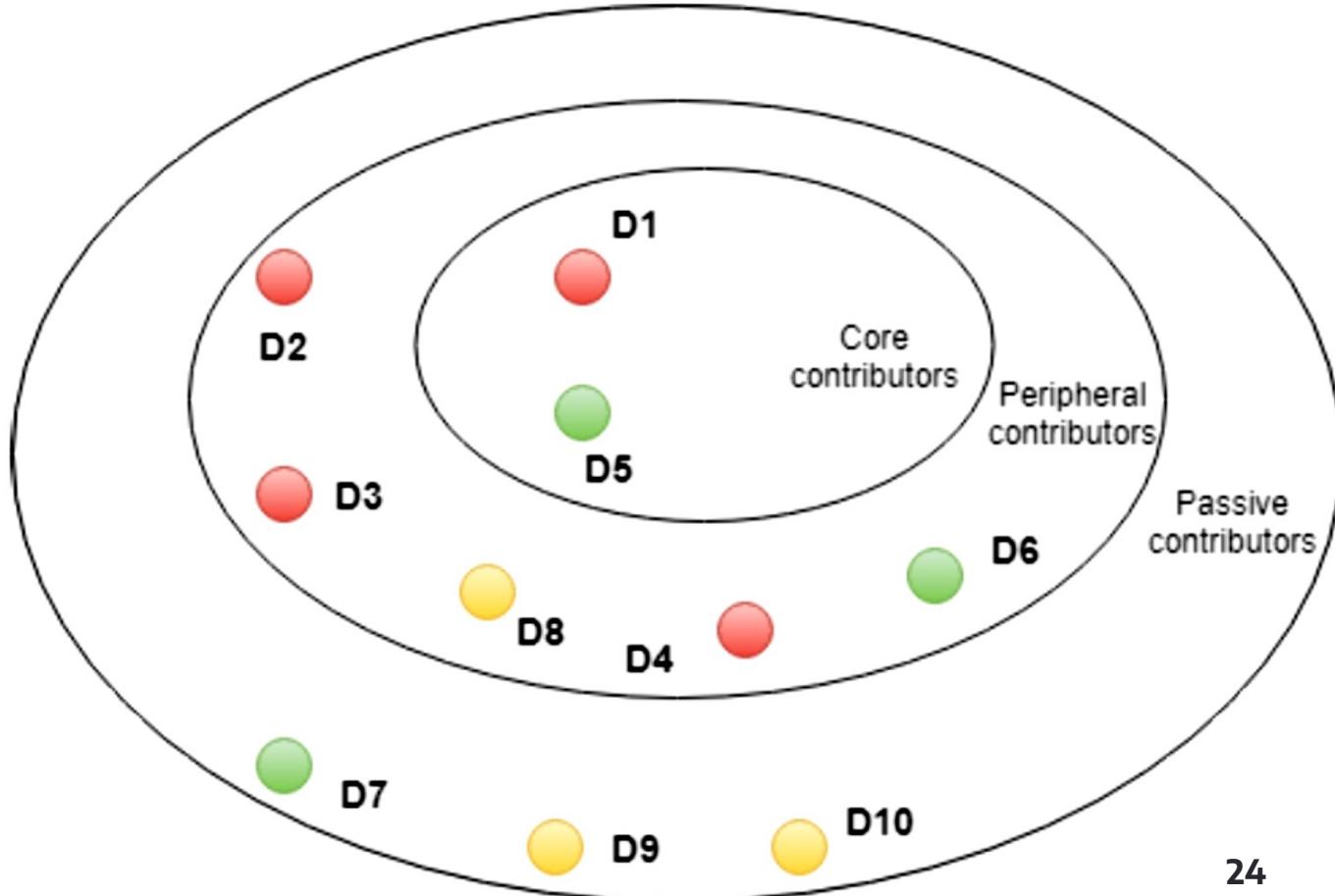
---



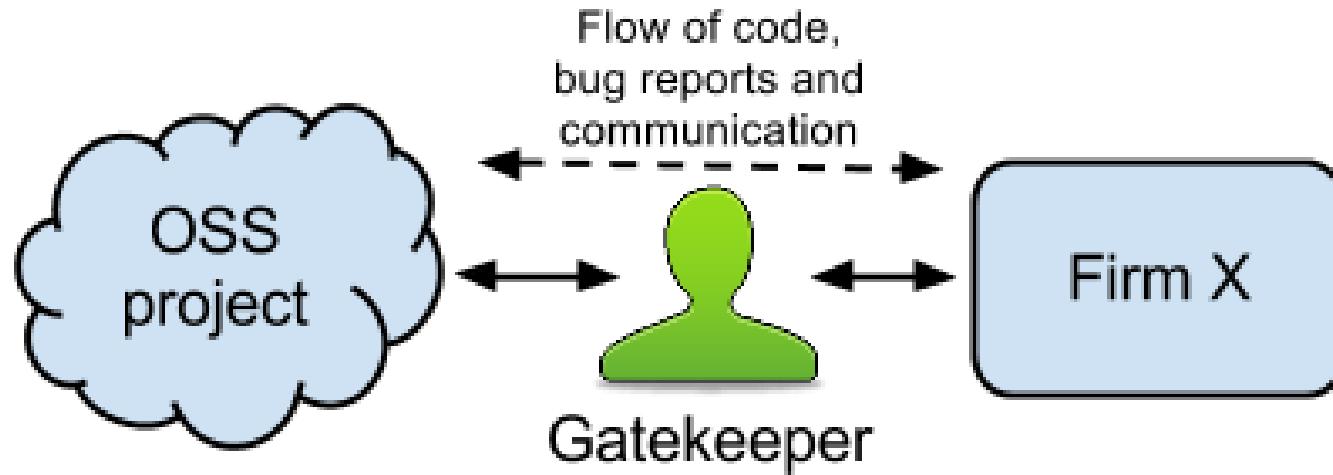
# Wireshark ecosystem - social analysis



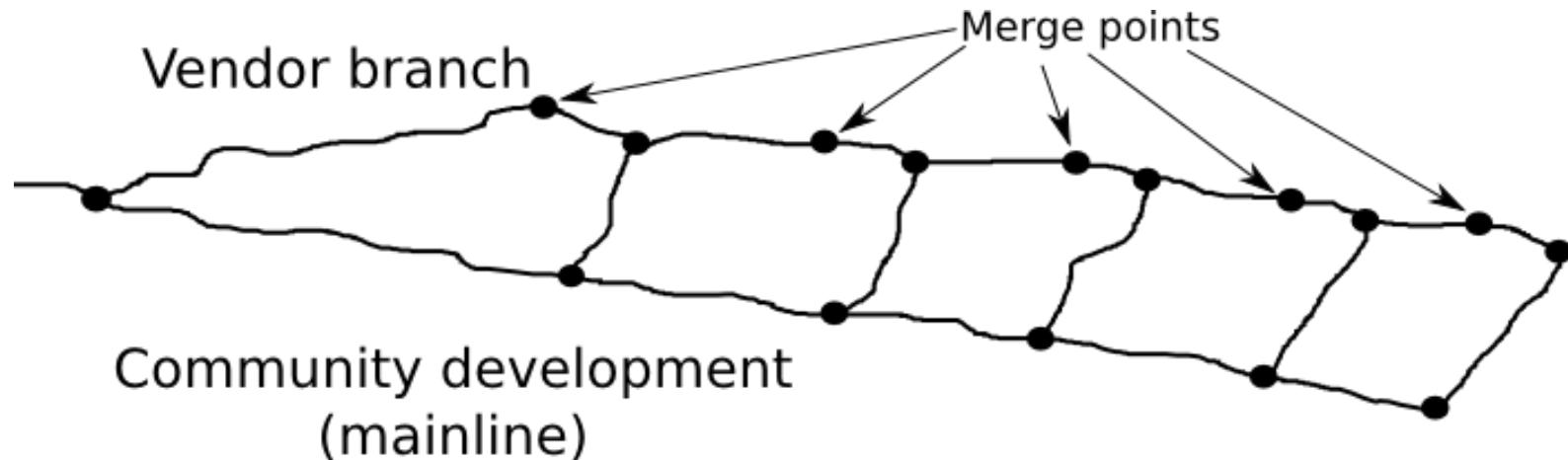
# Wireshark ecosystem - conceptual layers



# How do commercial firms manage coopetition with other firms in such context?

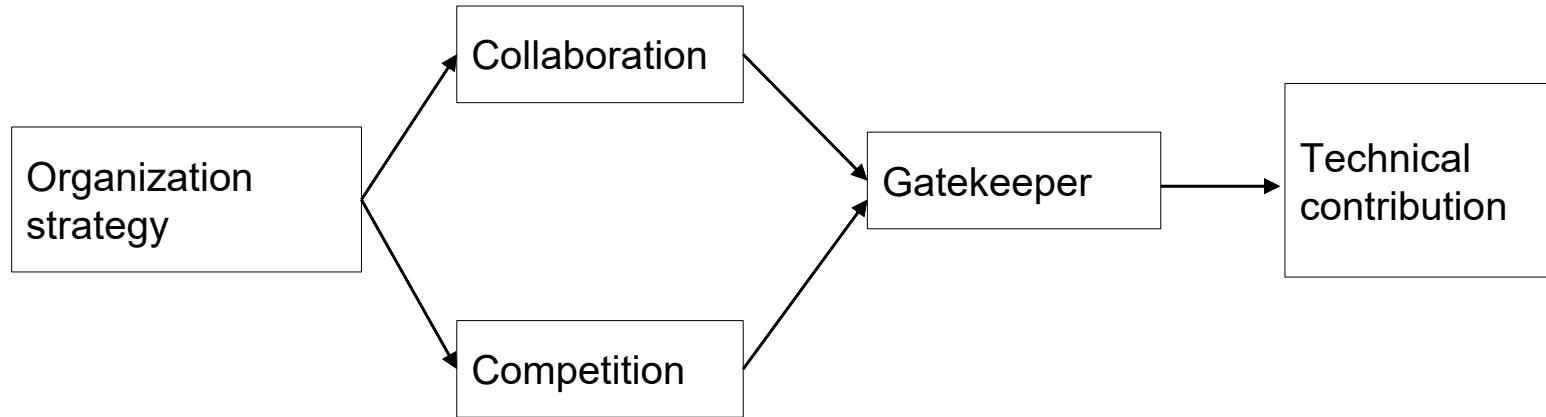


# How do commercial firms manage coopetition with other firms in such context?



- Open-core sourcing policy
- Upstream first - get features integrated into open source projects before integrating them into firm's product

# A model of open coopetition in SECO



# Software Startups

## Software startups

---

Startup companies are unique:

- Little or no operating history
- Limited resources
- Multiple influences
- Dynamic technologies and markets
- Scalable business model

And more ...

# Software startups

---

## Startup companies are unique:

- Software in value proposition
- Innovation focus
- Lack of resources
- High uncertainty
- Time-pressure
- Highly reactive
- Rapidly evolving
- Temporary organizational state
- Little/ no history of stable operations
- Product development and business development in parallel
- Seek for product-market fit
- Strong presence of entrepreneurial personalities
- **60%** of startups do not survive in the first five years
- **75%** of venture capital funded startups fail
- Little rigor and relevance exist in the studies about software startups
- Do not focus on investigating issues and challenges

# Menti.com

Name a software startup  
that you know in Norway



# Software startups



APPSUMO



Opera

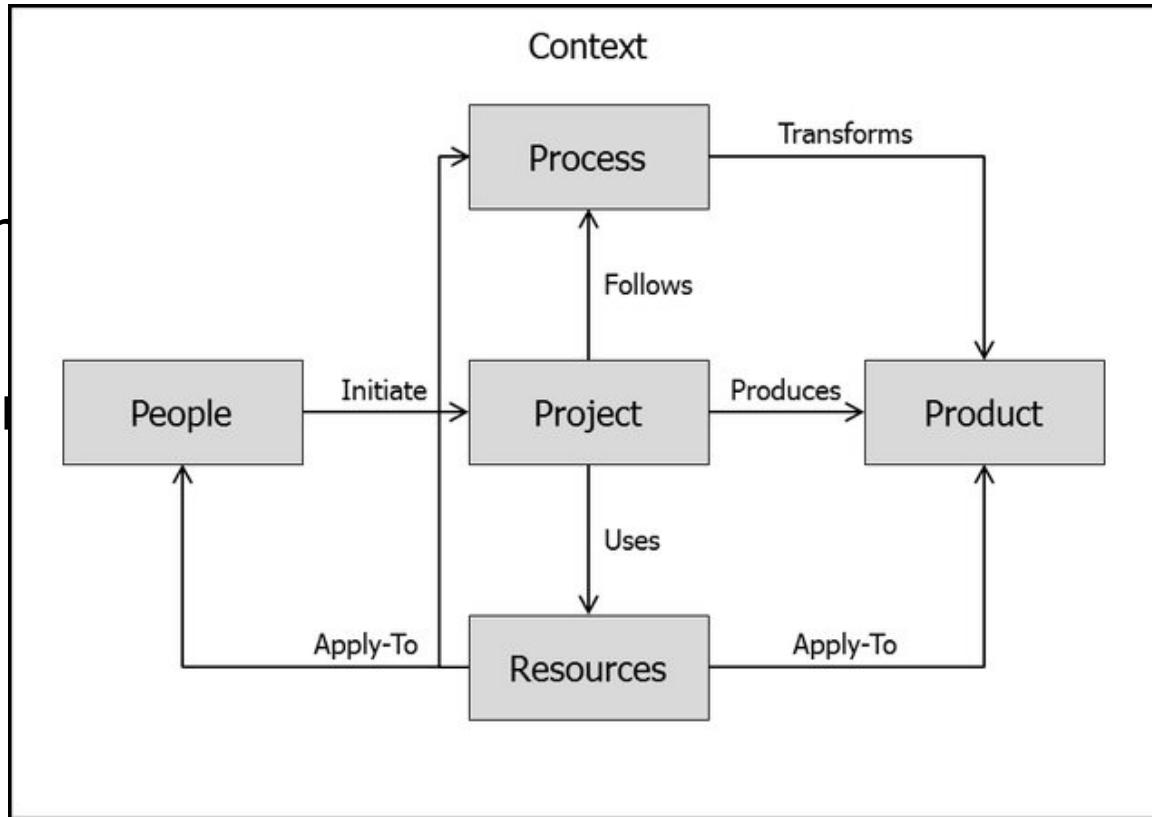
Kahoot!

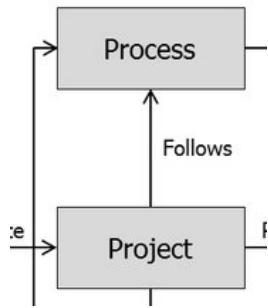


1. J. Melegati, A. Goldman, F. Kon, and X. Wang, "A model of requirements engineering in software startups," *Information and Software Technology*, vol. 109, pp. 92–107, May 2019
2. V Berg, J Birkeland, A Nguyen-Duc, IO Pappas, L Jaccheri, Software startup engineering: A systematic mapping study, *Journal of Systems and Software* 144, 255-274, 2018

# Software startup research

- Empirical
- Reveal and reflect on phenomena





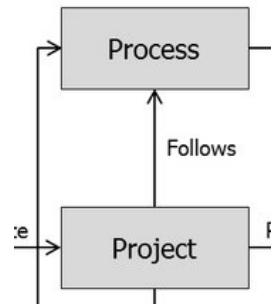
- **Technical debt:**

- when development teams take actions to expedite the delivery of a piece of functionality or a project which later needs to be refactored
- Intentional vs. unintentional
- prioritizing speedy delivery over perfect code

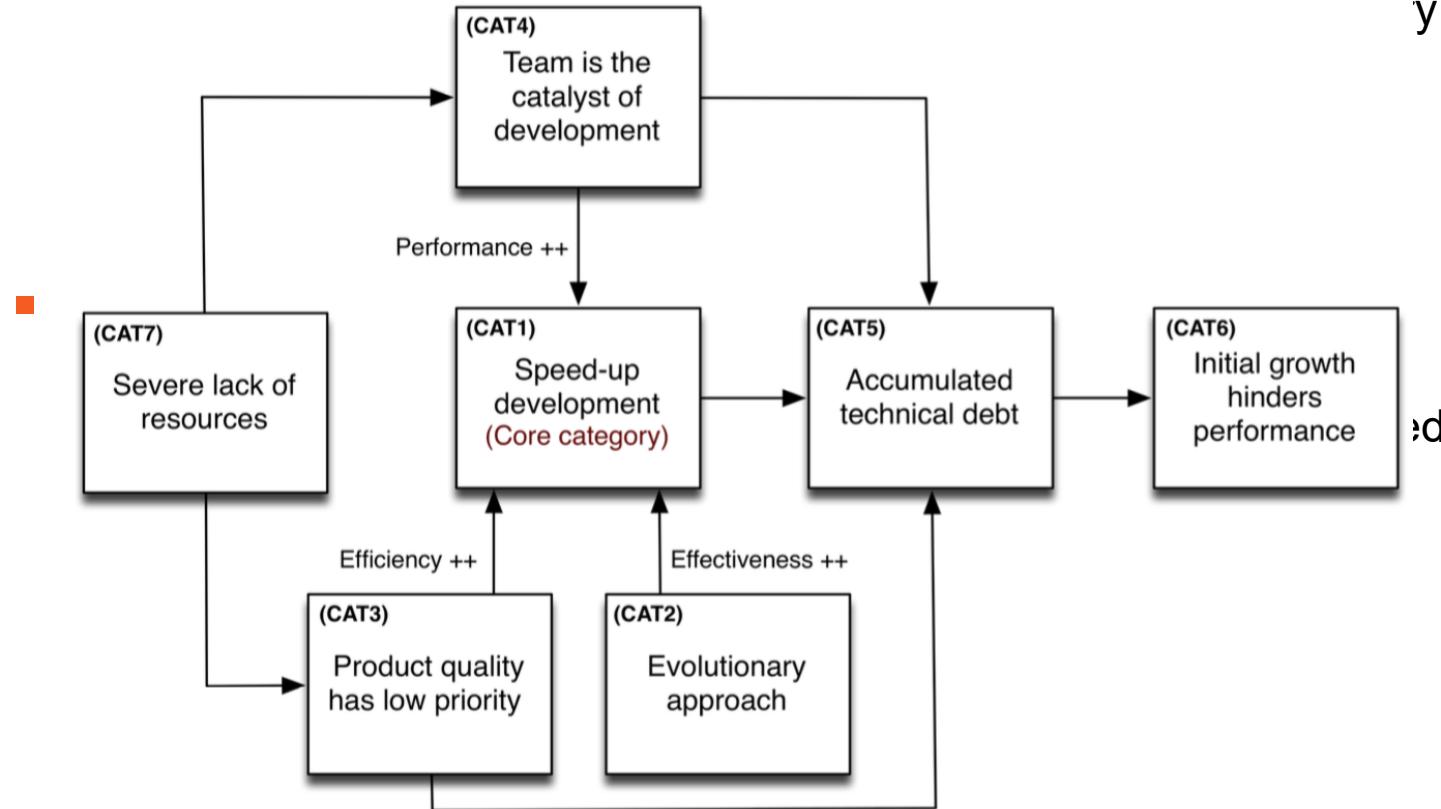
- Technical debt as integral parts of software startups: the need to shorten time-to-market, by speeding up the development through low-precision engineering activities, is counterbalanced by the need to restructure the product before targeting further growth

1. C. Giardino, N. Paternoster, M. Unterkalmsteiner, T. Gorschek, and P. Abrahamsson, "Software Development in Startup Companies: The Greenfield Startup Model," IEEE Transactions on Software Engineering, vol. 42, no. 6, pp. 585–604, Jun. 2016

Context

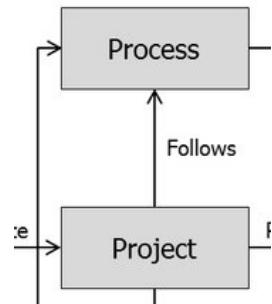


## ■ Technical debt:

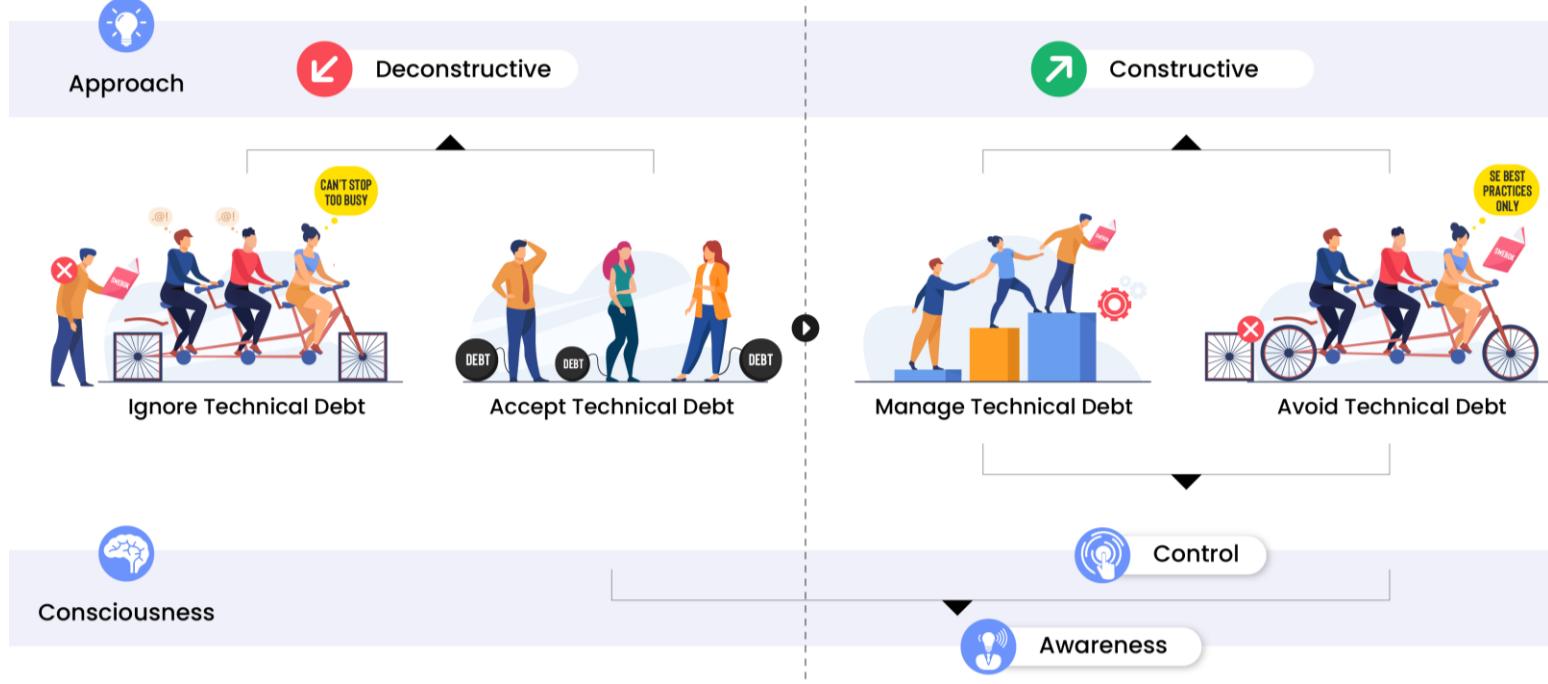


1. C. Giardino, N. Paternoster, M. Unterkalmsteiner, T. Gorschek, and P. Abrahamsson, "Software Development in Startup Companies: The Greenfield Startup Model," IEEE Transactions on Software Engineering, vol. 42, no. 6, pp. 585–604, Jun. 2016

Context

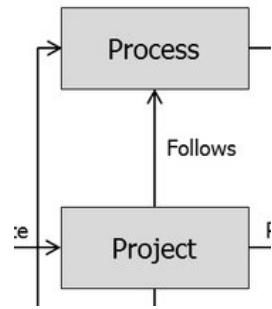


## Technical debt – four facet model in software startups

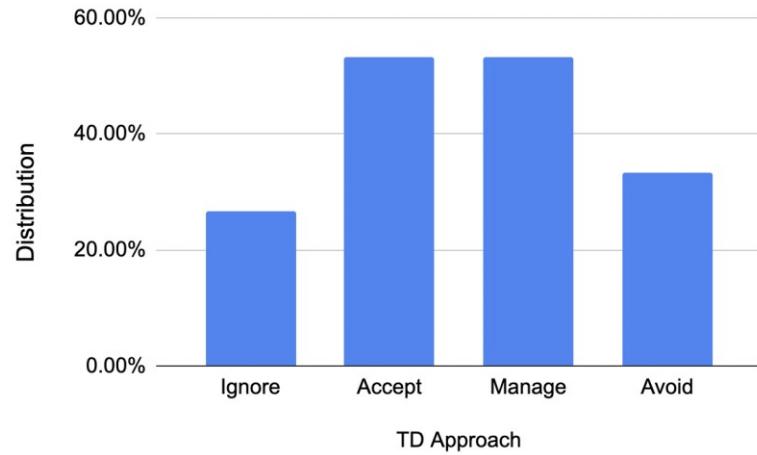


1. Cico et al. Technical Debt Approach and Consciousness in Startups Transitioning to Growth Phase - A Four Facet Model. Submitted to IEEE Access, 2022

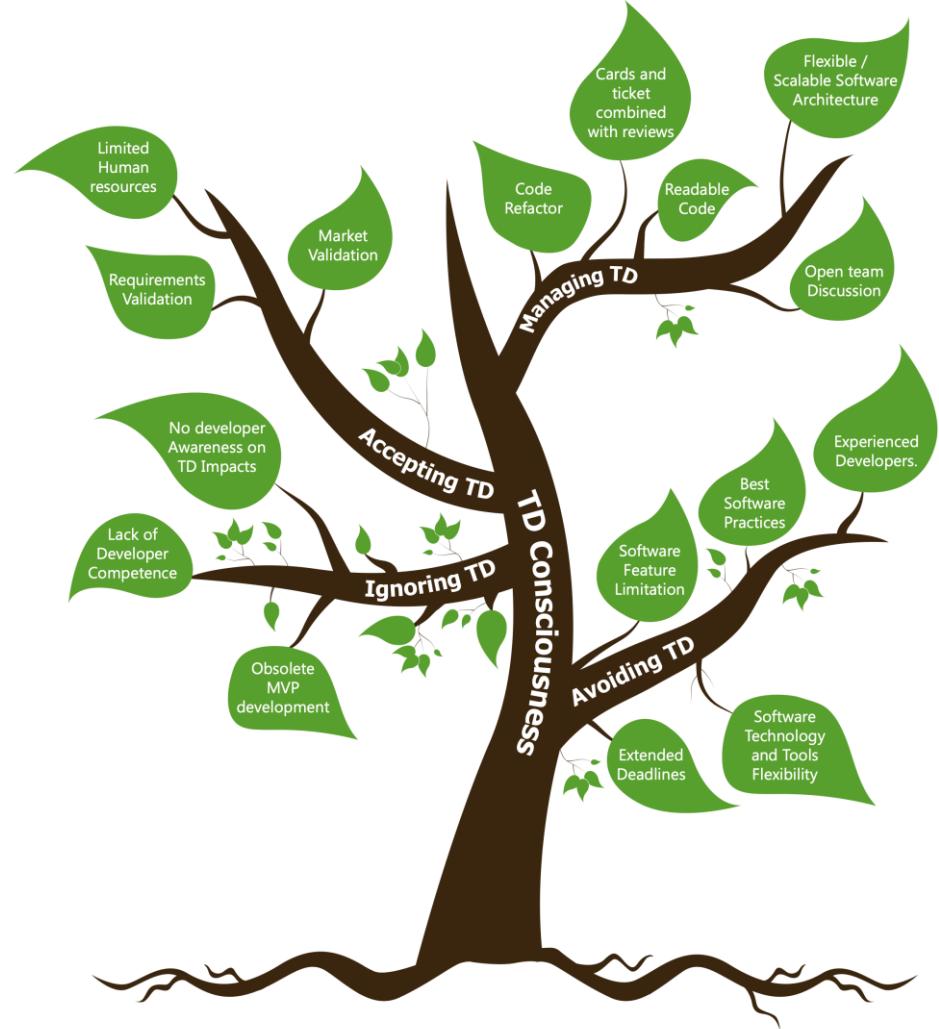
Context



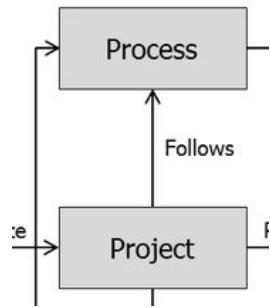
## Technical debt – forms



1. Cico et al. Technical Debt Approach and Consciousness in Software Development. In: *Software Quality and Reliability Access*, 2022

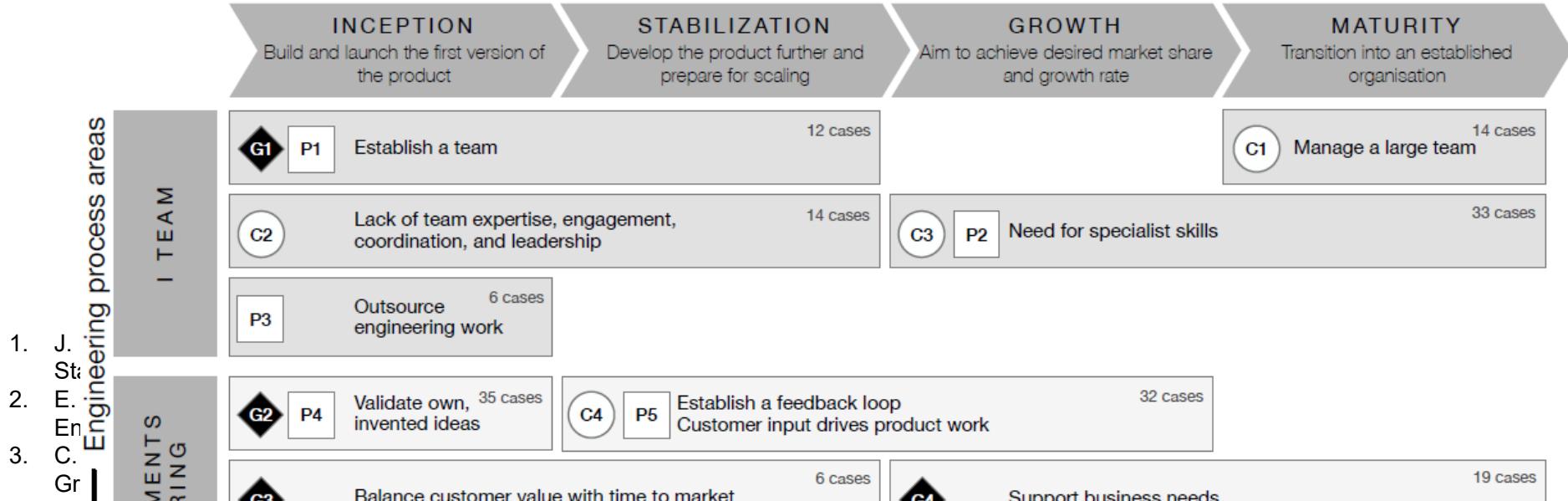


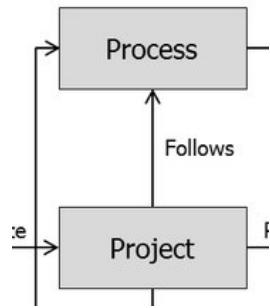
## Context



- Domain knowledge, technical expertise, and teamwork are the key components in the early stages of a start-up. Shortages of any of these components can be compensated with specific practices
- Key difference and difficulty to practice software engineering in

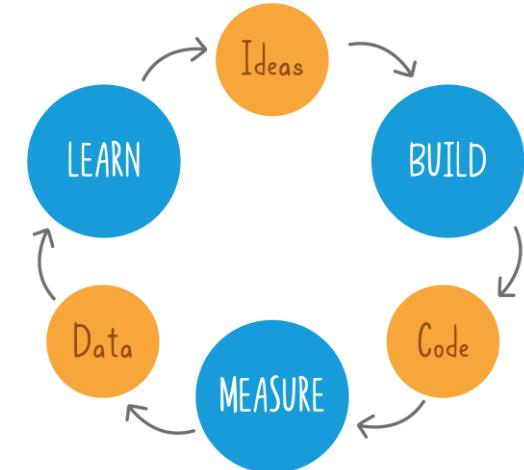
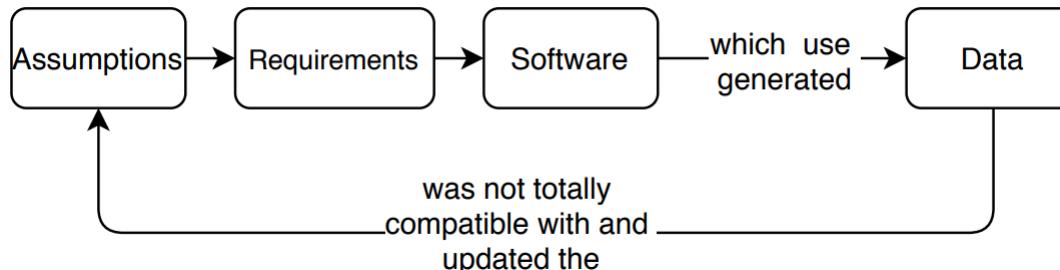
## Start-up life-cycle phases





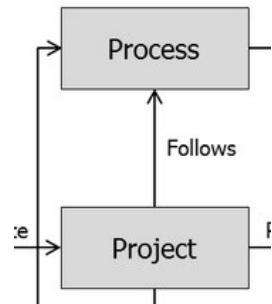
## ■ Hypothesis engineering in software startups

- There are always assumptions
- Decisions are often driven by the instinct of startup founders
- Lean startup – is this really used?

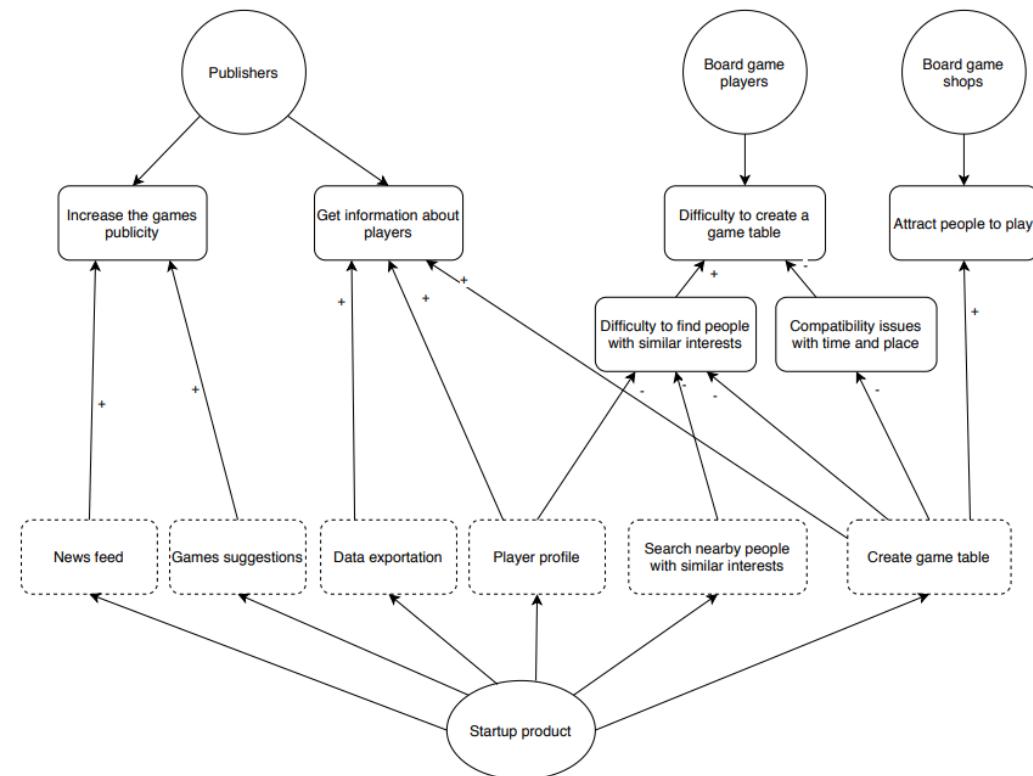


1. J. Melegati, H. Edison, and X. Wang, "XPro: a Model to Explain the Limited Adoption and Implementation of Experimentation in Software Startups," IEEE Transactions on Software Engineering, pp. 1–1, 2020
2. E. Ries, *The Lean Startup: How Today's Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses*, 1 edition. New York: Currency, 2011.

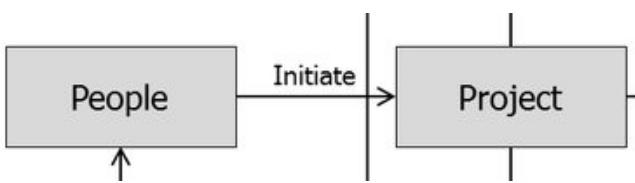
## Context



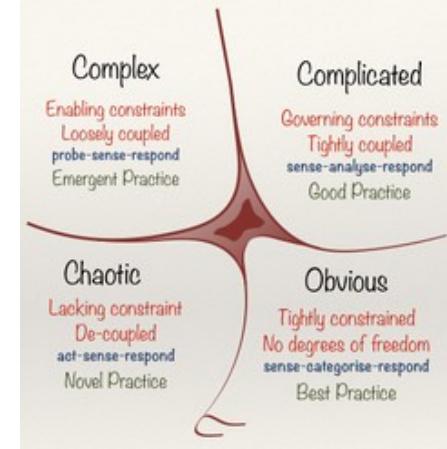
- Cognitive map to visualize the hypotheses and their relationships



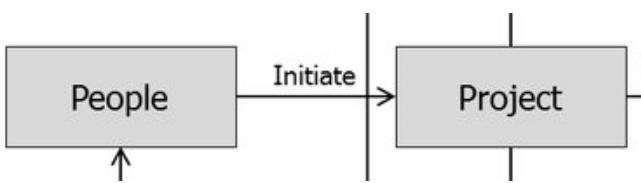
1. J. Melegati, H. Edison, and X. Wang, "XPro: a Model to Startups," IEEE Transactions on Software Engineering, ...
2. E. Ries, *The Lean Startup: How Today's Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses*, 1 edition. New York: Currency, 2011.



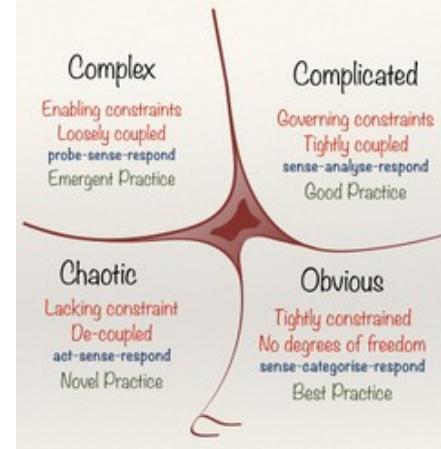
- People:
  - ▣ Startup Founders
  - ▣ Stakeholders in Startup Ecosystems
- Cynefine-based categories of startups' decisions:
- Simple decisions:
  - ▣ Setting up social media accounts for the startup
  - ▣ Focusing on acquiring funding
  - ▣ Creating a common team calendar



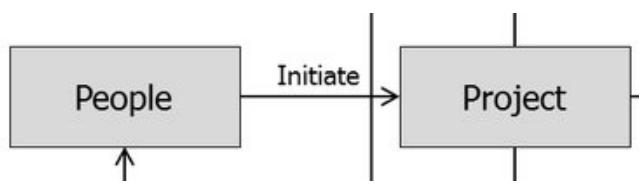
Nguyen Duc, A., Shah S. and Abrahamsson, P. (2016). Towards an early stage software startups evolution model. Euromicro SEAA



- Cynefine-based categories of startups' decisions:
- Complicated decisions:
  - Practicing public speaking
  - Deciding on different pricing models
- Complex decisions:
  - Validating the most important features
  - Gathering a team

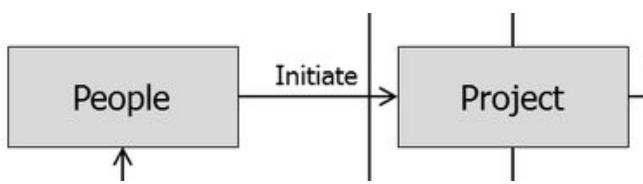


Nguyen Duc, A., Shah S. and Abrahamsson, P. (2016). Towards an early stage software startups evolution model. Euromicro SEAA

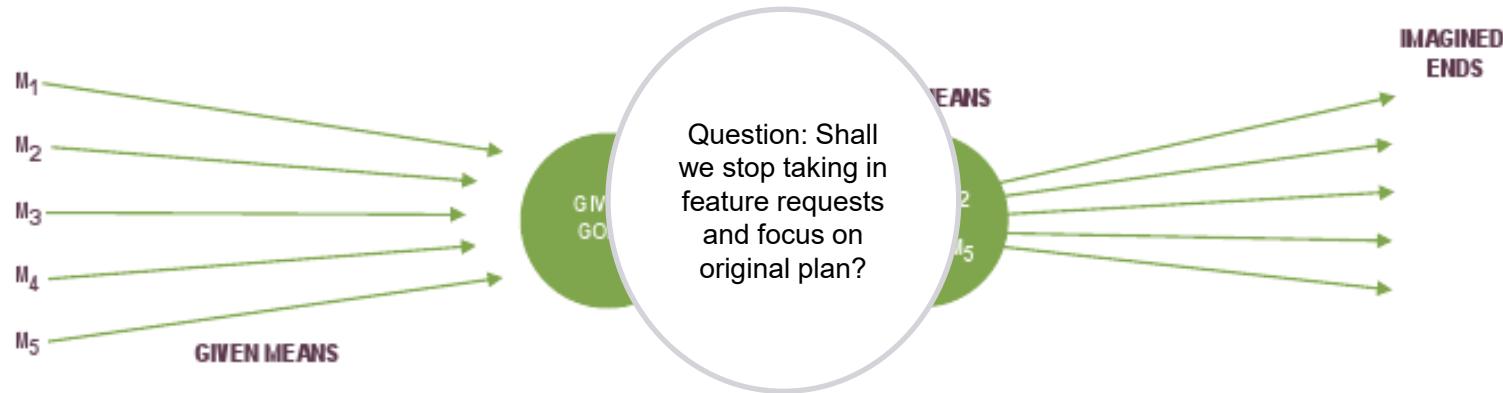


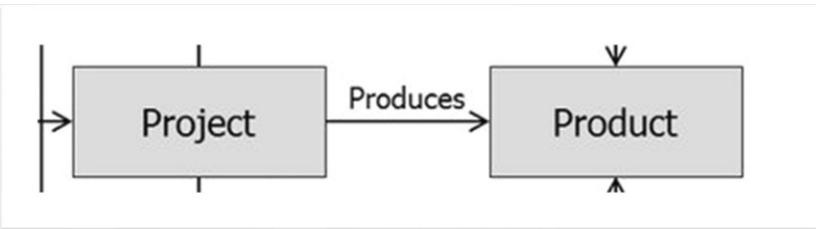
- Two logics of making decisions: Effectuation and Causations
- Effectuation:





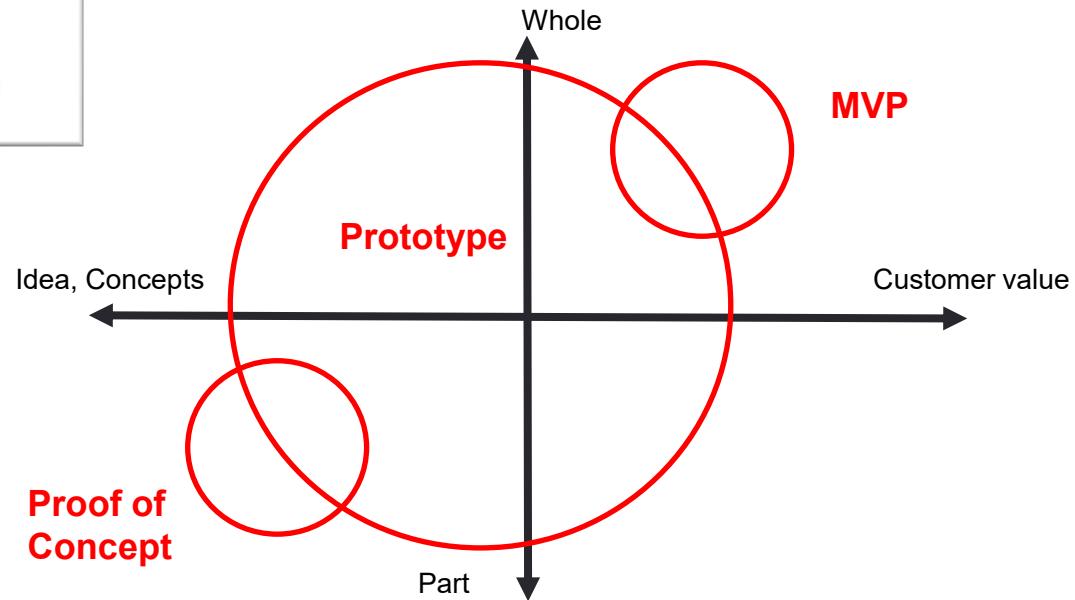
- Two logics of making decisions in startup projects:
  - Will they apply to decisions at technical and product-level as well?





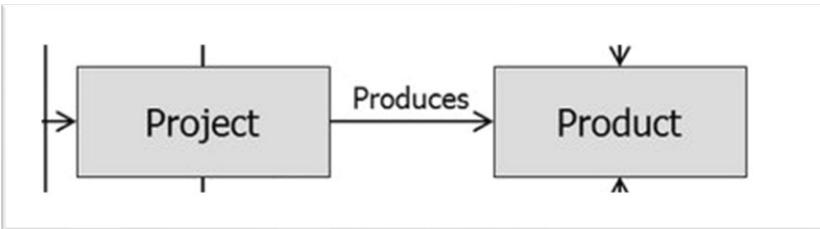
## ■ What - Minimum Viable Product

- ... as a prototype
- ... as a communication tool
- .... as a learning mechanism
- .... as documentation approach

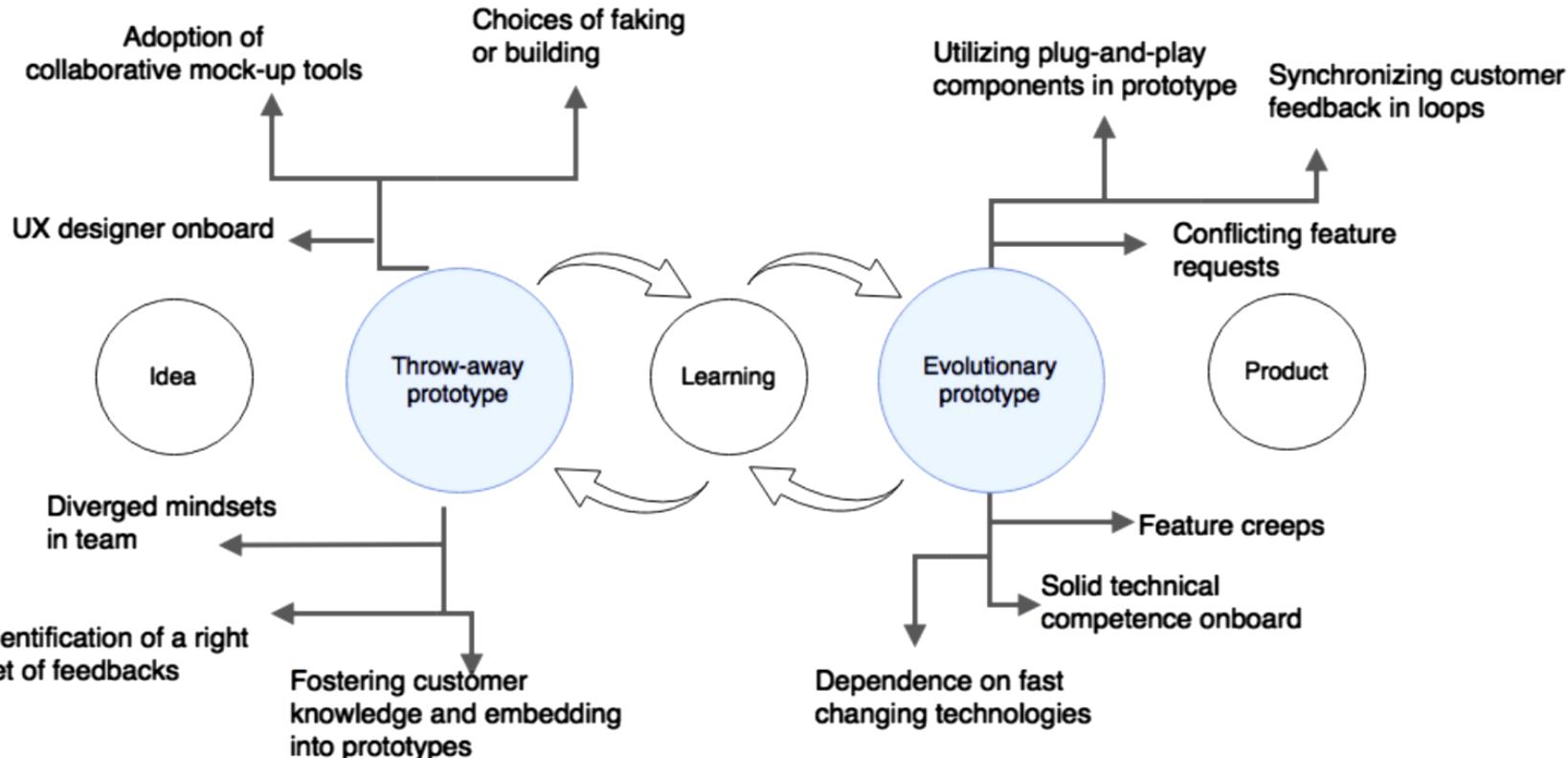


V. Berg, J. Birkeland, A. Nguyen-Duc, I. O. Pappas, and L. Jaccheri, "Achieving agility and quality in product development – an empirical study of hardware startups," *Journal of Systems and Software*, vol. 167, p. 110599, Sep. 2020

Nguyen-Duc, A., Wang, X., & Abrahamsson, P. (2017). What Influences the Speed of Prototyping? An Empirical Investigation of Twenty Software Startups. *XP Conference*



## How - Product



# Thank you! Questions?

Anh Nguyen-Duc  
[angu@usn.no](mailto:angu@usn.no)



# Goal-Oriented Requirements Engineering: A Guided Tour

Axel van Lamsweerde

Département d'Ingénierie Informatique

Université catholique de Louvain

B-1348 Louvain-la-Neuve (Belgium)

avl@info.ucl.ac.be

## Abstract

*Goals capture, at different levels of abstraction, the various objectives the system under consideration should achieve. Goal-oriented requirements engineering is concerned with the use of goals for eliciting, elaborating, structuring, specifying, analyzing, negotiating, documenting, and modifying requirements. This area has received increasing attention over the past few years.*

*The paper reviews various research efforts undertaken along this line of research. The arguments in favor of goal orientation are first briefly discussed. The paper then compares the main approaches to goal modeling, goal specification and goal-based reasoning in the many activities of the requirements engineering process. To make the discussion more concrete, a real case study is used to suggest what a goal-oriented requirements engineering method may look like. Experience with such approaches and tool support are briefly discussed as well.*

## 1. Introduction

Goals have long been recognized to be essential components involved in the requirements engineering (RE) process. As Ross and Schoman stated in their seminal paper, “*requirements definition must say why a system is needed, based on current or foreseen conditions, which may be internal operations or an external market. It must say what system features will serve and satisfy this context. And it must say how the system is to be constructed*” [Ros77]. Many informal system development methodologies from the good old times included some form of goal-based analysis, called context analysis [Ros77], definition study [Hic74], participative analysis [Mun81], and so forth. Typically, the current system under consideration is analyzed in its organizational, operational and technical setting; problems are pointed out and opportunities are identified; high-level goals are then identified and refined to address such problems and meet the opportunities; requirements are then elaborated to meet those goals. Such natural practice has led requirements documentation standards to require a specific document section devoted to the objectives the system should meet (see, e.g., the IEEE-Std-830/1993 standards).

Surprisingly enough, goals have been largely ignored both from the literature on software modeling and specification

and from the literature on object-oriented analysis (one notable exception is [Rub92]). UML advocates sometimes confess the need for higher-level abstractions: “*In my work, I focus on user goals first, and then I come up with use cases to satisfy them; by the end of the elaboration period, I expect to have at least one set of system interaction use cases for each user goal I have identified*” [Fow97, p.45]). The prominent tendency in software modeling research has been to abstract programming constructs up to requirements level rather than propagate requirements abstractions down to programming level [Myl99].

Requirements engineering research has increasingly recognized the leading role played by goals in the RE process [Yue87, Rob89, Ber91, Dar91, Myl92, Jar93, Zav97b]. Such recognition has led to a whole stream of research on goal modeling, goal specification, and goal-based reasoning for multiple purposes, such as requirements elaboration, verification or conflict management, and under multiple forms, from informal to qualitative to formal.

The objective of this paper is to provide a brief but hopefully comprehensive review of the major efforts undertaken along this line of research. Section 2 first provides some background material on what goals are, what they are useful for, where they are coming from, and when they should be made explicit in the RE process. Section 3 discusses the major efforts in modeling goals in terms of features and links to other artefacts found in requirements models. Section 4 reviews the major techniques used for specifying goals. Section 5 on goal-based reasoning reviews how goals are used in basic activities of the RE process such as requirements elicitation, elaboration, verification, validation, explanation, and negotiation, and in particular for difficult aspects of that process such as conflict management, requirements deidealization, and alternative selection. Section 6 then suggests what a goal-oriented RE method may look like by enacting it on a real case study of a safety-critical train control system. This naturally leads to a brief review, in Section 7, of industrial projects in which the use of such methods was felt conclusive; the supporting tools used in those projects are also briefly discussed there. Section 8 just opens some fairly recent pieces of goal-based work beyond requirements engineering.

## 2. The background picture

Reviewing the current state of the art in goal-oriented RE would not make much sense without first addressing the what, why, where and when questions about this area of research.

## What are goals?

A *goal* is an objective the system under consideration should achieve. Goal formulations thus refer to intended properties to be ensured; they are optative statements as opposed to indicative ones, and bounded by the subject matter [Jac95, Zav97a].

Goals may be formulated at different levels of abstraction, ranging from high-level, strategic concerns (such as “*serve more passengers*” for a train transportation system or “*provide ubiquitous cash service*” for an ATM network system) to low-level, technical concerns (such as “*acceleration command delivered on time*” for a train transportation system or “*card kept after 3 wrong password entries*” for an ATM system).

Goals also cover different types of concerns: functional concerns associated with the services to be provided, and non-functional concerns associated with quality of service --such as safety, security, accuracy, performance, and so forth.

The *system* which a goal refers to may be the current one or the system-to-be; both of them are involved in the RE process. High-level goals often refer to both systems. The system-to-be is in essence composite; it comprises both the software and its environment, and is made of active components such as humans, devices and software. As opposed to passive ones, active components have choice of behavior [Fea87, Yue87, Fic92]; henceforth we will call them *agents*. Unlike requirements, a goal may in general require the cooperation of a hybrid combination of multiple agents to achieve it [Dar93]. In a train transportation system, for example, the high-level goal of safe transportation will typically require the cooperation of on board train controllers, the train tracking system, station computers, the communication infrastructure, passengers, and so forth. In an ATM system, the goal of providing cash to eligible users will require the cooperation of the ATM software, sensors/actuators, the customer, etc. One of the important outcomes of the RE process is the decision on what parts of the system will be automated and what parts will not. A goal under responsibility of a single agent in the software-to-be becomes a *requirement* whereas a goal under responsibility of a single agent in the environment of the software-to-be becomes an *assumption* [Lam98b, Lam98c]. Unlike requirements, assumptions cannot be enforced by the software-to-be; they will hopefully be satisfied thanks to organizational norms and regulations, physical laws, etc.

## Why are goals needed?

There are many reasons why goals are so important in the RE process.

- Achieving requirements completeness is a major RE concern.. Goals provide a precise criterion for *sufficient completeness* of a requirements specification; the specification is complete with respect to a set of goals if all the goals can be proved to be achieved from the specification and the properties known about the domain considered [Yue87].
- Avoiding irrelevant requirements is another major RE concern. Goals provide a precise criterion for requirements *pertinence*; a requirement is pertinent with respect to a set

of goals in the domain considered if its specification is used in the proof of one goal at least [Yue87].

- Explaining requirements to stakeholders is another important issue. Goals provide the rationale for requirements, in a way similar to design goals in design processes [Mos85, Lee91]. A requirement appears because of some underlying goal which provides a base for it [Ros77, Dar91, Som97]. More explicitly, a goal refinement tree provides traceability links from high-level strategic objectives to low-level technical requirements. In particular, for business application systems, goals may be used to relate the software-to-be to organizational and business contexts [Yu93].
- Goal refinement provides a natural mechanism for structuring complex requirements documents for increased readability. (This at least has been our experience in all industrial projects we have been involved in, see Section 7.)
- Requirements engineers are faced with many alternatives to be considered during the requirements elaboration process. Our extensive experience revealed that alternative goal refinements provide the right level of abstraction at which decision makers can be involved for validating choices being made or suggesting other alternatives overlooked so far. Alternative goal refinements allow alternative system proposals to be explored [Lam00c].
- Managing conflicts among multiple viewpoints is another major RE concern [Nus94]. Goals have been recognized to provide the roots for detecting conflicts among requirements and for resolving them eventually [Rob89, Lam98b].
- Separating stable from more volatile information is another important concern for managing requirements evolution. A requirement represents one particular way of achieving some specific goal; the requirement is therefore more likely to evolve, towards another way of achieving that same goal, than the goal itself. The higher level a goal is, the more stable it will be. Others have made that same observation [Ant94]. It turns out that different system versions often share a common set of high-level goals; the current system and the system-to-be correspond to alternative refinements of common goals in the goal refinement graph, and can therefore be integrated into one single goal model (see Section 3).
- Last but not least, goals drive the identification of requirements to support them; they have been shown to be among the basic driving forces, together with scenarios, for a systematic requirements elaboration process [Dar91, Rub92, Dar93, Ant98, Dub98, Kai00, Lam00c]. We will come back to this in Sections 5 and 6.

## Where are goals coming from?

Goal identification is not necessarily an easy task [Lam95, Ant98, Hau98, Rol98]. Sometimes they are explicitly stated by stakeholders or in preliminary material available to requirements engineers. Most often they are implicit so that goal elicitation has to be undertaken.

The preliminary analysis of the current system is an important source for goal identification. Such analysis usually results in a list of problems and deficiencies that can be for-

mulated precisely. Negating those formulations yields a first list of goals to be achieved by the system-to-be.

In our experience, goals can also be identified systematically by searching for intentional keywords in the preliminary documents provided, interview transcripts, etc. [Lam00c].

Once a preliminary set of goals and requirements is obtained and validated with stakeholders, many other goals can be identified by *refinement* and by *abstraction*, just by asking HOW and WHY questions about the goals/requirements already available, respectively [Lam95, Lam00c].

More sophisticated techniques for goal refinement and abstraction (notably, from scenarios) will be reviewed in Section 5. Other goals are identified by resolving conflicts among goals or obstacles to goal achievement, see Section 5 too.

A common misunderstanding about goal-oriented approaches is that they are inherently top-down; this is by no means the case as it should hopefully be clear now from the discussion above.

### When should goals be made explicit?

It is generally argued that goal models are built during the early phases of the RE process [Dar93, Yu97, Dub98]. The basis for the argument is the driving role played by goals in that process; the soonest a goal is identified and validated, the best. This does not imply any sort of waterfall-like requirements elaboration process, however. As requirements "implement" goals much the same way as programs implement design specifications, there is some inevitable intertwining of goal identification and requirements elaboration [Lam95, Swa82]. Goals may thus sometimes be identified fairly lately in the RE process --especially when WHY questions about technical details or scenarios, initially taken for granted, are raised lately in the process.

## 3. Modeling goals

The benefit of goal modeling is to support heuristic, qualitative or formal reasoning schemes during requirements engineering (see Section 5). Goals are generally modelled by *intrinsic features* such as their type and attributes, and by their *links* to other goals and to other elements of a requirements model.

**Goal types and taxonomies.** Goals can be of different types. Several classification axes have been proposed in the literature.

*Functional* goals underlie services that the system is expected to deliver whereas *non-functional* goals refer to expected system qualities such as security, safety, performance, usability, flexibility, customizability, interoperability, and so forth [Kel90]. This typology is overly general and can be specialized. For example, *satisfaction* goals are functional goals concerned with satisfying agent requests; *information* goals are functional goals concerned with keeping such agents informed about object states [Dar93]. Non-functional goals can be specialized in a similar way. For example, *accuracy* goals are non-functional goals requiring the state of software objects to accurately reflect the state of the corresponding monitored/controlled objects in the environment

[MyI92, Dar93] --such goals are often overlooked in the RE process; their violation may be responsible for major failures [Lam00a]. *Performance* goals are specialized into time and space performance goals, the former being specialized into *response time* and *throughput* goals [Nix93]. *Security* goals are specialized into *confidentiality*, *integrity* and *availability* goals [Amo94]; the latter can be specialized in turn until reaching domain-specific security goals. A rich taxonomy for non-functional goals can be found in [Chu00].

Another distinction often made in the literature is between *soft goals*, whose satisfaction cannot be established in a clear-cut sense [MyI92], and (hard) goals whose satisfaction can be established through verification techniques [Dar93, Dar96]. Soft goals are especially useful for comparing alternative goal refinements and choosing one that contributes the "best" to them, see below.

Another classification axis is based on types of temporal behaviour prescribed by the goal. [Dar93]. *Achieve* (resp. *cease*) goals generate system behaviours, in that they require some target property to be eventually satisfied in some future state (resp. denied); *Maintain* (resp. *avoid*) goals restrict behaviours, in that they require some target property to be permanently satisfied in every future state (resp. denied) unless some other property holds. *Optimize* goals compare behaviours to favor those which better ensure some soft target property.

In a similar vein, [Sut93] proposes a classification according to desired system states (e.g., positive, negative, alternative, feedback, or exception-repair) and to goal level (e.g., policy level, functional level, domain level). [Ant94] makes a distinction between objective goals, that refer to objects in the system, and adverbial goals, that refer to ways of achieving objective goals.

Goal types and taxonomies are used to define heuristics for goal acquisition, goal refinement, requirements derivation, and semi-formal consistency/completeness checking [Dar93, Sut93, Ant98, Chu00, Ant01], or to retrieve goal specifications in the context of specification reuse [Mas97].

**Goal attributes.** Beside their type, goals can also be intrinsically characterized by attributes such as their *name* and their *specification* (see Section 4). *Priority* is another important attribute that can be attached to goals [Dar93]. Qualitative values for this attribute allow mandatory or optional goals to be modelled with various degrees of optionality. Priorities are often used for resolving conflicts among goals [Rob89, Lam98b]. Other goal attributes that have been proposed include goal *utility* and *feasibility* [Rob89].

**Goal Links.** Many different types of links have been introduced in the literature to relate goals (a) with each other and (b) with other elements of requirements models. Such links form the basis for defining *goal structures*. We discuss inter-goal links first, and then links between goals and other elements of requirements models such as agents, scenarios, or operations.

Links between goals are aimed at capturing situations where goals positively or negatively *support* other goals. Directly borrowed from problem reduction methods in Artificial

Intelligence [Nil71], AND/OR graphs may be used to capture goal *refinement* links [Dar91, Dar93]. *AND-refinement* links relate a goal to a set of subgoals (called refinement); this means that satisfying all subgoals in the refinement is sufficient for satisfying the parent goal. *OR-refinement* links relate a goal to an alternative set of refinements; this means that satisfying one of the refinements is sufficient for satisfying the parent goal. In this framework, a *conflict* link between two goals is introduced when the satisfaction of one of them may prevent the other from being satisfied. Those link types are used to capture alternative goal refinements and potential conflicts, and to prove the correctness of goal refinements (see Section 5).

Weaker versions of those link types have been introduced to relate soft goals [Rob89, Myl92, Chu00] as the latter can rarely be said to be satisfied in a clear-cut sense. Instead of goal satisfaction, goal *satisficing* is introduced to express that subgoals are expected to achieve the parent goal within acceptable limits, rather than absolutely. A subgoal is then said to *contribute* partially to the parent goal, regardless of other subgoals; it may contribute *positively* or *negatively*. The semantic rules are now as follows. If a goal is AND-decomposed into subgoals and all subgoals are satisfied, then the parent goal is satisficeable; but if a subgoal is denied then the parent goal is deniable. If a goal contributes negatively to another goal and the former is satisfied, then the latter is deniable. These rules are used for qualitative reasoning about goal satisficing (see Section 5).

Beside inter-goal links, goals are in general also linked to other elements of requirements models. KAOS introduces AND/OR *operationalization* links to relate goals to the operations which ensure them through corresponding required pre-, post-, and trigger conditions [Lam98c, Lam00c] (the older notion of operationalization [Dar91, Dar93] was revised and simplified from practical experience). Others have used similar links between goals and operations, e.g., [Ant94, Ant98, Kai00]. In [Myl92], the inter-goal *contribution* link types are extended to capture the positive/negative contribution of requirements to goals; *argumentation* links are also introduced to connect supporting arguments to contribution links.

There has been a massive amount of work on linking goals and scenarios together --e.g., [Fic92, Dar93, Pot95, Lei97, Sut98, Ant98, Hau98, Lam98b, Rol98, Kai00, Ant01]. The obvious reason is that scenarios and goals have complementary characteristics; the former are concrete, narrative, procedural, and leave intended properties implicit; the latter are abstract, declarative, and make intended properties explicit. Scenarios and goals thus complement each other nicely for requirements elicitation and validation. By and large the link between a goal and a scenario is a *coverage link*; the main differences between the various modeling proposals lie in the fact that a scenario may be type-level or instance-level, may be an example or a counter-example of desired behavior, and may exercise a goal or an obstacle to goal achievement.

Goal models may also be related to object models as goal formulations refer to specific objects, e.g., entities, relationships or agents [Dar93]. This link type allows pertinent

object models to be systematically derived from goal models [Lam00c].

Various proposals have also been made to relate goals to agents. In KAOS, *responsibility* links are introduced to relate the goal and agent submodels. A goal may be assigned to alternative agents through OR responsibility links; this allows alternative boundaries to be explored between the software-to-be and its environment. “Responsibility” means that the agent is committed to restrict its behavior by performing the operations it is assigned to only under restricted conditions, namely, those prescribed by the required pre-, post-, and trigger conditions [Dar93]. This notion of responsibility derives from [Fea87, Fin87]; it is studied in depth in [Let01]. *Wish* links are also sometimes used in heuristics for agent assignment [Dar91]; e.g., one should avoid assigning a goal to an agent wishing other goals in conflict with that goal..

In the i\* framework [Yu93, Yu97], various types of agent dependency links are defined to model situations where an agent depends on another for a goal to be achieved, a task to be achieved, or a resource to become available. For each type of dependency an operator is defined; operators may be combined to define plans that agents may use to achieve goals. The purpose of this modelling is to support various kinds of checks such as the viability of an agent's plan or the fulfillment of a commitment between agents. Although initially conceived for modeling the organizational environment of the software-to-be, the TROPOS project is currently aiming at propagating this framework to later stages of the software lifecycle, notably, for modeling agent-oriented software architectures.

Various authors have also suggested representing the links between goals and organizational policies, e.g., [Sib93, Fea93, Sut93].

At the process level, it may be useful for traceability purpose [Got95] to record which actor owns which goal or some view of it [Lam98b].

## 4. Specifying goals

Goals must obviously be specified precisely to support requirements elaboration, verification/validation, conflict management, negotiation, explanation and evolution.

An informal (but precise) specification should always be given to make it precise what the goal name designates [Zav97a].

Semi-formal specifications generally *declare* goals in terms of their type, attribute, and links (see Section 3). Such declarations may in general be provided alternatively using a textual or a graphical syntax (see, e.g., [Dar98]). In the NFR framework [Myl92], a goal is specified by the most specific subtype it is an instance of, parameters that denote the object attributes it refers to, and the degree of satisficing/denial by child goals. Semi-formal specifications often include keyword verbs with some predefined semantics. For example, Achieve, Maintain and Avoid verbs in KAOS specify a temporal logic pattern for the goal name appearing as parameter [Dar93]; they implicitly specify that a corresponding target condition should hold some time in the future, always in the

future unless some other condition holds, or never in the future. The intent is to provide a lightweight alternative to full formalization of the goal formulation, still amenable to some form of analysis. This basic set has been extended with qualitative verbs such as Improve, Increase, Reduce, Make, and so forth [Ant98]. In a similar spirit, goals in [Rol98] are represented by verbs with different parameters playing different roles with respect to the verb --e.g., target entities affected by the goal, beneficiary agents of the goal achievement, resource entities needed for goal achievement, source or destination of a communication goal, etc.

Formal specifications *assert* the goal formulation in a fully formal system amenable to analysis. In KAOS, such assertions are written in a real-time linear temporal logic heavily inspired from [Man92, Koy92] with the usual operators over past and future states, bound by time variables; semantically, they capture maximal sets of desired behaviors [Dar93, Let01]. The KAOS language is “2-button” in that the formal assertion layer is optional; it is used typically for critical aspects of the system only.

More formal specifications yield more powerful reasoning schemes at the price of higher specification effort and lower usability by non-experts; the various techniques briefly reviewed here should thus be seen as complementary means rather than alternative ones; their suitability may heavily depend on the specific type of system being considered.

## 5. Reasoning about goals

The ultimate purpose of goal modelling and specification is to support some form of goal-based reasoning for RE subprocesses such as requirements elaboration, consistency and completeness checking, alternative selection, evolution management, and so forth.

### 5.1 Goal verification

One of the benefits of goal-oriented RE is that one can verify that the requirements entail the goals identified, and check that the set of requirements specified is sufficiently complete for the set of goals identified [Yue87]. More precisely, if  $R$  denotes the set of requirements,  $As$  the set of environmental assumptions,  $D$  the set of domain properties, and  $G$  the set of goals, the following satisfaction relation must hold for each goal  $g$  in  $G$ :

$$R, As, D \models g \text{ with } R, As, D \not\models \text{false}$$

This may be checked informally, or formally if the goal specifications and domain properties are formalized. For temporal logic specifications one may rely on the proof theory of temporal logic and use tools such as, e.g., STeP [Man96].

A lightweight alternative is to use formal refinement patterns for Achieve, Maintain and Avoid goals [Dar96]. Such patterns are proved correct and complete once for all; refinements in the goal graph are then verified by matching them to one applicable pattern from the library. The mathematical proof intricacies are thereby hidden. A frequently used pattern is the decomposition-by-milestone pattern that refines a parent Achieve goal

$$P \Rightarrow \diamond Q$$

into two subgoals:

$$P \Rightarrow \diamond R, R \Rightarrow \diamond Q$$

where the “ $\diamond$ ” temporal operator means “sometime in the future”. Another frequently used pattern is the decomposition-by-case pattern that refines the same parent Achieve goal into three subgoals:

$$P \wedge R \Rightarrow \diamond Q, P \Rightarrow \diamond R, P \Rightarrow P W Q$$

where the “ $W$ ” temporal operator means “always in the future unless”.

The techniques above can be used for goals that can be said to be established in a clear-cut sense. For soft goals, the qualitative reasoning procedure provided by the NFR framework is particularly appropriate [Myl92]. This procedure determines the degree to which a goal is satisfied/denied by lower-level goals/requirements. A node or link in the goal graph is labelled  $S$  (satisfied) if it is satisficeable and not deniable;  $D$  (denied) if it is deniable but not satisficeable;  $C$  (conflicting) if it is both satisficeable and deniable; and  $U$  (undetermined) if it is neither satisficeable nor deniable. The general idea is to propagate such labels along satisfied links bottom-up, from lower-level nodes (i.e. requirements) to higher-level nodes (i.e. goals). Additional label values can be assigned at intermediate stages of the procedure, namely,  $U^+$  (inconclusive positive support),  $U^-$  inconclusive negative support, and ? (requiring user intervention to specify an appropriate label value). Rules for bottom-up propagation of labels are then defined accordingly. An example of application of this framework to performance goals can be found in [Nix93].

### 5.2 Goal validation

Goals can be validated by identifying or generating scenarios that are covered by them [Hau98]. One may even think of enacting such scenarios to produce animations [Hey98]. The scenario identification process is generally based on heuristics [Sut98, Ant98].

In [And89], plan-based techniques are used to tentatively generate scenarios showing that a goal can be achieved without reaching prohibited conditions. Goals, prohibited conditions and operations are specified formally by simple state predicates. An automated planner first produces a trial scenario to achieve the goal condition; it then checks for faults in the proposed scenario by looking for scenarios achieving the prohibited conditions; finally it assists the specifier in modifying the set of operations in case faults are found. [Fic92] explores this deficiency-driven paradigm further. The system is specified by a set of goals, formalized in some restricted temporal logic, a set of scenarios, expressed in a Petri net-like language, and a set of agents producing restricted scenarios to achieve the goals they are assigned to. The general approach consists of (a) trying to detect inconsistencies between scenarios and goals, and (b) applying operators that modify the specification to remove the inconsistencies. Step (a) is carried out by a planner that searches for behaviours leading to some goal violation. The operators offered to the analyst in Step (b) encode heuristics for specification debugging --e.g., introduce an agent whose responsibility is to prevent the state transitions that are the last step

in breaking the goal. There are operators for introducing new types of agents with appropriate responsibilities, splitting existing types, introducing communication and synchronization protocols between agents, weakening idealized goals, etc. The repeated application of deficiency detection and debugging operators allows the analyst to explore the design space and hopefully converge towards a satisfactory specification.

### 5.3 Goal-based requirements elaboration

The technique just sketched above is a first step towards making verification/validation contribute to the requirements elaboration process. The main reason for goal-oriented RE after all is to let goals help elaborating the requirements supporting them. A goal-based elaboration typically consists of a hybrid of top-down and bottom-up processes, plus additional processes driven by the handling of possible abnormal agent behaviors, the management of conflicting goals, the recognition of analogical situations from which specifications can be transposed, and so forth. Note, however, that for explanatory purpose the resulting requirements document is in general better presented in a top-down way.

#### Goal/requirement elicitation by refinement

An obvious (but effective) informal technique for finding out subgoals and requirements is to keep asking HOW questions about the goals already identified [Lam95, Lam00c].

Formal goal refinement patterns may also prove effective when goal specifications are formalized; typically, they help finding out subgoals that were overlooked but are needed to establish the parent goal. Consider a simple train control system, for example, and the functional goal of train progress through consecutive blocks:

##### **Goal** Achieve [TrainProgress]

**FormalDef**  $(\forall \text{tr: Train}, b: \text{Block}) [\text{On}(\text{tr}, b) \Rightarrow \diamond \text{On}(\text{tr}, b+1)]$

A particular case that comes directly to mind is when block  $b+1$ 's signal is set to 'go'. Two subgoals coming naturally to mind are the following:

##### **Goal** Achieve [ProgressWhenGoSignal]

**FormalDef**  $\forall \text{tr: Train}, b: \text{Block}$

$\text{On}(\text{tr}, b) \wedge \text{Go}[b+1] \Rightarrow \diamond \text{On}(\text{tr}, b+1)$

##### **Goal** Achieve [SignalSetToGo]

**FormalDef**  $\forall \text{tr: Train}, b: \text{Block}$

$\text{On}(\text{tr}, b) \Rightarrow \diamond \text{Go}[b+1]$

This tentative refinement matches the decomposition-by-case pattern in Section 5.1 and therefore allows the following missing subgoal to be pointed out:

##### **Goal** Maintain [TrainWaiting]

**FormalDef**  $\forall \text{tr: Train}, b: \text{Block}$

$\text{On}(\text{tr}, b) \Rightarrow \text{On}(\text{tr}, b) W \text{On}(\text{tr}, b+1)$

Another effective way of driving the refinement process is based on the determination that an agent candidate to goal assignment cannot realize the goal, e.g., because it cannot monitor the variables appearing in the goal antecedent or control the variables appearing in the goal consequent. [Let01] gives a set of conditions for goal unrealizability; this set is proved complete and provides the basis for a rich, systematic set of agent-driven refinements tactics for generating realizable subgoals and auxiliary agents.

#### Goal/requirement elicitation by abstraction

An obvious (but effective) informal technique for finding out more abstract, parent goals is to keep asking WHY questions about operational descriptions already available [Lam95, Lam00c].

More sophisticated techniques have been devised to elicit goals from scenarios. Based on a bidirectional coupling between type-level scenarios and goal verb templates as discussed in Section 4, [Rol98] proposes heuristic rules for finding out alternative goals covering a scenario (corresponding to alternative values for the verb parameters), missing companion goals, or subgoals of the goal under consideration. On a more formal side, [Lam98c] describes an inductive learning technique that takes scenarios as examples and counterexamples of intended behavior and generates goal specifications in temporal logic that cover all the positive scenarios and exclude all the negative ones.

Note also that refinement patterns when applied in the reverse way correspond to abstraction patterns that may produce more coarse-grained goals.

#### Goal operationalization

A few efforts have been made to support the process of deriving pre-, post-, and trigger conditions on software operations so as to ensure the terminal goals in the refinement process. The principle is to apply derivation rules whose premise match the goal under consideration [Dar93, Let01]. Consider, for example, the following goal:

##### **Goal** Maintain [DoorsClosedWhileMoving]

**FormalDef**  $\forall \text{tr: Train}, \text{loc}, \text{loc}': \text{Location}$

$\text{At}(\text{tr}, \text{loc}) \wedge \circ \text{At}(\text{tr}, \text{loc}') \wedge \text{loc} \leftrightarrow \text{loc}'$   
 $\Rightarrow \text{tr.Doors} = \text{'closed'} \wedge \circ (\text{tr.Doors} = \text{'closed'})$

where the " $\circ$ " temporal operator means "in the next state". Applying the following derivation rule

$G: P \wedge (P_1 \wedge \circ P_2 \Rightarrow Q_1 \wedge \circ Q_2), \text{DomPre}: P_1, \text{DomPost}: P_2$

---

*ReqPre* for  $G: Q_1$ , *ReqPost* for  $G: Q_2$

we derive the following operationalization:

##### **Operation** Move

**Input**  $\text{tr: Train}; \text{loc}, \text{loc}': \text{Location}$ ; **Output**  $\text{At}$

$\text{DomPre} \text{At}(\text{tr}, \text{loc}) \wedge \text{loc} \leftrightarrow \text{loc}'$

$\text{DomPost} \text{At}(\text{tr}, \text{loc}')$

**RequiredPre** for DoorsClosedWhileMoving:  $\text{tr.Doors} = \text{'closed'}$

**RequiredPost** for DoorsClosedWhileMoving:  $\text{tr.Doors} = \text{'closed'}$

#### Analogical reuse

Goal-based specifications can also be acquired by retrieving structurally and semantically analog specifications in a repository of reusable specification components, and then transposing the specifications found according to the structural and semantic matching revealed by the retrieval process [Mas97].

#### Obstacle-driven elaboration

First-sketch specifications of goals, requirements and assumptions are often too ideal; they are likely to be violated from time to time in the running system due to unexpected behaviors of agents. The lack of anticipation of exceptional behaviors may result in unrealistic, unachievable and/or incomplete requirements.

Such exceptional behaviors are captured by assertions called *obstacles* to goal satisfaction. An obstacle  $O$  is said to obstruct a goal  $G$  in a domain  $Dom$  iff

$$\begin{array}{ll} \{O, Dom\} \models \neg G & \text{obstruction} \\ Dom \models \neg O & \text{domain consistency} \end{array}$$

Obstacles thus need to be identified and resolved at RE time in order to produce robust requirements and hence more reliable software. The notion of obstacle was just mentioned in [Yue87]. It was elaborated further in [Pot95] where scenarios are shown to be a good vehicle for identifying goal obstructions. Some heuristics for identifying obstacles can be found in [Pot95] and [Ant98]. More formal techniques are described in [Lam98a] and then [Lam00a] for:

- the abductive generation of obstacles from goal specifications and domain properties,
- the systematic generation of various types of obstacle resolution, e.g., goal substitution, agent substitution, goal weakening, goal restoration, obstacle mitigation, or obstacle prevention.

Obstacles can also be resolved at run time in some cases, see [Fea98].

#### 5.4 Conflict management

Requirements engineers live in a world where conflicts are the rule, not the exception [Eas94]. Conflicts generally arise from multiple viewpoints and concerns [Nus94]. They must be detected and eventually resolved even though they may be temporarily useful for eliciting further information [Hun98]. Various forms of conflict are studied in [Lam88b], in particular, a weak form called *divergence* which occurs frequently in practice.

The goals  $G_1, \dots, G_n$  are said to be *divergent* iff there exists a non-trivial *boundary condition*  $B$  such that :

$$\begin{array}{ll} \{B, \forall_i G_i, Dom\} \models \text{false} & \text{inconsistency} \\ \{B, \forall_{j \neq i} G_j, Dom\} \models \text{false} & \text{minimality} \end{array}$$

(“Non-trivial” means that  $B$  is different from the bottom **false** and the complement  $\neg \forall_i G_i$ ). Note that the traditional case of conflict, in the sense of logical inconsistency, amounts to a particular case of divergence.

Divergences need to be identified and resolved at RE time in order to eventually produce consistent requirements. Formal and heuristic techniques are described in [Lam98b] for:

- the abductive generation of boundary conditions from goal specifications and domain properties,
- the systematic generation of various types of divergence resolution.

A qualitative procedure is suggested in [Rob89] for handling conflicts. The idea is to detect them at requirements level and characterize them as differences at goal level. The user of the procedure first identifies the requirements elements that correspond to each other in the various viewpoints at hand; conflict detection is then carried out by mapping syntactic differences between the corresponding requirements elements to differences in values of variables involved in the goals supported by these elements. Conflict resolution is attempted next by appealing to compromises (e.g., through

compensations or restriction specialization), or goal substitutions. Finally, the conflict resolution at goal level is down propagated to the requirements level.

#### 5.5 Goal-based negotiation

Conflict resolution often requires negotiation. [Boe95] proposes an iterative 3-step process model for goal-based negotiation of requirements. At each iteration of a spiral model for requirements elaboration,

- (1) all stakeholders involved are identified together with their wished goals (called *win conditions*);
- (2) conflicts between these goals are captured together with their associated risks and uncertainties;
- (3) goals are reconciled through negotiation to reach a mutually agreed set of goals, constraints, and alternatives for the next iteration.

#### 5.6 Alternative selection

Which goal refinement should be selected when alternative ones are identified? Which agent assignment should be selected when alternative ones are identified? This is by and large an open problem. There are local tactics of course, such as favoring alternatives with less critical obstacles or conflicts, but a systematic approach has not emerged so far in the RE literature.

One promising direction would be to use qualitative reasoning schemes à la NFR [Myl92] to select an alternative refinement that contributes the best to the satisficing of soft goals related to cost, reliability, performance etc. Multicriteria analysis techniques could be helpful here.

### 6. A goal-oriented RE method in action

It is now time to demonstrate how some of the techniques reviewed above can fit together in a goal-oriented RE method. We come back to a case study we have already presented in [Lam00c] because it illustrates many of the issues raised here; the initial document is unbiased as it comes from an independent source involved in the development; it is publicly available [BAR99] --unlike most documents from the industrial projects we have been involved in; the system is a real, complex, real-time, safety-critical one (this allows one to suggest that goal-oriented RE is not only useful for business applications). The initial document focuses on the control of speed and acceleration of trains under responsibility of the Advanced Automatic Train Control being developed for the San Francisco Bay Area Rapid Transit (BART) system.

We follow the KAOS method [Dar93, Lam95, Lam00c] in order to incrementally elaborate four complementary sub-models: (1) the goal model, (2) the object model; (3) the agent responsibility model, leading to alternative system boundaries; (4) the operation model. The goal refinement graph is elaborated by eliciting goals from available sources and asking *why* and *how* questions (*goal elaboration step*); objects, relationships and attributes are derived from the goal specifications (*object modeling step*); agents are identified, alternative responsibility assignments are explored, and agent interfaces are derived (*responsibility assignment step*);

operations and their domain pre- and postconditions are identified from the goal specifications, and strengthened pre-/postconditions and trigger conditions are derived so as to ensure the corresponding goals (*operationalization step*). These steps are not strictly sequential as progress in one step may prompt parallel progress in the next one or backtracking to a previous one.

The presentation will be sketchy for lack of space; the interested reader may refer to [Let01] for a much greater level of details.

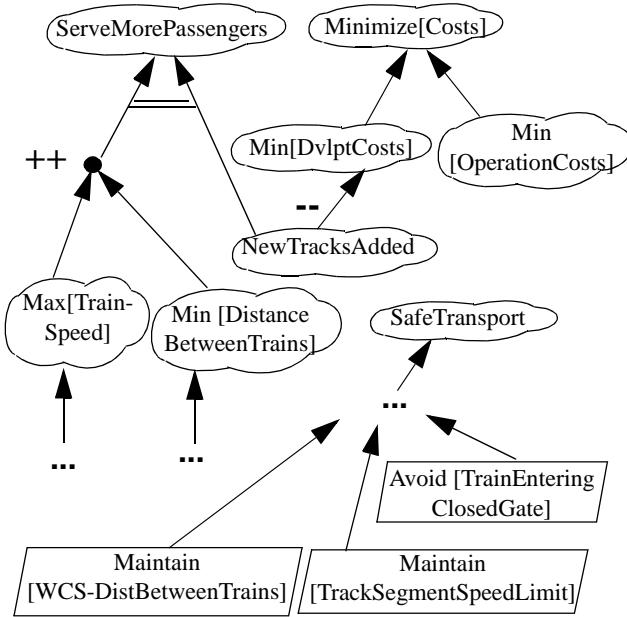


Figure 1 - Preliminary goal graph for the BART system

### Goal identification from the initial document

A first set of goals is identified from a first reading of the available source [BART99] by searching for intentional keywords such as “objective”, “purpose”, “intent”, “concern”, “in order to”, etc. A number of soft goals are thereby identified, e.g., “ServeMorePassengers”, “NewTracksAdded”, “Minimize[DevelopmentCosts]”, “Minimize[DistanceBetweenTrains]”, “SafeTransportation”, etc. These goals are qualitatively related to each other through support links: Contributes (+), ContributesStrongly (++), Conflicts (-), ConflictsStrongly (- -). These weights are used to select among alternatives. Where possible, keywords from the semi-formal layer of the KAOS language are used to indicate the goal category. The *Maintain* and *Avoid* keywords specify “always” goals having the temporal pattern  $\square(P \rightarrow Q)$  and  $\square(P \rightarrow \neg Q)$ , respectively. The *Achieve* keyword specifies “eventually” goals having the pattern  $P \Rightarrow \diamond Q$ . The “ $\rightarrow$ ” connective denotes logical implication;  $\square(P \rightarrow Q)$  is denoted by  $P \Rightarrow Q$  for short.

Figure 1 shows the result of this first elicitation. Clouds denote soft-goals, parallelograms denote formalizable goals, arrows denote goal-subgoal links, and a double line linking arrows denotes an OR-refinement into alternative subgoals.

### Formalizing goals and identifying objects

The object modeling step can start as soon as goals can be

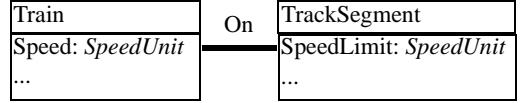
formulated precisely enough. The principle here is to identify objects, relationships and attributes from goal specifications. Consider, for example, the following goal at the bottom of Figure 1:

**Goal** `Maintain[TrackSegmentSpeedLimit]`

**InformalDef** *A train should stay below the maximum speed the track segment can handle.*

**FormalDef**  $\forall \text{tr: Train}, s: \text{TrackSegment} :$   
 $\text{On}(\text{tr}, s) \Rightarrow \text{tr.Speed} \leq s.\text{SpeedLimit}$

From the predicate, objects, and attributes appearing in this goal formalization we derive the following portion of the object model:



Similarly, the other goal at the bottom of Figure 5 is specified as follows:

**Goal** `Maintain[WCS-DistBetweenTrains]`

**InformalDef** *A train should never get so close to a train in front so that if the train in front stops suddenly (e.g., derailment) the next train would hit it.*

**FormalDef**  $\forall \text{tr1}, \text{tr2: Train} :$   
 $\text{Following}(\text{tr1}, \text{tr2}) \Rightarrow \text{tr1.Loc} - \text{tr2.Loc} > \text{tr1.WCS-Dist}$

(The InformalDef statements in those goal definitions are taken literally from the initial document; *WCS-Dist* denotes the physical worst-case stopping distance based on the physical speed of the train.) This new goal specification allows the above portion of the object model to be enriched with *Loc* and *WCS-Dist* attributes for the *Train* object together with a reflexive *Following* relationship on it. The formalization of the goal *Avoid[TrainEnterinClosedGate]* in Figure 1 will further enrich the object model by elements that are strictly necessary to the goals considered. *Goals thus provide a precise driving criterion for identifying elements of the object model.*

### Eliciting new goals through WHY questions

It is often the case that higher-level goals underpinning goals easily identified from initial sources are kept implicit in such sources. They may, however, be useful for finding out other important subgoals of the higher-level goal that were missing for the higher-level goal to be achieved.

As mentioned before, higher-level goals are identified by asking WHY questions about the goals available.

For example, asking a WHY question about the goal *Maintain[WCS-DistBetweenTrains]* yields the parent goal *Avoid[TrainCollision]*; asking a WHY question about the goal *Avoid[TrainEnteringClosedGate]* yields a new portion of the goal graph, shown in Figure 2.

In this goal subgraph, the companion subgoal *Maintain[Gate-ClosedWhenSwitchInWrongPosition]* was elicited *formally* by matching a formal refinement pattern to the formalization of the parent goal *Avoid[TrainOnSwitchInWrongPosition]*, found by a WHY question, and to the formalization of the initial goal *Avoid[TrainEnteringClosedGate]* [Dar96, Let01]. The dot joining the two lower refinement links together in Figure 2

means that the refinement is (provably) complete.

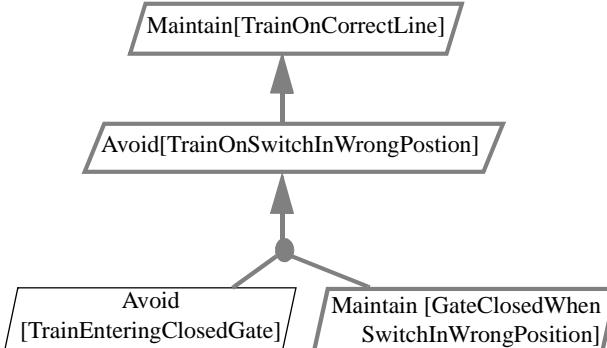


Figure 2 - Enriching the goal graph by WHY elicitation

### Eliciting new goals through HOW questions

Goals need to be refined until subgoals are reached that can be assigned to individual agents in the software-to-be and in the environment. Terminal goals become requirements in the former case and assumptions in the latter.

More concrete goals are identified by asking HOW questions. For example, a HOW question about the goal `Maintain[WCS-DistBetweenTrains]` in Figure 1 yields an extension of the goal graph shown in Figure 3.

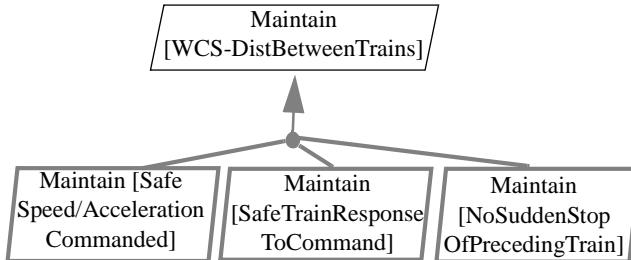


Figure 3 - Goal refinement

The formalization of the three subgoals in Figure 3 may be used to prove that together they entail the parent goal `Maintain[WCS-DistBetweenTrains]` formalized before [Let01]. These subgoals need be refined in turn until assignable subgoals are reached. A complete refinement tree is given in Annex 1.

### Identifying potential responsibility assignments

Annex 1 also provides a possible goal assignment among individual agents. This assignment seems the one suggested in the initial document [BAR99]. For example, the accuracy goal `Maintain[AccurateSpeed/PositionEstimates]` is assignable to the `TrackingSystem` agent; the goal `Maintain[SafeTrainResponseToCommand]` is assignable to the `OnBoardTrainController` agent; the goal `Maintain[SafeCmdMsg]` is assignable to the `Speed/AccelerationControlSystem` agent.

It is worth noticing that goal refinements and agent assignments are both captured by AND/OR relationships. Alternative refinements and assignments can be (and probably have been) explored. For example, the parent goal `Maintain[WCS-DistBetweenTrains]` in Figure 3 may alternatively be refined by the following three `Maintain` subgoals:

`PreceedingTrainSpeed/PositionKnownToFollowingTrain`  
`SafeAccelerationBasedOnPreceedingTrainSpeed/Position`  
`NoSuddenStopOfPreceedingTrain`

The second subgoal above could be assigned to the `OnBoardTrainController` agent. This alternative would give rise to a fully distributed system.

As suggested before, qualitative reasoning techniques in the style of [Myl99] might be applied to the softgoals identified in Figure 1 to help making choices among alternatives.

### Deriving agent interfaces

Let us now assume that the goal `Maintain[SafeCmdMsg]` at the bottom of the tree in Annex 1 has been actually assigned to the `Speed/AccelerationControlSystem` agent. The interfaces of this agent in terms of monitored and controlled variables can be derived from the formal specification of this goal (we just take its general form here for sake of simplicity):

#### Goal `Maintain[SafeCmdMsg]`

**FormalDef**  $\forall \text{cm: CommandMessage}, \text{ti1}, \text{ti2}: \text{TrainInfo}$   
 $\text{cm.Sent} \wedge \text{cm.TrainID} = \text{ti1.TrainID} \wedge \text{FollowingInfo}(\text{ti1}, \text{ti2})$   
 $\Rightarrow \text{cm.Accel} \leq F(\text{ti1}, \text{ti2}) \wedge \text{cm.Speed} > G(\text{ti1})$

To fulfil its responsibility for this goal the `Speed/AccelerationControlSystem` agent must be able to *evaluate* the goal antecedent and *establish* the goal consequent. The agent's monitored variable is therefore `TrainInfo` whereas its controlled variables are `CommandMessage.Accel` and `CommandMessage.Speed`. The latter will in turn become monitored variables of the `OnBoardTrainController` agent, by similar analysis. The technique for deriving the agent's monitored and controlled variables is fairly systematic, see [Let01] for details.

### Identifying operations

The operationalization step starts by identifying the operations relevant to goals and defining their domain pre- and postconditions. Goals refer to specific state transitions; for each such transition an operation causing it is identified; its domain pre- and postcondition capture the state transition. For the goal `Maintain[SafeCmdMsg]` formalized above we get, for example,

**Operation** `SendCommandMessage`  
**Input** `Train {arg tr}`  
**Output** `ComandMessage {res cm}`  
**DomPre**  $\neg \text{cm.Sent}$   
**DomPost**  $\text{cm.Sent} \wedge \text{cm.TrainID} = \text{tr.ID}$

This definition minimally captures what any sending of a command to a train is about in the domain considered; it does not ensure any of the goals it should contribute to.

### Operationalizing goals

The next operationalization sub-step is to strengthen such domain conditions so that the various goals linked to the operation are ensured. For goals assigned to software agents, this step produces *requirements* on the operations for the corresponding goals to be achieved. As mentioned before, derivation rules for an operationalization calculus are available [Dar93, Let01]. In our example, they yield the following requirements that strengthen the domain pre- and postconditions:

```

Operation SendCommandMessage
  Input Train {arg tr}, TrainInfo; Output CommandMsg {res cm}
  DomPre ... ; DomPost ...
ReqPost for SafeCmdMsg:
  Tracking (ti1, tr)  $\wedge$  Following (ti1, ti2)
   $\rightarrow$  cm.Acc  $\leq$  F (ti1, ti2)  $\wedge$  cm.Speed > G (ti1)
ReqTrig for CmdMsgSentInTime:
   $\blacksquare_{\leq 0.5 \text{ sec}} \neg \exists \text{cm2: CommandMessage:}$ 
  cm2.Sent  $\wedge$  cm2.TrainID = tr.ID

```

(The trigger condition captures an obligation to trigger the operation as soon as the condition gets true and provided the domain precondition is true. In the example above the condition says that no command has been sent in every past state up to one half-second [BAR99].)

Using a mix of semi-formal and formal techniques for goal-oriented requirements elaboration, we have reached the level at which most formal specification techniques would start.

### Anticipating obstacles

As mentioned before, goals also provide a basis for early generation of high-level exceptions which, if handled properly at requirements engineering time, may generate new requirements for more robust systems.

The following obstacles were generated to obstruct the subgoal Achieve[CommandMsgIssuedInTime]:

- CommandMsgNotIssued,
- CommandMsgIssuedLate,
- CommandMsgSentToWrongTrain

For the companion subgoal Achieve[CommandMsgDeliveredInTime] we similarly generated obstacles such as:

- CommandMsgDeliveredLate,
- CommandMsgCorrupted

The last companion subgoal Maintain[SafeCmdMsg] may be obstructed by the condition

UnsafeAcceleration,

and so on. The obstacle generation process for a single goal results in a goal-anchored fault-tree, that is, a refinement tree whose root is the goal negation. Compared with standard fault-tree analysis [Lev95], obstacle analysis is goal-oriented, formal, and produces obstacle trees that are provably complete with respect to what is known about the domain [Lam00a].

Alternative obstacle resolutions may then be generated to produce new or alternative requirements. For example, the obstacle CommandMsgSentLate above could be resolved by an alternative design in which accelerations are calculated by the on-board train controller instead; this would correspond to a *goal substitution* strategy. The obstacle UnsafeAcceleration above could be resolved by assigning the responsibility for the subgoal SafeAccelerationCommanded of the goal Maintain[SafeCmdMsg] to the VitalStationComputer agent instead [BART99]; this would correspond to an *agent substitution* strategy. An *obstacle mitigation* strategy could be applied to resolve the obstacle OutOfDateTrainInfo obstructing the accuracy goal Maintain[AccurateSpeed/PositionEstimates], by introducing a new subgoal of the goal Avoid[TrainCollisions], namely, the goal Avoid[CollisionWhenOutOfDateTrainInfo]. This new goal has to be refined in turn, e.g., by subgoals requiring full braking when the message origination time tag has

expired.

### Handling conflicts

The initial BART document suggests an interesting example of divergence [BART99, p.13]. Roughly speaking, the train commanded speed may not be too high, because otherwise it forces the distance between trains to be too high, in order to achieve the DistanceIncreasedWithCommandedSpeed subgoal of the SafeTransportation goal; on the other hand, the commanded speed may not be too low, in order to achieve the LimitedAccelerAbove7mphOfPhysicalSpeed subgoal of the SmoothMove goal. There seems to be a flavor of divergence here.

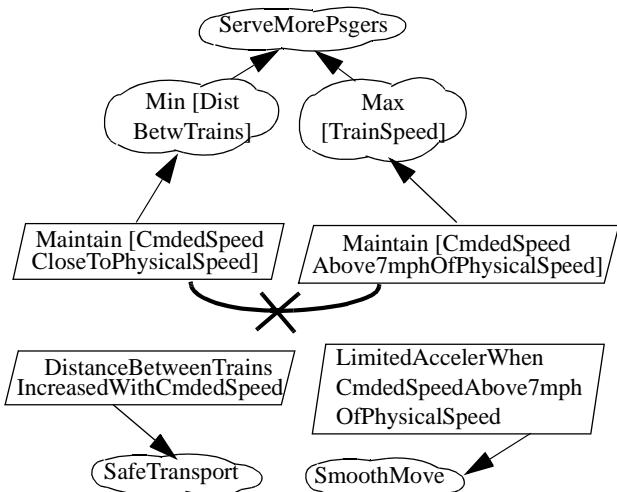


Figure 4 - Conflict in speed/acceleration control

We therefore look at the formalization of the suspect goals:

**Goal** Maintain [CmdedSpeedCloseToPhysicalSpeed]

**FormalDef**  $\forall \text{tr: Train}$

$$\begin{aligned} \text{tr.Acc}_{\text{CM}} &\geq 0 \\ \Rightarrow \text{tr.Speed}_{\text{CM}} &\leq \text{tr.Speed} + f(\text{dist-to-obstacle}) \end{aligned}$$

and

**Goal** Maintain [CmdedSpeedAbove7mphOfPhysicalSpeed]

**FormalDef**  $\forall \text{tr: Train}$

$$\text{tr.Acc}_{\text{CM}} \geq 0 \Rightarrow \text{tr.Speed}_{\text{CM}} > \text{tr.Speed} + 7$$

These two goals are formally detected to be divergent using the techniques described in [Lam98b]. The generated boundary condition for making them logically inconsistent is

$$\Diamond (\exists \text{tr: Train}) (\text{tr.Acc}_{\text{CM}} \geq 0 \wedge f(\text{dist-to-obstacle}) \leq 7)$$

The resolution operators from [Lam98b] may be used to generate possible resolutions; in this case one should keep the safety goal as it is and weaken the other conflicting goal to remove the divergence:

**Goal** Maintain [CmdedSpeedAbove7mphOfPhysicalSpeed]

**FormalDef**  $\forall \text{tr: Train}$

$$\begin{aligned} \text{tr.Acc}_{\text{CM}} \geq 0 &\Rightarrow \text{tr.Speed}_{\text{CM}} > \text{tr.Speed} + 7 \\ &\vee f(\text{dist-to-obstacle}) \leq 7 \end{aligned}$$

## 7. Experience and tool support

The purpose of this paper is obviously not to deliver an experience report. We would just like to mention here that

experience with goal-oriented requirements engineering is growing significantly, in different domain, different types of projects, and different project sizes. For example, Anton and colleagues have reported their experience with BPR applications [Ant94] and various electronic commerce systems [Ant98, Ant01]. Our understanding is that the NFR and i\* frameworks have been experienced in real settings as well.

Our KAOS method has been used in 11 industrial projects to date. These include the goal-oriented reengineering of a complex, unintelligible requirements document for a phone system on TV cable; the goal-oriented modeling of a complex air traffic control application; the goal-oriented engineering of requirements for a variety of systems such as: a copyright management system for a major editor of cartoon strips, a management system for a hospital emergency service, a drug delivery management system for a big drug distributor, a new information system for a big daily newspaper, a web-based job information server, a web-based language translation system, and various e-learning systems. To give an idea, the copyright management system has 65 goals, 75 entity types and relationships, 11 agents, and 45 operations; the goal-oriented deliverable is 115 pages long. The size of the goal refinement graph for the other applications ranges from 50 to 100 goals and requirements.

Those projects could not have been undertaken without tool support. Our current GRAIL environment provides a graphical editor tightly coupled with a syntax-directed editor, an object-oriented specification database server supporting queries for model analysis, static semantics checkers, view filtering mechanisms, a HTML generator for model browsing in hypertext mode, and various types of report generators. Current efforts are devoted to an open, full Java version; the plan then is to integrate more formal support such as animators, model checkers, test data generators, formal verification tools, and so forth.

## 8. Goal orientation beyond RE

It has been suggested recently that the functional and (especially) non-functional goals elaborated in the RE process could be used for deriving and refining architectures [Lam00c] and for annotating design patterns [Chu00]. These are just preliminary efforts that should be expanded in a near future.

## 9. Conclusion

Goal-oriented requirements engineering has many advantages, some of which were recurrently felt in the aforementioned projects, to restate a few of them:

- object models and requirements can be derived systematically from goals;
- goals provide the rationale for requirements;
- a goal graph provides vertical traceability from high-level strategic concerns to low-level technical details; it allows evolving versions of the system under consideration to be integrated as alternatives into one single framework;
- goal AND/OR graphs provide the right abstraction level at which decision makers can be involved for important decisions;

sions;

- the goal refinement structure provides a comprehensible structure for the requirements document;
- alternative goal refinements and agent assignments allow alternative system proposals to be explored;
- goal formalization allows refinements to be proved correct and complete.

We hope to have convinced the reader that this area of RE is worth pursuing. There are many open issues to work on in the future, of course; the reader may refer to [Lam00c] for a discussion of them.

**Acknowledgment.** Discussions with Robert Darimont and Emmanuel Letier were a permanent source of inspiration and confrontation of some of the issues raised in this paper; they were in particular instrumental in developing KAOS specifications for various non-trivial systems, including the one outlined here [Let01]. I am also grateful to the KAOS/GRAIL crew at CEDITI for using some of the ideas presented here in industrial projects and providing regular feedback, among others, Emmanuelle Delor, Philippe Massonet, and André Rifaut. All the people whose work is mentioned in this paper had some influence on it in some way or another (whether they recognize and like it or not!).

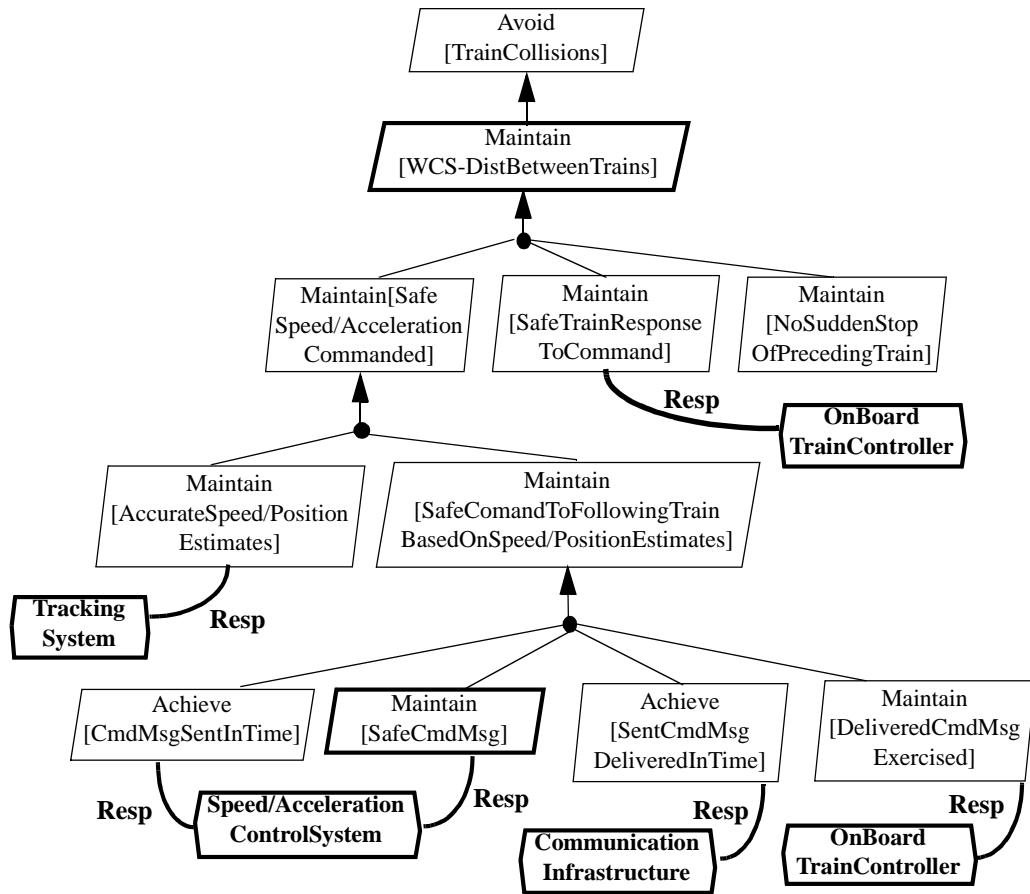
## References

- [Amo94] E.J. Amoroso, *Fundamentals of Computer Security*. Prentice-Hall, 1994.
- [And89] J.S. Anderson and S. Fickas, "A Proposed Perspective Shift: Viewing Specification Design as a Planning Problem", *Proc. 5th Intl. Workshop on Software Specification and Design*, IEEE, 1989, 177-184.
- [Ant94] A.I. Anton, W.M. McCracken, and C. Potts, "Goal Decomposition and Scenario Analysis in Business Process Reengineering", *Proc. CAiSE'94*, LNCS 811, Springer-Verlag, 1994, 94-104.
- [Ant98] A.I. Anton and C. Potts, "The Use of Goals to Surface Requirements for Evolving Systems", *Proc. ICSE-98: 20th International Conference on Software Engineering*, Kyoto, April 1998.
- [Ant01] A.I. Anton, R. Carter, A. Dagnino, J. Dempster and D.F. Siege, "Deriving Goals from a Use-Case Based Requirements Specification", *Requirements Engineering Journal*, Vol. 6, 2001, 63-73.
- [BAR99] Bay Area Rapid Transit District, Advance Automated Train Control System, Case Study Description. Sandia National Labs, <http://www.hcecs.sandia.gov/bart.htm>.
- [Ber91] V. Berzins and Luqi, *Software Engineering with Abstractions*. Addison-Wesley, 1991.
- [Boe95] B. W. Boehm, P. Bose, E. Horowitz, and Ming June Lee, "Software Requirements Negotiation and Renegotiation Aids: A Theory-W Based Spiral Approach", *Proc. ICSE-17 - 17th Intl. Conf. on Software Engineering*, Seattle, 1995, pp. 243-253.
- [Chu00] L. Chung, B. Nixon, E. Yu and J. Mylopoulos, *Non-functional requirements in software engineering*. Kluwer Academic, Boston, 2000.
- [Dar91] A. Dardenne, S. Fickas and A. van Lamsweerde, "Goal-Directed Concept Acquisition in Requirements Elicitation", *Proc. IWSSD-6 - 6th Intl. Workshop on Software Specification and Design*, Como, 1991, 14-21.
- [Dar93] A. Dardenne, A. van Lamsweerde and S. Fickas, "Goal-Directed Requirements Acquisition", *Science of Computer Programming*, Vol. 20, 1993, 3-50.
- [Dar96] R. Darimont and A. van Lamsweerde, "Formal Refinement Patterns for Goal-Driven Requirements Elaboration", *Proc. FSE'4 - Fourth ACM SIGSOFT Symp. on the Foundations of Software Engineering*, San Francisco, October 1996, 179-190.

- [Dar98] R. Darimont, E. Delor, P. Massonet, and A. van Lamsweerde, "GRAIL/KAOS: An Environment for Goal-Driven Requirements Engineering", *Proc. ICSE'98 - 20th Intl. Conf. on Software Engineering*, Kyoto, April 1998, vol. 2, 58-62. (Earlier and shorter version found in *Proc. ICSE'97 - 19th Intl. Conf. on Software Engineering*, Boston, May 1997, 612-613.)
- [Dub98] E. Dubois, E. Yu and M. Petit, "From Early to Late Formal Requirements: A Process-Control Case Study", *Proc. IWSSD'98 - 9th International Workshop on Software Specification and Design*, Isobe, IEEE CS Press, April 1998, 34-42.
- [Dwy99] M.B. Dwyer, G.S. Avrunin and J.C. Corbett, "Patterns in Property Specifications for Finite-State Verification", *Proc. ICSE-99: 21th International Conference on Software Engineering*, Los Angeles, 411-420.
- [Eas94] S. Easterbrook, "Resolving Requirements Conflicts with Computer-Supported Negotiation". In *Requirements Engineering: Social and Technical Issues*, M. Jirotka and J. Goguen (Eds.), Academic Press, 1994, 41-65.
- [Fea87] M. Feather, "Language Support for the Specification and Development of Composite Systems", *ACM Trans. on Programming Languages and Systems* 9(2), Apr. 87, 198-234.
- [Fea93] M. Feather, "Requirements Reconnoitering at the Juncture of Domain and Instance", *Proc. RE'93 - 1st Intl. IEEE Symp. on Requirements Engineering*, Jan. 1993, 73-77.
- [Fea98] M. Feather, S. Fickas, A. van Lamsweerde, and C. Ponsard, "Reconciling System Requirements and Runtime Behaviour", *Proc. IWSSD'98 - 9th International Workshop on Software Specification and Design*, Isobe, IEEE CS Press, April 1998.
- [Fic92] S. Fickas and R. Helm, "Knowledge Representation and Reasoning in the Design of Composite Systems", *IEEE Trans. on Software Engineering*, June 1992, 470-482.
- [Fin87] A. Finkelstein and C. Potts, "Building Formal Specifications Using Structured Common Sense", *Proc. IWSSD-4 - 4th International Workshop on Software Specification and Design* (Monterey, Ca.), IEEE, April 1987, 108-113.
- [Fow97] M. Fowler, *UML Distilled*. Addison-Wesley, 1997.
- [Got95] O. Gotel and A. Finkelstein, "Contribution Structures", *Proc. RE'95 - 2nd Intl. IEEE Symp. on Requirements Engineering*, York, IEEE, 1995, 100-107.
- [Gro01] D. Gross and E. Yu, "From Non-Functional Requirements to Design through Patterns", *Requirements Engineering Journal* Vol. 6, 2001, 18-36.
- [Hau98] P. Haumer, K. Pohl, and K. Weidenhaupt, "Requirements Elicitation and Validation with Real World Scenes", *IEEE Trans. on Software Engineering*, Special Issue on Scenario Management, December 1998, 1036-1054.
- [Hey98] P. Heymans and E. Dubois, "Scenario-Based Techniques for Supporting the Elaboration and the Validation of Formal Requirements", *Requirements Engineering Journal* Vol. 3 No. 3-4, 1998, 202-218.
- [Hic74] G.F. Hice, W.S. Turner, and L.F. Cashwell, *System Development Methodology*. North Holland, 1974.
- [Hun98] A. Hunter and B. Nuseibeh, "Managing Inconsistent Specifications: Reasoning, Analysis and Action", *ACM Transactions on Software Engineering and Methodology*, Vol. 7 No. 4. October 1998, 335-367.
- [Jac95] M. Jackson, *Software Requirements & Specifications - A Lexicon of Practice, Principles and Prejudices*. ACM Press, Addison-Wesley, 1995.
- [Jar93] M. Jarke and K. Pohl, "Vision-Driven Requirements Engineering", *Proc. IFIP WG8.1 Working Conference on Information System Development Process*, North Holland, 1993, 3-22.
- [Kai00] H. Kaindl, "A Design Process Based on a Model Combining Scenarios with Goals and Functions", *IEEE Trans. on Systems, Man and Cybernetic*, Vol. 30 No. 5, September 2000, 537-551.
- [Kel90] S.E. Keller, L.G. Kahn and R.B. Panara, "Specifying Software Quality Requirements with Metrics", in Tutorial: System and Software Requirements Engineering, R.H. Thayer and M. Dorfman, Eds., IEEE Computer Society Press, 1990, 145-163.
- [Koy92] R. Koymans, *Specifying message passing and time-critical systems with temporal logic*, LNCS 651, Springer-Verlag, 1992.
- [Lam95] A. van Lamsweerde, R. Darimont, and Ph. Massonet, "Goal-Directed Elaboration of Requirements for a Meeting Scheduler: Problems and Lessons Learnt", *Proc. RE'95 - 2nd Intl. IEEE Symp. on Requirements Engineering*, March 1995, 194-203.
- [Lam98a] A. van Lamsweerde and E. Letier, "Integrating Obstacles in Goal-Driven Requirements Engineering", *Proc. ICSE-98: 20th International Conference on Software Engineering*, Kyoto, April 1998.
- [Lam98b] A. van Lamsweerde, R. Darimont and E. Letier, "Managing Conflicts in Goal-Driven Requirements Engineering", *IEEE Trans. on Software Engineering*, Special Issue on Inconsistency Management in Software Development, November 1998.
- [Lam98c] A. van Lamsweerde and L. Willemet, "Inferring Declarative Requirements Specifications from Operational Scenarios", *IEEE Trans. on Software Engineering*, Special Issue on Scenario Management, December 1998, 1089-1114.
- [Lam00a] A. van Lamsweerde and E. Letier, "Handling Obstacles in Goal-Oriented Requirements Engineering", *IEEE Transactions on Software Engineering*, Special Issue on Exception Handling, 2000.
- [Lam00b] A. van Lamsweerde, "Formal Specification: a Roadmap". In *The Future of Software Engineering*, A. Finkelstein (ed.), ACM Press, 2000.
- [Lam00c] A. van Lamsweerde, "Requirements Engineering in the Year 00: A Research Perspective". Invited Keynote Paper, *Proc. ICSE'2000: 22nd International Conference on Software Engineering*, ACM Press, 2000, pp. 5-19.
- [Lee91] J. Lee, "Extending the Potts and Bruns Model for Recording Design Rationale", *Proc. ICSE-13 - 13th Intl. Conf. on Software Engineering*, IEEE-ACM, 1991, 114-125.
- [Lei97] J.C. Leite, G. Rossi, F. Balaguer, V. Maiorana, G. Kaplan, G. Hadad and A. Oliveira, "Enhancing a Requirements Baseline with Scenarios", *Requirements Engineering Journal* Vol. 2 No. 4, 1997, 184-198.
- [Let01] E. Letier, *Reasoning about Agents in Goal-Oriented Requirements Engineering*. Ph. D. Thesis, University of Louvain, May 2001.
- [Lev95] N. Leveson, *Safeware - System Safety and Computers*. Addison-Wesley, 1995.
- [Man92] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems*, Springer-Verlag, 1992.
- [Man96] Z. Manna and the STeP Group, "STeP: Deductive-Algorithmic Verification of Reactive and Real-Time Systems", *Proc. CAV'96 - 8th Intl. Conf. on Computer-Aided Verification*, LNCS 1102, Springer-Verlag, July 1996, 415-418.
- [Mas97] P. Massonet and A. van Lamsweerde, "Analogical Reuse of Requirements Frameworks", *Proc. RE-97 - 3rd Int. Symp. on Requirements Engineering*, Annapolis, 1997, 26-37.
- [Mos85] J. Mostow, "Towards Better Models of the Design Process", *AI Magazine*, Vol. 6, 1985, pp. 44-57.
- [Mun81] E. Munford, "Participative Systems Design: Structure and Method", *Systems, Objectives, Solutions*, Vol. 1, North-Holland, 1981, 5-19.
- [Myl92] Mylopoulos, J., Chung, L., Nixon, B., "Representing and Using Nonfunctional Requirements: A Process-Oriented Approach", *IEEE Trans. on Software Engineering*, Vol. 18 No. 6, June 1992, pp. 483-497.
- [Myl99] J. Mylopoulos, L. Chung and E. Yu, "From Object-Oriented to Goal-Oriented Requirements Analysis", *Communications of the ACM*, Vol. 42 No. 1, January 1999, 31-37.
- [Nil71] N.J. Nilsson, *Problem Solving Methods in Artificial Intelligence*. McGraw Hill, 1971.
- [Nix93] B. A. Nixon, "Dealing with Performance Requirements During the Development of Information Systems", *Proc. RE'93 - 1st Intl. IEEE Symp. on Requirements Engineering*, Jan. 1993, 42-49.

- [Nus94] B. Nuseibeh, J. Kramer and A. Finkelstein, "A Framework for Expressing the Relationships Between Multiple Views in Requirements Specifications", *IEEE Transactions on Software Engineering*, Vol. 20 No. 10, October 1994, 760-773.
- [Par95] D.L. Parnas and J. Madey, "Functional Documents for Computer Systems", *Science of Computer Programming*, Vol. 25, 1995, 41-61.
- [Pot94] C. Potts, K. Takahashi and A.I. Anton, "Inquiry-Based Requirements Analysis", *IEEE Software*, March 1994, 21-32.
- [Pot95] C. Potts, "Using Schematic Scenarios to Understand User Needs", *Proc. DIS'95 - ACM Symposium on Designing interactive Systems: Processes, Practices and Techniques*, University of Michigan, August 1995.
- [Rob89] Robinson, W.N., "Integrating Multiple Specifications Using Domain Goals", *Proc. IWSSD-5 - 5th Intl. Workshop on Software Specification and Design*, IEEE, 1989, 219-225.
- [Rol98] C. Rolland, C. Souveyet and C. Ben Achour, "Guiding Goal Modeling Using Scenarios", *IEEE Trans. on Software. Engineering*, Special Issue on Scenario Management, December 1998, 1055-1071.
- [Ros77] D.T. Ross and K.E. Schoman, "Structured Analysis for Requirements Definition", *IEEE Transactions on Software Engineering*, Vol. 3, No. 1, 1977, 6-15.
- [Rub92] K.S. Rubin and A. Goldberg, "Object Behavior Analysis", *Communications of the ACM* Vol. 35 No. 9, September 1992, 48-62.
- [Som97] I. Sommerville and P. Sawyer, *Requirements Engineering: A Good Practice Guide*. Wiley, 1997.
- [Sut93] A. Sutcliffe and N. Maiden, "Bridging the Requirements Gap: Policies, Goals and Domains", *Proc. IWSSD-7 - 7th Intl. Workshop on Software Specification and Design*, IEEE, 1993.
- [Sut98] A. Sutcliffe, "Scenario-Based Requirements Analysis", *Requirements Engineering Journal* Vol. 3 No. 1, 1998, 48-65.
- [Swa82] W. Swartout and R. Balzer, "On the Inevitable Intertwining of Specification and Implementation", *Communications of the ACM*, Vol. 25 No. 7, July 1982, 438-440.
- [Yue87] K. Yue, "What Does It Mean to Say that a Specification is Complete?", *Proc. IWSSD-4, Fourth International Workshop on Software Specification and Design*, Monterey, 1987.
- [Yu93] E.S.K. Yu, "Modelling Organizations for Information Systems Requirements Engineering", *Proc. RE'93 - 1st Intl Symp. on Requirements Engineering*, IEEE, 1993, 34-41.
- [Yu97] E. Yu, "Towards Modeling and Reasoning Support for Early-Phase Requirements Engineering", *Proc. RE-97 - 3rd Int. Symp. on Requirements Engineering*, Annapolis, 1997, 226-235.
- [Zav97a] P. Zave and M. Jackson, "Four Dark Corners of Requirements Engineering", *ACM Transactions on Software Engineering and Methodology*, 1997, 1-30.
- [Zav97b] P. Zave, "Classification of Research Efforts in Requirements Engineering", *ACM Computing Surveys*, Vol. 29 No. 4, 1997, 315-321.

## ANNEX 1: GOAL REFINEMENT TREE AND RESPONSIBILITY ASSIGNMENT IN THE BART SYSTEM



# Ontology-Driven Guidance for Requirements Elicitation

Stefan Farfeleder<sup>1</sup>, Thomas Moser<sup>2</sup>, Andreas Krall<sup>1</sup>, Tor Stålhane<sup>3</sup>,  
Inah Omoronyia<sup>4</sup>, and Herbert Zojer<sup>5</sup>

<sup>1</sup> Institute of Computer Languages, Vienna University of Technology  
[stefanf.andi@complang.tuwien.ac.at](mailto:stefanf.andi@complang.tuwien.ac.at)

<sup>2</sup> Christian Doppler Laboratory "Software Engineering Integration for Flexible Automation Systems", Vienna University of Technology  
[thomas.moser@tuwien.ac.at](mailto:thomas.moser@tuwien.ac.at)

<sup>3</sup> Department of Computer and Information Science,  
Norwegian University of Science and Technology  
[stalhane@idi.ntnu.no](mailto:stalhane@idi.ntnu.no)

<sup>4</sup> Irish Software Engineering Research Centre, University of Limerick  
[inah.omoronyia@lero.ie](mailto:inah.omoronyia@lero.ie)

<sup>5</sup> Infineon Technologies Austria AG  
[herbert.zojer@infineon.com](mailto:herbert.zojer@infineon.com)

**Abstract.** Requirements managers aim at keeping their sets of requirements well-defined, consistent and up to date throughout a project's life cycle. Semantic web technologies have found many valuable applications in the field of requirements engineering, with most of them focusing on requirements analysis. However the usability of results originating from such requirements analyses strongly depends on the quality of the original requirements, which often are defined using natural language expressions without meaningful structures. In this work we present the prototypic implementation of a semantic guidance system used to assist requirements engineers with capturing requirements using a semi-formal representation. The semantic guidance system uses concepts, relations and axioms of a domain ontology to provide a list of suggestions the requirements engineer can build on to define requirements. The semantic guidance system is evaluated based on a domain ontology and a set of requirements from the aerospace domain. The evaluation results show that the semantic guidance system effectively supports requirements engineers in defining well-structured requirements.

**Keywords:** requirements elicitation, domain ontology, elicitation guidance, requirements engineering.

## 1 Introduction

A major goal of requirements engineering is to achieve a common understanding between all project stakeholders regarding the set of requirements. Modern IT projects are complex due to the high number and complexity of requirements, as well as due to geographically distributed project stakeholders with different

backgrounds and terminologies. Therefore, adequate requirements management tools are a major contribution to address these challenges. Current requirements management tools typically work with a common requirements database, which can be accessed by all stakeholders to retrieve information on requirements content, state, and interdependencies.

Requirements management tools help project managers and requirements engineers to keep the overview on large amounts of requirements by supporting: (a) Requirements categorization by clustering requirements into user-defined subsets to help users find relevant requirements more quickly, e.g., by sorting and filtering attribute values; (b) Requirements conflict analysis (or consistency checking) by analyzing requirements from different stakeholders for symptoms of inconsistency, e.g., contradicting requirements; and (c) Requirements tracing by identifying dependencies between requirements and artifacts to support analyses for change impact and requirements coverage. Unfortunately, requirements management suffers from the following challenges and limitations:

- Incompleteness [5] of requirements categorization and conflict identification, in particular, when performed manually.
- High human effort for requirements categorization, conflict analysis and tracing, especially with a large number of requirements [5].
- Tracing on syntactic rather than on concept level: requirements are often traced on the syntactic level by explicitly linking requirements to each other. However, requirements engineers actually want to trace concepts, i.e., link requirements based on their meaning, which can be achieved only partially by information retrieval approaches like "keyword matching" [11][12].

The use of semantic technologies seems promising for addressing these challenges: Ontologies provide the means for describing the concepts of a domain and the relationships between these concepts in a way that allows automated reasoning [18]. Automated reasoning can support tasks such as requirements categorization, requirements conflict analysis, and requirements tracing. While these are very valuable efforts, we think what is missing here is additionally having a proactive and interactive guidance system that tries to improve requirements quality while actually eliciting requirements.

In this work we present the prototypic implementation of a semantic guidance system used to assist requirements engineers with capturing requirements using a semi-formal representation. Compared to the usual flow - write requirements, analyze requirements using the domain ontology, improve requirements - our approach directly uses the domain ontology knowledge in a single step. The semantic guidance system uses concepts, relations and axioms of domain ontologies to provide a list of suggestions the requirements engineer can build on to define requirements.

We evaluate the proposed semantic guidance system based on a domain ontology and a set of requirements from the aerospace domain. The evaluation results show that the semantic guidance system supports the requirements engineer in defining well-structured requirements. The tool managed to provide useful suggestions for filling missing parts of requirements in the majority of the cases (>85%).

This work is organized in the following way: Section 2 presents related work. Section 3 motivates our research; section 4 introduces our approach to ontology-based guidance. Section 5 presents an evaluation of the tool and section 6 concludes and gives ideas about future work.

## 2 Related Work

This section summarizes related work, going from the broad field of requirements engineering to the more specific areas of elicitation guidance and finally pattern-based requirements.

### 2.1 Requirements Engineering

Requirements Engineering is a discipline that deals with understanding, documenting, communicating and implementing customers' needs. Thus, insufficient understanding and management of requirements can be seen as the biggest cause of project failure. In order to improve this situation a systematic process to handle requirements is needed [8]. The main activities of a requirements engineering process can be defined as follows [15]:

- **Requirements Elicitation.** Requirements elicitation involves technical staff working with customers to find out about the application domain, the services the system should provide and the system's operational constraints. The goal is to gather raw requirements.
- **Requirements Analysis and Negotiation.** Requirements analysis and negotiation is an activity which aims to discover problems and conflicts with the requirements and reach agreement on changes to satisfy all system stakeholders (people that are affected by the proposed system). The final goal is to reach a common understanding of the requirements between all project participants.
- **Requirements Documentation and Validation.** The defined requirements, written down in a software requirements specification, are validated against criteria like correctness, completeness, consistency, verifiability, unambiguity, traceability, etc.
- **Requirements Management.** Requirements management consists of managing changes of requirements (keeping them consistent), e.g., by ensuring requirements traceability (identification of interdependencies between requirements, other requirements, and artifacts).

These four steps can be summarized as requirements development. In addition, requirements management is a supporting discipline to control all requirements and their changes during the development life cycle and to identify and resolve inconsistencies between the requirements and the project plan and work products. One important method of requirements management is requirements tracing. Traceability can be defined as the *degree to which a relationship between*

*two or more products of the development process can be established* [1]. Gotel [7] and Watkins [21] describe why requirements tracing can help project managers in verification, cost reduction, accountability, change management, identification of conflicting requirements and consistency checking of models.

## 2.2 Elicitation Guidance

There are several approaches to guide users to specify requirements. PROPEL [3] is a tool that provides guidance to define property specifications which are expressed as finite-state automata. For the definition of a property the user is guided by a *question tree*, a hierarchical sequence of questions. There are separate scope trees for a property's behavior and its scope. Based on the answers, the tool chooses an appropriate property template. A team of medical personnel and computer scientists used the tool to formulate properties of medical guidelines.

Kitamura et al. [14] present a requirements elicitation tool that improves requirements quality by analysis. The tool analyzes natural language requirements and finds domain ontology entities occurring in the statements. According to these occurrences and their relations in the ontology, requirements are analyzed in terms of completeness, correctness, consistency and unambiguity. Suggestions are provided to the user in order to improve these metrics, e.g., if the tool finds that a domain concept is not defined in the requirements set, the suggestion would be to add a requirement about the missing concept.

REAS [6] is a tool that interactively detects imprecisions in natural language requirements. It consists of a spelling checker, a rule imposer, a lexical analyzer and a parser. Some of the rules enforced by the tool are writing short requirements, using active voice instead of passive and avoiding possessive pronouns (e.g., “it” and “its”). If the tool detects a violation of a rule, the requirements engineer is asked to improve the offending expression.

Compared to our own approach, PROPEL only deals with a rather specific kind of requirements. The other two approaches try to identify weaknesses after the requirements have been defined and do not actively propose wording while writing them.

## 2.3 Pattern-Based Requirements

Hull, Jackson and Dick [9] first used the term *boilerplate* to refer to a textual requirement template. A boilerplate consists of a sequence of attributes and fixed syntax elements. As an example, a common boilerplate is “*<system> shall <action>*”. In this boilerplate *<system>* and *<action>* are attributes and *shall* is a fixed syntax element. It is possible to combine several boilerplates by means of simple string concatenation. This allows keeping the number of required boilerplates low while at the same time having a high flexibility. During instantiation textual values are assigned to the attributes of the boilerplates; a boilerplate requirement is thus defined by its boilerplates and its attribute values. The

authors did not propose a fixed list of boilerplates<sup>1</sup> but instead envisioned a flexible language that can be adapted or enriched when necessary.

Stålthane, Omoronyia and Reichenbach [20] extended boilerplates with a domain ontology by linking attribute values to ontology concepts. They adapted the requirements analyses introduced by Kaiya [13] to boilerplate requirements and added a new analysis called opacity. The requirement language used in this work is based on their combination of boilerplates and the domain ontology.

Ibrahim et al. [10] use boilerplate requirements in their work about requirements change management. They define a mapping from boilerplate attributes to software design artifacts (e.g., classes, attributes, operations) and add traceability links between requirements and artifacts accordingly. There are several other pattern based languages similar to boilerplates, e.g., requirements based on EBNF grammars [19]. Denger et al. [4] propose natural language patterns to specify requirements in the embedded systems domain. They include a meta-model for requirement statements and one for events and reactions which they use to check the completeness of the pattern language. Compared to boilerplate requirements, their patterns seem to be a bit less generic, e.g., some of the non-functional requirements used in our evaluation would be impossible to express.

Matsuo, Ogasawara and Ohnishi [16] use controlled natural language for requirements, basically restraining the way in which simple sentences can be composed to more complex ones. They use a frame model to store information about the domain. There are three kind of frames. The noun frame classifies a noun into one of several predefined categories. The case frame classifies verbs into operations and contains the noun types which are required for the operation. Finally the function frame represents a composition of several simple operations. The authors use these frames to parse requirements specifications, to organize them according to different viewpoints and to check requirements completeness. In contrast to domain ontologies, the frame-based approach seems to be harder to understand and to adapt by non-experts.

### 3 Research Issues

There have been presented several approaches to use ontologies to analyze requirements. These approaches try to measure quality aspects like completeness, correctness and consistency on a set of requirements. In [20] there is an analysis called *opacity* that basically checks if, for two concepts occurring in a requirement, there is a relation between them in the domain ontology. A conclusion of our review of this analysis was that, rather than first writing an incorrect requirement, then analyzing and improving it, a better approach would be to actually suggest the very same domain information which is used for the opacity analysis to the requirements engineer in the first place. There are two points to this idea:

---

<sup>1</sup> J. Dick maintains a list of suggestions at  
<http://freespace.virgin.net/gbjadi/books/re/boilerplates.htm> though.

- We want a system that automatically proposes at least parts of the requirements by using information originating from a domain ontology.
- We want a system that exploits relations and axioms of the domain ontology, i.e., a system that is more powerful than just a simple dictionary. The relations and axioms of the domain ontology represent agreed upon knowledge of stakeholders; by using them we hope to improve the precision of requirements.

We believe that a tool supporting these two points will increase efficiency by speeding up the process to get a high-quality requirements specification.

A semantic guidance system implementing these two points was added to our boilerplates elicitation tool (DODT). To test our assumption, we used the tool on requirements from the domain of the *Doors Management System* (DMS). The DMS is a use case developed by EADS for experimenting with and evaluating requirements engineering and modeling techniques within the CESAR project<sup>2</sup>. The DMS controls the doors of an airplane; its main objective is to lock the doors of an airplane while it is moving (in the air or on the ground) and to unlock them when the airplane is parked on the ground. The system consists of sensors that measure the state of the doors, actuators to lock and unlock the doors and computing elements that control the entire system.

## 4 Guidance for Boilerplate Requirements

This section presents our approach for guiding the requirement engineer using information of a domain ontology.

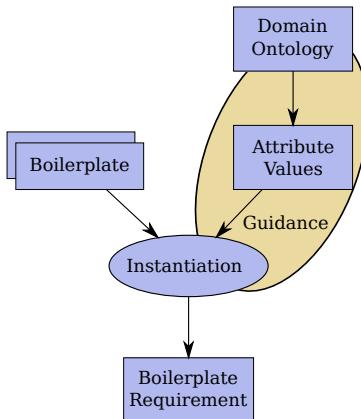
### 4.1 Boilerplate Requirements Elicitation

Figure 1 shows how requirements elicitation works using the boilerplates method. To specify a requirement, one or more boilerplates are chosen by the requirements engineer. The attribute values refer to entities in the domain ontology. During instantiation the attributes of the chosen boilerplates are set and a final boilerplate-based requirement is produced. The semantic guidance system affects the domain ontology, the attribute values and the instantiation. Its purpose is to suggest potential and suitable values for the attributes to the requirements engineer.

Table 1 lists the boilerplate attributes implemented by the tool. We reduced the number of attributes compared to the original suggestions in [9] and [20]. This was done in accordance with user wishes who were uncomfortable with the subtle differences between similar attributes and had problems deciding on which to use.

---

<sup>2</sup> <http://cesarproject.eu/>

**Fig. 1.** Boilerplate Requirements Elicitation Flow**Table 1.** Boilerplate Attributes and Values

Attribute	Description	Example Value
$\langle action \rangle$	A behavior that is expected to be fulfilled by the system, or a capability	open the door
$\langle entity \rangle$	A separate entity in the domain; not fitting into $\langle user \rangle$ or $\langle system \rangle$	door status
$\langle number \rangle$	A numeric value denoting a quantity	100
$\langle operational condition \rangle$	A condition or event that occurs during system operation	the user tries to open the door
$\langle system \rangle$	Any part of the system; sub-class of entity	door
$\langle unit \rangle$	Unit of measurement	millisecond
$\langle user \rangle$	A person somehow interacting with the system, e.g., operating it; sub-class of entity	pilot

## 4.2 Domain Ontology

The domain ontology contains facts about the domain that are relevant to requirements engineering, i.e., facts that can be used to formulate and to analyze requirements. The domain ontology should be usable to specify requirements for several projects in the same domain. Thus adding concepts which are only relevant to a single product should be avoided. See section 6 for a discussion about combining several ontologies. This approach could be used to split the ontology into a common domain part and a product-specific one.

There are three kinds of facts about the domain stored in the domain ontology. The following list describes them. The tool uses an OWL[2] representation to store ontologies. A detailed mapping to OWL can be found in Table 2.

**Table 2.** Mapping of domain facts to OWL

Fact	OWL Expressions
Concept(name, definition)	Declaration(Class( <i>concept-iri</i> )) AnnotationAssertion(rdfs:label <i>concept-iri name</i> ) AnnotationAssertion(rdfs:comment <i>concept-iri definition</i> )
Relation(subj, label, obj)	Declaration(ObjectProperty( <i>label-iri</i> )) AnnotationAssertion(rdfs:label <i>label-iri label</i> ) SubClassOf( <i>subj-iri</i> ObjectAllValuesFrom( <i>label-iri</i> <i>obj-iri</i> ))
SubClass(sub, super)	SubClassOf( <i>sub-iri</i> <i>super-iri</i> )
Equivalence(concept <sub>1</sub> , concept <sub>2</sub> )	EquivalentClasses( <i>concept<sub>1</sub>-iri</i> <i>concept<sub>2</sub>-iri</i> )
Deprecated(concept)	AnnotationAssertion(deprecated-iri <i>concept-iri 1</i> )

**Concept:** A concept represents an entity in the problem domain. The entity can be material (e.g., a physical component of the system) or immaterial (e.g., a temporal state). OWL classes are used to represent concepts. The reason for using classes instead of individuals is the built-in support for sub-classing. A concept has two attributes, its name and a textual definition. The definition is intended to provide the possibility to check whether the correct term is used.

**Relation:** A relation is a labeled directed connection between two concepts. A relation contains a label which is expected to be a verb. The label, the relation's source and destination concepts form a subject-verb-object triple. Relations are used for guidance (section 4.3). Relations map to OWL object properties and object property restrictions.

**Axiom:** There are two types of axioms that are relevant to guidance: sub-class and equivalence axioms. The first one specifies that one concept is a sub-class of another concept, e.g., *cargo door* is a sub-class of *door*. This information is used to “inherit” suggestions to sub-class concepts, e.g., the guidance system infers the suggestion *the user opens the cargo door* from the base class' *the user opens the door*.

The equivalence axiom is used to express that two concepts having different names refer to the same entity in the domain. An example from DMS is the equivalence of *aircraft* and *airplane*. Ideally each real-world phenomenon has exactly one name. However, due to requirements coming from different stakeholders or due to legacy reasons, at times several names are required. It is possible to mark a concept as being *deprecated*; the tool will warn about occurrences of such concepts and will suggest using an equivalent non-deprecated concept instead.

In this work we assume the pre-existence of a suitable domain ontology. See [17] for ways of constructing new domain ontologies. The tool contains an ontology

editor that is tailored to the information described here. We found this editor to be more user-friendly than generic OWL editors like Protégé<sup>3</sup>.

### 4.3 Guidance

When filling the attributes of a boilerplate, the tool provides a list of suggestions to the requirements engineer. The provided guidance depends on the attribute the requirements engineer is currently filling, e.g., the suggestions for *<system>* will be completely different than for *<action>*. The idea is to apply an attribute-based pre-filter to avoid overwhelming the user with the complete list of ontology entities. Typing characters further filters this list of suggestions to only those entries matching the typed string.

It is not mandatory to choose from the list of suggestions; the tool will not stop the requirements engineer from entering something completely different. In case information is missing from the domain ontology, an update of the ontology should be performed to improve the guidance for similar requirements. All changes to the domain ontology should be validated by a domain expert to ensure data correctness.

There are three types of suggestions for an attribute; Table 3 provides an overview over the suggestion types.

**Concept:** The tool suggests to use the name of a concept for an attribute.

The tool generates two variants, just the plain name and once prefixed with the article “the”. The idea is that most of the times using “the” will be appropriate but sometimes other determiners like “all” or “each” are more suitable and are typed in manually.

**Verb-Object:** The tool uses a relation from the domain ontology to suggest a verb phrase to the requirements engineer. The suggestion is the concatenation of the verb’s infinitive form<sup>4</sup>, the word “the” and the relation’s destination object. This construction is chosen in order to be grammatically correct following a modal verb like “shall”. An example from Figure 2 is the suggestion *check the door status*.

**Subject-Verb-Object:** For this kind of suggestion the entire relation including subject and object is taken into account. The suggestion text is “the”, the subject, the verb conjugated into third person singular form, “the” and the object. An example from Figure 2 is the suggestion *the person tries to open the door*.

It is possible to combine several suggestions simply by selecting the first one, manually typing in “and” and selecting another suggestion.

For the classification of concepts into different attributes a separate ontology, the *attributes ontology*, is used. The attributes ontology contains an OWL class

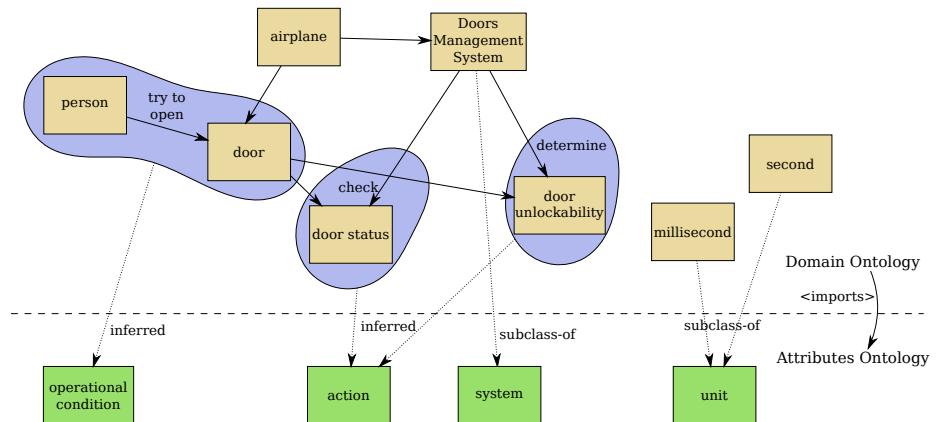
---

<sup>3</sup> <http://protege.stanford.edu/>

<sup>4</sup> For building verb infinitives the *morphological analyzer* of the GATE project (<http://gate.ac.uk/>) is used.

**Table 3.** Suggestion Types

Type	Suggestion
Concept	<i>concept</i> <i>the concept</i>
Verb-Object	<i>verb (inf.) the object</i>
Subject-Verb-Object	<i>the subject verb (3<sup>rd</sup> sing.) the object</i>

**Fig. 2.** Domain Ontology and Attributes Ontology

per attribute and the sub-class axioms mentioned in Table 1. The domain ontology imports the attributes ontology to use its classes. Domain concepts are linked to attributes by means of sub-class axioms which are stored in the domain ontology.

An example for the semantic guidance system is given in Figure 2. The domain ontology is shown in the upper part of the figure, the attributes ontology is below. The concept *Doors Management System* is a sub-class of class *system*, which in turn allows the tool to suggest using the *Doors Management System* for a boilerplate containing the attribute *<system>*. The blue regions represent verb-object and subject-verb-object suggestions in the domain ontology. Their mapping to the attributes *<action>* and *<operational condition>* is inferred automatically by the tool.

Figure 3 shows the boilerplates for two requirements and some of the suggestions provided by the guidance system. The information that *Doors Management System* is a *system* and that *second* and *millisecond* are values for attribute *unit* is stored in the domain ontology itself. The suggestions *check the door status* and *determine the door unlockability* are inferred from the domain ontology relations. The knowledge to suggest verb-object pairs for the attribute *action* is a built-in feature of the tool. The attribute *operational condition* turns out to be the most difficult one in terms of providing useful suggestions. The reason for this is that

<b>If</b> <operational condition>,	<system> <b>shall</b> <action>	<b>within</b> <number> <unit>	
the user tries to open the door	the Doors Management System	determine the door unlockability	milliseconds
...	...	...	...
<b>&lt;system&gt; <b>shall</b> &lt;action&gt;</b>	<b>at a minimum rate of</b> <number> <b>times per</b> <unit>		
the Doors Management System	check the door status		second
...	...	...	...

**Fig. 3.** Boilerplates and Suggestions

there are many grammatical ways to describe conditions, a subject-verb-object triple being only one of them. Therefore the tool does not only suggest those triples for conditions; instead all concepts, verb-object pairs and subject-verb-object triples are provided in order to use those phrases in conditions.

## 5 Evaluation

As mentioned before we evaluated the semantic guidance system with a domain ontology and a set of requirements from the Doors Management System.

### 5.1 Setting

The use case contains a set of 43 requirements specified using natural language text. Various types of requirements are included: functional, safety, performance, reliability, availability and cost. Each requirement was reformulated into a boilerplate requirement using DODT. The semantic guidance system was used to assist in filling the boilerplate attributes.

The domain ontology for DMS was specifically developed for usage with the boilerplates tool. The data for the DMS ontology was initially provided by EADS and was then completed by the authors. Table 4 lists the number of concepts, relations and axioms of the DMS ontology.

Figure 4 shows the graphical user interface of the tool. At the top of the interface boilerplates can be selected. The center shows the currently selected boilerplates and text boxes for the attribute values of the requirements. The list of phrases below the text boxes are the suggestions provided by the semantic guidance system. Typing in the text boxes filters the list of suggestions. The tool shows the textual definitions of the concepts as tooltips. Selecting a list entry will add the text to the corresponding text box. The bottom of the interface lists all requirements. Expressions that refer to entities from the domain ontology are

underlined with green lines; fixed syntax elements of boilerplates with black. If nouns missing from the domain ontology were to be seen, they would be highlighted with the color red.

Table 5 present statistics about the suggestions produced by the guidance system for the DMS ontology.

**Table 4.** Ontology Measurements

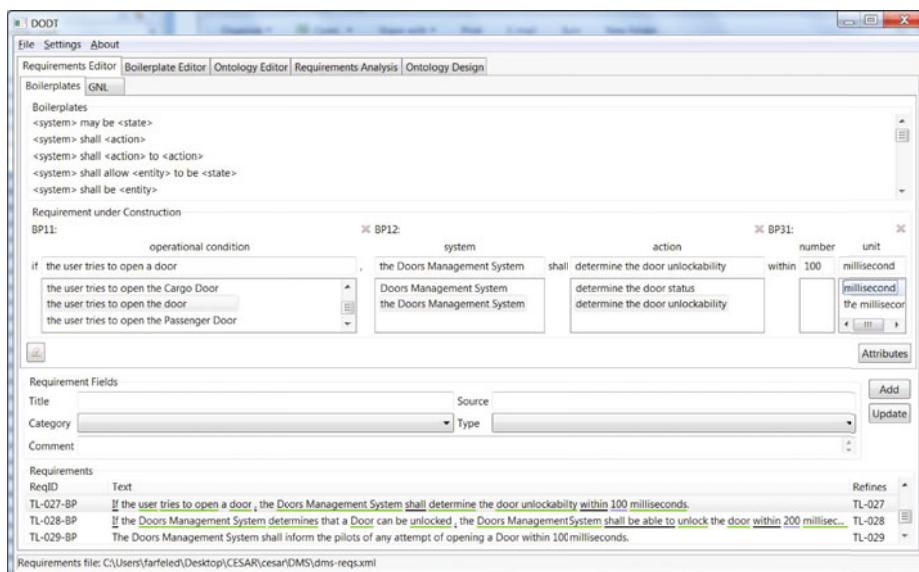
Entity	Count
Concepts	107
Relations	70
Axioms	123
SubClass	108
to Concepts	15
to Attributes	93
Equivalence	15

**Table 5.** Guidance Suggestions

Type	Count
Concept	212
Verb-Object	69
Subject-Verb-Object	101
Total	382

**Table 6.** Evaluation Results

Item	Count
Requirements	43
Boilerplates	21
Attributes	120
Complete suggestions	36
Partial suggestions	39



**Fig. 4.** DODT Screenshot

## 5.2 Results

Table 6 lists the major results of the evaluation. For 43 requirements, we used 21 different boilerplates. The boilerplate which was used most often (16 times) is *<system> shall <action>*. The 43 boilerplate requirements have a total of 120 attributes. For 36 attributes out of 120 (30%) the semantic guidance system was able to suggest the entire attribute value without any need for a manual change. For another 59 attributes (57.5%) the guidance could suggest at least parts of the attribute value. This leaves 25 attribute values (12.5%) for that the guidance was no help. For partial matches, these are some of the reasons the attribute values had to be modified:

- A different determiner is used than the suggested “the”, e.g., “a”, “each” or “all”.
- The plural is used instead of singular.
- A combination of two or more suggestions is used.
- A subordinate clause is added, e.g., “each door that could be a hazard if it unlatches”.

Reasons for no guidance are these:

- Numbers for the *<number>* attribute cannot be suggested.
- Words are used that do not exist in the domain ontology.

Future work will include setting up an evaluation to compare the elicitation time with and without the semantic guidance system. However, due to the high percentage where the guidance was able to help (>85%) we are confident that efficiency improved, even though the presentation of explicit numbers has to be postponed to future work.

We also hope to improve the quality of requirements using the tool. We did a qualitative comparison of the original DMS requirements and the boilerplate requirements. These are our findings:

- Boilerplate requirements encourage using the active voice. In our evaluation the original requirement “Information concerning the door status shall be sent from the Doors Management System to ground station...” was turned into “The Doors Management System shall send information concerning the door status to ground station...” 8 requirements were improved in this way. In some cases missing subjects were added.
- Requirements like “There shall be...” and “It shall not be possible to...” were changed into “The *subject* shall have” and “The *subject* shall not allow...”. Such changes make it obvious what part of the system is responsible to fulfill the requirement. To determine the right value for *subject* the original stakeholders should be asked for clarification. Due to timing constraints this was not possible and plausible values were inserted by the authors.
- During the requirements transformation we found that the original requirements used different expressions for seemingly identical things, e.g., “provision to prevent pressurization” and “pressure prevention means” or “airplane”

and “aircraft”. Such synonyms are either stored as an equivalence axiom in the domain ontology or, preferably, stakeholders agree upon the usage of one term.

- Using boilerplates improved the overall consistency of the requirements. The original requirements set contains a mixture of “must” and “shall”. While using one or the other is probably a matter of taste, one form should be picked and used consistently.
- The guidance system corrected a few typographic errors in the original requirements, e.g., “milliseconds”.

We found the tool to be easy to use and user-friendly. This sentiment is shared by the partners in the CESAR project who are also currently evaluating the tool.

## 6 Conclusion and Future Work

Requirements should be made as consistent, correct and complete as possible to prevent detecting and correcting errors in later design phases. With respect to the three points raised in the introduction about requirements engineering, this work intends to be the foundation for further analyses, e.g., by facilitating requirements categorization with ontology knowledge.

We presented a tool for the elicitation of boilerplate requirements that includes a semantic guidance system which suggests concept names and phrases that were built from relations and axioms of the domain ontology. The tool managed to provide useful suggestions in the majority of the cases ( $>85\%$ ). We realize that the tool needs to be evaluated in a larger context, i.e., more requirements and a larger ontology. We will address this in our future research work.

The selection of suitable boilerplates for a requirement is not always trivial and requires a bit of experience with the boilerplates method. Thus a feature we want to explore and possibly add to the tool is the semi-automatic conversion of natural language requirements into boilerplate requirements.

While creating the DMS ontology, we found there are concepts for which the suggestions provided by the semantic guidance system really help but which are not specific to the problem domain. The most prominent example are measurement units like “second” or “kg”. So what we want to do is to collect such entities into a separate ontology and to extend the tool to be able manage several ontologies. This could be handled by adding OWL import declarations to the “main” domain ontology. Such domain-independent ontologies can then be easily reused for guidance in other domains.

## Acknowledgments

The research leading to these results has received funding from the ARTEMIS Joint Undertaking under grant agreement N° 100016 and from specific national programs and/or funding authorities. This work has been supported by the Christian Doppler Forschungsgesellschaft and the BMWFJ, Austria.

## References

1. IEEE Recommended Practice for Software Requirements Specifications. IEEE Std 830 (1998)
2. OWL 2 Web Ontology Language Direct Semantics. Tech. rep., W3C (2009), <http://www.w3.org/TR/2009/REC-owl2-direct-semantics-20091027/>
3. Cobleigh, R., Avrunin, G., Clarke, L.: User Guidance for Creating Precise and Accessible Property Specifications. In: 14th International Symposium on Foundations of Software Engineering, pp. 208–218. ACM, New York (2006)
4. Denger, C., Berry, D., Kamsties, E.: Higher Quality Requirements Specifications through Natural Language Patterns. In: 2003 IEEE International Conference on Software - Science, Technology and Engineering, pp. 80–90. IEEE, Los Alamitos (2003)
5. Egyed, A., Grumbacher, P.: Identifying Requirements Conflicts and Cooperation: How Quality Attributes and Automated Traceability Can Help. IEEE Software 21(6), 50–58 (2004)
6. Elazhary, H.H.: REAS: An Interactive Semi-Automated System for Software Requirements Elicitation Assistance. IJEST 2(5), 957–961 (2010)
7. Gotel, O., Finkelstein, C.: An Analysis of the Requirements Traceability Problem. In: 1st International Conference on Requirements Engineering, pp. 94–101 (1994)
8. Gottesdiener, E.: Requirements by Collaboration: Workshops for Defining Needs. Addison-Wesley, Reading (2002)
9. Hull, E., Jackson, K., Dick, J.: Requirements Engineering. Springer, Heidelberg (2005)
10. Ibrahim, N., Kadir, W., Deris, S.: Propagating Requirement Change into Software High Level Designs towards Resilient Software Evolution. In: 16th Asia-Pacific Software Engineering Conference, pp. 347–354. IEEE, Los Alamitos (2009)
11. Jackson, J.: A Keyphrase Based Traceability Scheme. IEEE Colloquium on Tools and Techniques for Maintaining Traceability During Design, 2/1-2/4 (1991)
12. Kaindl, H.: The Missing Link in Requirements Engineering. Software Engineering Notes 18, 30–39 (1993)
13. Kaiya, H., Saeki, M.: Ontology Based Requirements Analysis: Lightweight Semantic Processing Approach. In: 5th Int. Conf. on Quality Software, pp. 223–230 (2005)
14. Kitamura, M., Hasegawa, R., Kaiya, H., Saeki, M.: A Supporting Tool for Requirements Elicitation Using a Domain Ontology. Software and Data Technologies, 128–140 (2009)
15. Kotonya, G., Sommerville, I.: Requirements Engineering. John Wiley & Sons, Chichester (1998)
16. Matsuo, Y., Ogasawara, K., Ohnishi, A.: Automatic Transformation of Organization of Software Requirements Specifications. In: 4th International Conference on Research Challenges in Information Science, pp. 269–278. IEEE, Los Alamitos (2010)
17. Omoronyia, I., Sindre, G., Stålhane, T., Biffl, S., Moser, T., Sunindyo, W.: A Domain Ontology Building Process for Guiding Requirements Elicitation. In: 16th REFSQ, pp. 188–202 (2010)
18. Pedrinaci, C., Domingue, J., Alves de Medeiros, A.K.: A Core Ontology for Business Process Analysis. In: Bechhofer, S., Hauswirth, M., Hoffmann, J., Koubarakis, M. (eds.) ESWC 2008. LNCS, vol. 5021, pp. 49–64. Springer, Heidelberg (2008)
19. Rupp, C.: Requirements-Engineering und -Management. Hanser (2002)
20. Stålhane, T., Omoronyia, I., Reichenbach, F.: Ontology-Guided Requirements and Safety Analysis. In: 6th International Conference on Safety of Industrial Automated Systems (2010)
21. Watkins, R., Neal, M.: Why and How of Requirements Tracing. IEEE Software 11(4), 104–106 (1994)

## Bidirectional Requirements Traceability

By Linda Westfall



[www.westfallteam.com](http://www.westfallteam.com)

Traceability is one of the essential activities of good requirements management. Traceability is used to ensure that the right products are being built at each phase of the software development life cycle, to trace the progress of that development and to reduce the effort required to determine the impacts of requested changes. This article explores:

- What is traceability?
- Why is traceability a good practice?
- How is traceability performed?

### What is Traceability?

The IEEE Standard Glossary of Software Engineering Terminology defines traceability as “the degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor-successor or master-subordinate relationship to one another.” [IEEE-610]

Traceability is used to track the relationship between each unique product-level requirement and its source. For example, a product requirement might trace from a business need, a user request, a business rule, an external interface specification, an industry standard or regulation, or to some other source.

Traceability is also used to track the relationship between each unique product-level requirement and the work products to which that requirement is allocated. For example, a single product requirement might trace to one or more architectural elements, detail design elements, objects/classes, code units, tests, user documentation topics, and/or even to people or manual processes that implements that requirement.

Good traceability practices allow for bidirectional traceability, meaning that the traceability chains can be traced in both the forwards and backwards directions as illustrated in Figure 1.

Forward traceability looks at:

- Tracing the requirements sources to their resulting product requirement(s) to ensure the completeness of the product requirement specification.
- Tracing each unique product requirement forward into the design that implements that requirement, the code that implements that design and the tests that validate that requirement and so on. The objective is to ensure that each requirement is implemented in the product and that each requirement is thoroughly tested.

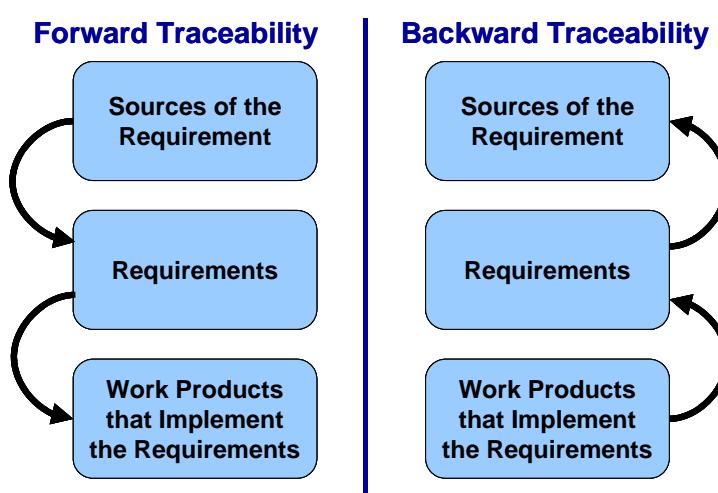


Figure 1: Bidirectional (Forward & Backward) Traceability

Backwards traceability looks at:

- Tracing each unique work product (e.g., design element, object/class, code unit, test) back to its associated requirement. Backward traceability can verify that the requirements have been kept current with the design, code, and tests.
- Tracing each requirement back to its source(s).

### **Why is Traceability a Good Practice?**

The Software Engineering Institute (SEI) Capability Maturity Model Integration® (CMMI®) states that the purpose of the process area in “Requirements Management is to manage the requirements of the project’s product and product components and to identify inconsistencies between those requirements and the project’s plans and work products.” [SEI-00]. One of the specific practices under the Requirements Management process area is to “Maintain Bidirectional Traceability of Requirements.” [SEI-00] What is the benefit of putting in the effort to maintain bidirectional traceability?

Forward traceability ensures proper direction of the evolving product (that we are building the right product) and indicates the completeness of the subsequent implementation. For example, if a business rule can’t be traced forward to one or more product requirements then the product requirements specification is incomplete and the resulting product may not meet the needs of the business. If a product requirement cannot be traced forward to its associated architectural design elements, then the architectural design is not complete and so on.

If, on the other hand, there are changes in the business environment (e.g., a business rule change or a standard change), then if good forward traceability has been maintained, that change can be traced forward to the associated requirements and all of the work products that are impacted by that change. This greatly reduces the amount of effort required to do a thorough job of impact analysis. It also reduces the risk that one of the effected work products is forgotten, resulting in an incomplete implementation of the change (i.e., a defect).

Backwards traceability helps ensure that the evolving product remains on the correct track with regards to the original and/or evolving requirements (that we are building the product right). The objective is to ensure that we are not expanding the scope of the project by adding design elements, code, tests or other work products that are not called out in the requirements (i.e., “gold plating”). If there is a change needed in the implementation or if the developers come up with a creative, new technical solution, that change or solution should be traced backwards to the requirements and the business needs to ensure that it is within the scope of the desired product. For example, if there is a work product element that doesn’t trace backwards to the product requirements one of two things is true. The first possibility is that there is a missing requirement because the work product element really is needed. In this case, traceability has helped identify the missing requirement and can also be used to evaluate the impacts of adding that requirement to project plans and other work products (forward traceability again). The second possibility is that there is “gold plating” going on – something has been added that should not be part of the product. Gold plating is a high risk activity because project plans have not allocated time or resources to the work and the existence of that part of the product may not be well communicated to other project personnel (e.g., tester doesn’t test it, it’s not included in user documentation).

Another benefit of backward traceability comes when a defect is identified in one of the work products. For example, if a piece of code has a defect, the traceability matrix can be used to help determine the root cause of that defect. For example, is it just a code defect or does it trace back to a defect in the design or requirements? If it’s a design or requirements defect, what other work products might be impacted by the defect?

Benefits of bi-directional requirements traceability include the ability to:

- Analyze the impact of a change
  - All work products affected by a changed requirement
  - All requirements affected by a change or defect in a work product
- Assess current status of the requirements and the project
  - Identify missing requirements

- Identify gold plating

### How is Traceability Performed?

The classic way to perform traceability is by constructing a traceability matrix. As illustrated in Table 1, a traceability matrix summarizes in matrix form the traceability from original identified stakeholder needs to their associated product requirements and then on to other work product elements. In order to construct a traceability matrix, each requirement, each requirements source and each work product element must have a unique identifier that can be used as a reference in the matrix. The requirement matrix has the advantage of being a single repository for documenting both forwards and backwards traceability across all of the work products.

Requirement Source	Product Requirements	HLD Section #	LLD Section #	Code Unit	UTS Case #	STS Case #	User Manual
Business Rule #1	R00120 Credit Card Types	4.1 Parse Mag Strip	4.1.1 Read Card Type	Read_Card_Type.c Read_Card_Type.h	UT 4.1.032 UT 4.1.033 UT 4.1.038 UT 4.1.043	ST 120.020 ST 120.021 ST 120.022	Section 12
			4.1.2 Verify Card Type	Ver_Card_Type.c Ver_Card_Type.h Ver_Card_Types.dat	UT 4.2.012 UT 4.2.013 UT 4.2.016 UT 4.2.031 UT 4.2.045	ST 120.035 ST 120.036 ST 120.037 ST 120.037	Section 12
Use Case #132 step 6	R00230 Read Gas Flow	7.2.2 Gas Flow Meter Interface	7.2.2 Read Gas Flow Indicator	Read_Gas_Flow.c	UT 7.2.043 UT 7.2.044	ST 230.002 ST 230.003	Section 21.1.2
	R00231 Calculate Gas Price	7.3 Calculate Gas price	7.3 Calculate Gas price	Cal_Gas_Price.c	UT 7.3.005 UT 7.3.006 UT 7.3.007	ST 231.001 ST 231.002 ST 231.003	Section 21.1.3

**Table 1: Example of a Traceability Matrix**

Modern requirements management tools include traceability mechanisms as part of their functionality.

A third mechanism for implementing traceability is trace tagging. Again, each requirement, each requirement source and each work product element must have a unique identifier. In trace tagging however, those identifiers are used as tags in the subsequent work products to identify backwards tracing to the predecessor document. As illustrated in Figure 2 for example, the Software Design Specification

**SRS** R00104 ← The system shall cancel the transaction if at any time prior to the actual dispensing of gasoline, the cardholder requests cancellation.

**SDS**

SDS Identifier	SRS Tag	Component Name	Component Description	Type	Etc
7.01.032	R00104	Cancel_Transaction	Cancel transaction when the customer presses cancel button	Module	

**UTS**

Test Case #	SDS Identifier	Test Case Name	Inputs	Expected Result	Etc
23476	7.01.032	Cancel_Before_PIN_Entry			
23477	7.01.032	Cancel_After_Invalid_PIN_Entry			
23478	7.01.032	Cancel_After_Start_Pump_Gas			

**Figure 2: Example of Trace Tagging**

(SDS) includes tags that identify the requirements implemented by each uniquely identified design element and the Unit Test Specification (UTS) includes trace tags that trace back to the design elements that each test case verifies. This tagging propagates through the work product set with source code units that include trace tags back to design elements, integration test cases with tags back to architecture elements and system test cases with trace tags back to requirements as appropriate depending on the hierarchy of work products used on the product. Trace tags have the advantage of being part of the work products so a separate matrix isn't maintained. However, while backwards tracing is easy with trace tags, forward tracing is very difficult using this mechanism.

All traceability implementation techniques require the commitment of a cross-functional team of participants to create and maintain the linkages between the requirements, their source and their allocation to subsequent work products. The requirements analyst must initiate requirements traceability and document the original tracing of the product requirements to their source. As system and software architects create the high-level design, those practitioners add their information to the traceability documentation. Developers doing low-level design, code and unit testing add additional traceability information for the elements they create, as do the integration, system and alpha, beta and acceptance testers. For small projects, some of these roles may not exist or may be done by the same practitioner, which limits the number of different people working with the traceability information. For larger projects, where traceability information comes from many different practitioners, it may be necessary to have someone who coordinates, documents and ensures periodic audits of the traceability information from all its various sources to achieve completeness and consistency.

## References

- |          |   |
|----------|---|
| IEEE-610 | IEEE Standards Software Engineering, <i>IEEE Standard Glossary of Software Engineering Terminology</i> , IEEE Std. 610-1990, The Institute of Electrical and Electronics Engineers, 1999, ISBN 0-7381-1559-2.   |
| SEI-00   | <i>CMMI<sup>SM</sup> for Systems Engineering/Software Engineering, Version 1.02 (CMMI-SW/SW, V 1.02); CMMI Staged Representation, CMU/SEI-2000-TR-018, ESC-TR-2000-018; Continuous Representation, CMU/SEI-2000-TR-019, ESC-TR-2000-019;</i> Product Development Team; Software Engineering Institute; November 2000. |

# Regression testing minimization, selection and prioritization: a survey

S. Yoo<sup>\*,†</sup> and M. Harman

*King's College London, Centre for Research on Evolution, Search and Testing, Strand, London WC2R 2LS, U.K.*

## SUMMARY

Regression testing is a testing activity that is performed to provide confidence that changes do not harm the existing behaviour of the software. Test suites tend to grow in size as software evolves, often making it too costly to execute entire test suites. A number of different approaches have been studied to maximize the value of the accrued test suite: minimization, selection and prioritization. Test suite minimization seeks to eliminate redundant test cases in order to reduce the number of tests to run. Test case selection seeks to identify the test cases that are relevant to some set of recent changes. Test case prioritization seeks to order test cases in such a way that early fault detection is maximized. This paper surveys each area of minimization, selection and prioritization technique and discusses open problems and potential directions for future research. Copyright © 2010 John Wiley & Sons, Ltd.

Received 4 September 2008; Revised 4 January 2010; Accepted 6 January 2010

KEY WORDS: regression testing; test suite minimization; regression test selection; test case prioritization

## 1. INTRODUCTION

Regression testing is performed when changes are made to existing software; the purpose of regression testing is to provide confidence that the newly introduced changes do not obstruct the behaviours of the existing, unchanged part of the software. It is a complex procedure that is all the more challenging because of some of the recent trends in software development paradigms. For example, the component-based software development method tends to result in the use of many black-box components, often adopted from a third party. Any change in the third-party components may interfere with the rest of the software system, yet it is hard to perform regression testing because the internals of the third-party components are not known to their users. The shorter life-cycle of software development, such as the one suggested by the *agile* programming discipline, also imposes restrictions and constraints on how regression testing can be performed within limited resources.

Naturally, the most straightforward approach to this problem is to simply execute all the existing test cases in the test suite; this is called a *retest-all* approach. However, as software evolves, the test suite tends to grow, which means that it may be prohibitively expensive to execute the

---

\*Correspondence to: S. Yoo, King's College London, Centre for Research on Evolution, Search and Testing, Strand, London WC2R 2LS, U.K.

†E-mail: Shin.Yoo@kcl.ac.uk

Contract/grant sponsor: EPSRC; contract/grant numbers: EP/D050863, GR/S93684, GR/T22872

Contract/grant sponsor: EU; contract/grant number: IST-33472

Contract/grant sponsor: DaimlerChrysler Berlin and Vizuri Ltd., London

entire test suite. This limitation forces consideration of techniques that seek to reduce the effort required for regression testing in various ways.

A number of different approaches have been studied to aid the regression testing process. The three major branches include test suite minimization, test case selection and test case prioritization. Test suite minimization is a process that seeks to identify and then eliminate the obsolete or redundant test cases from the test suite. Test case selection deals with the problem of selecting a subset of test cases that will be used to test the *changed* parts of the software. Finally, test case prioritization concerns the identification of the ‘ideal’ ordering of test cases that maximize desirable properties, such as early fault detection. Existing empirical studies show that the application of these techniques can be cost-effective.

This paper surveys work undertaken in these three related branches of regression testing. Section 2 introduces the nomenclature. Section 3 describes different test suite minimization techniques as well as their efficiency and effectiveness. Section 4 examines test case selection techniques according to the specific analysis technique used, and evaluates the strengths and weaknesses of each approach. Section 5 introduces test case prioritization techniques. Section 6 introduces meta-empirical studies concerning evaluation methodologies and cost-effectiveness analysis of regression testing techniques. Section 7 presents a summary of the field and identifies trends and issues. Section 8 introduces some gaps in the existing literature, thereby suggesting potential directions for future work. Section 9 concludes.

### 1.1. Motivation

When writing a survey paper there are two natural questions that need to be asked:

1. Why is this the right set of topics for a survey?
2. Is there already a recent survey in this area?

The first question concerns the motivation for the scope chosen for the survey, whereas the second concerns the perceived need for such a survey, once a suitable scope is established. For this paper the scope has been chosen to include topics on test suite minimization, Regression Test Selection (RTS) and test case prioritization. The reason for the choice of this scope is that these three topics are related by a common thread of optimization; each is an approach that optimizes the application of an existing pool of test cases.

The fact that all three approaches assume that the existence of a pool of test cases distinguishes these topics from test case generation, which seeks to create pools of test data. The three topics form a coherent set of approaches, each of which shares a common starting point; that the tester has a pool of test cases that is simply too large to allow all cases to be applied to the System Under Test (SUT). Each of the three approaches denotes a different way of coping with this problem of scale.

The relationship between the three techniques goes deeper than the mere shared application to pools of test data. It is not only the sets of problems addressed by each that exhibit overlap, but also the solution approaches that are applied. There is an intimate relationship between solutions to the three related problems, as this survey reveals. For instance, one way of selecting (or minimizing) a set of  $n$  test cases from a test pool would be to prioritize the whole set and pick the first  $n$  in priority order. Of course, there may be more optimal choices, since prioritization has to contend with any possible choice of  $n$ , whereas selection merely requires that an optimal choice is found for a given value of  $n$ .

Turning to the second question, the previous papers closest to this survey are a survey of RTS techniques undertaken in 1996 [1] and a recent systematic review on RTS [2] that concerns a specific set of research questions rather than a survey of the whole area. No previous survey on regression testing considered minimization, selection and prioritization collectively. The present survey claims that these classes of techniques are closely related to each other and denote a coherent sub-area of study. Our survey also includes recent applications of techniques that were surveyed in the earlier work of Rothermel and Harrold [1]. There are existing comparative studies on RTS [3–7], but these papers only concern a small number of specific RTS techniques and do not provide an overview of the field.

### 1.2. Selection of papers

This survey aims to collect and consider papers that deal with three regression testing techniques: test suite minimization, RTS and test case prioritization. Our intention is not to undertake a systematic review, but rather to provide a broad state-of-the-art view on these related fields. Many different approaches have been proposed to aid regression testing, which has resulted in a body of literature that is spread over a wide variety of domains and publication venues. The majority of surveyed literature has been published in the software engineering domain, and especially in the software testing and software maintenance literature. However, the regression testing literature also overlaps with those of programming language analysis, empirical software engineering and software metrics.

Therefore, the paper selection criteria on which this survey is based are the problems considered in papers, while focusing on the specific topics of minimization, selection and prioritization. The formal definitions of these problems are presented in Section 2.2. The selected papers are listed in the Appendix. Fast abstracts and short papers have been excluded.

## 2. BACKGROUND

This section introduces the basic concepts and definitions that form a nomenclature of regression testing and minimization, selection and prioritization techniques.

### 2.1. Regression testing

Regression testing is performed between two different versions of software in order to provide confidence that the newly introduced features of the SUT do not interfere with the existing features. While the exact details of the modifications made to SUT will often be available, they may not be easily available in some cases. For example, when the new version is written in a different programming language or when the source code is unavailable, modification data will also be unavailable.

The following notations are used to describe concepts in the context of regression testing. Let  $P$  be the current version of the program under test, and  $P'$  be the next version of  $P$ . Let  $S$  be the current set of specifications for  $P$ , and  $S'$  be the set of specifications for  $P'$ .  $T$  is the existing test suite. Individual test cases will be denoted by lower case:  $t$ .  $P(t)$  stands for the execution of  $P$  using  $t$  as input.

### 2.2. Distinction between classes of techniques

It is necessary at this point to establish a clear terminological distinction between the different classes of techniques described in the paper. Test suite minimization techniques seek to reduce the size of a test suite by eliminating redundant test cases from the test suite. Minimization is sometimes also called ‘test suite reduction’, meaning that the elimination is permanent. However, these two concepts are essentially interchangeable because all reduction techniques can be used to produce a temporary subset of the test suite, whereas any minimization techniques can be used to permanently eliminate test cases. More formally, following Rothermel *et al.* [8], the test suite minimization is defined as follows:

#### *Definition 1 (Test Suite Minimization Problem)*

*Given:* A test suite,  $T$ , a set of test requirements  $\{r_1, \dots, r_n\}$ , that must be satisfied to provide the desired ‘adequate’ testing of the program, and subsets of  $T$ ,  $T_1, \dots, T_n$ , one associated with each of the  $r_i$ s such that any one of the test cases  $t_j$  belonging to  $T_i$  can be used to achieve requirement  $r_i$ .

*Problem:* Find a representative set,  $T'$ , of test cases from  $T$  that satisfies all  $r_i$ s.

The testing criterion is satisfied when every test requirement in  $\{r_1, \dots, r_n\}$  is satisfied. A test requirement,  $r_i$ , is satisfied by any test case,  $t_j$ , that belongs to the  $T_i$ , a subset of  $T$ . Therefore, the representative set of test cases is the hitting set of the  $T_i$ s. Furthermore, in order to maximize the effect of minimization,  $T'$  should be the minimal hitting set of the  $T_i$ s. The minimal hitting set problem is an NP-complete problem as is the dual problem of the minimal set cover problem [9].

While test case selection techniques also seek to reduce the size of a test suite, the majority of selection techniques are *modification-aware*. That is, the selection is not only temporary (i.e. specific to the current version of the program), but also focused on the identification of the modified parts of the program. Test cases are selected because they are relevant to the changed parts of the SUT, which typically involves a white-box static analysis of the program code. Throughout this survey, the meaning of ‘test case selection problem’ is restricted to this modification-aware problem. It is also often referred to as the RTS problem. More formally, following Rothermel and Harrold [1], the selection problem is defined as follows (refer to Section 4 for more details on how the subset  $T'$  is selected):

*Definition 2 (Test Case Selection Problem)*

*Given:* The program,  $P$ , the modified version of  $P$ ,  $P'$  and a test suite,  $T$ .

*Problem:* Find a subset of  $T$ ,  $T'$ , with which to test  $P'$ .

Finally, test case prioritization concerns ordering test cases for early maximization of some desirable properties, such as the rate of fault detection. It seeks to find the optimal permutation of the sequence of test cases. It does not involve selection of test cases, and assumes that all the test cases may be executed in the order of the permutation it produces, but that testing may be terminated at some arbitrary point during the testing process. More formally, the prioritization problem is defined as follows:

*Definition 3 (Test Case Prioritization Problem)*

*Given:* A test suite,  $T$ , the set of permutations of  $T$ ,  $PT$ , and a function from  $PT$  to real numbers,  $f : PT \rightarrow \mathbb{R}$ .

*Problem:* To find  $T' \in PT$  such that  $(\forall T'')(T'' \in PT)(T'' \neq T')[f(T') \geq f(T'')]$ .

This survey focuses on papers that consider one of these three problems. Throughout the paper, these three techniques will be collectively referred to as ‘regression testing techniques’.

### 2.3. Classification of test cases

Leung and White present the first systematic approach to regression testing by classifying types of regression testing and test cases [10]. Regression testing can be categorized into *progressive* regression testing and *corrective* regression testing. Progressive regression testing involves changes of specifications in  $P'$ , meaning that  $P'$  should be tested against  $S'$ . On the other hand, corrective regression testing does not involve changes in specifications, but only in design decisions and actual instructions. It means that the existing test cases can be reused without changing their input/output relation.

Leung and White categorize test cases into five classes. The first three classes consist of test cases that already exist in  $T$ .

- *Reusable:* Reusable test cases only execute the parts of the program that remain unchanged between two versions, i.e. the parts of the program that are common to  $P$  and  $P'$ . It is unnecessary to execute these test cases in order to test  $P'$ ; however, they are called *reusable* because they may still be retained and reused for the regression testing of the future versions of  $P$ .
- *Retestable:* Retestable test cases execute the parts of  $P$  that have been changed in  $P'$ . Thus, retestable test cases should be re-executed in order to test  $P'$ .
- *Obsolete:* Test cases can be rendered obsolete because (1) their input/output relation is no longer correct due to changes in specifications, (2) they no longer test what they were designed to test due to modifications to the program or (3) they are ‘structural’ test cases that no longer contribute to structural coverage of the program.

The remaining two classes consist of test cases that have yet to be generated for the regression testing of  $P'$ .

- *New-structural:* New-structural test cases test the modified program constructs, providing structural coverage of the modified parts in  $P'$ .

- *New-specification*: New-specification test cases test the modified program specifications, testing the new code generated from the modified parts of the specifications of  $P'$ .

Leung and White go on to propose a retest strategy, in which a *test plan* is constructed based on the identification of changes in the program and classification of test cases. Although the definition of a test plan remains informal, it provides a basis for the subsequent literature. It is of particular importance to regression test case selection techniques, since these techniques essentially concern the problem of identifying *retestable* test cases. Similarly, test suite minimization techniques concern the identification of *obsolete* test cases. Test case prioritization also can be thought of as a more sophisticated approach to the construction of a test plan.

It should be noted that the subsequent literature focusing on the idea of selecting and reusing test cases for regression testing is largely concerned with corrective regression testing only. For progressive regression testing, it is very likely that new test cases are required in order to test the new specifications. So far, this aspect of the overall regression testing picture has been a question mainly reserved for test data generation techniques. However, the early literature envisages a ‘complete’ regression testing strategy that should also utilize test data generation techniques.

### 3. TEST SUITE MINIMIZATION

Test suite minimization techniques aim to identify redundant test cases and to remove them from the test suite in order to reduce the size of the test suite. The minimization problem described by Definition 1 can be considered as the minimal hitting set problem.

Note that the minimal hitting set formulation of the test suite minimization problem depends on the assumption that each  $r_i$  can be satisfied by a single test case. In practice, this may not be true. For example, suppose that the test requirement is functional rather than structural and, therefore, requires more than one test case to be satisfied. The minimal hitting set formulation no longer applies. In order to apply the given formulation of the problem, the functional granularity of test cases needs to be adjusted accordingly. The adjustment process may be either that a higher level of abstraction would be required so that each test case requirement can be met with a single test scenario composed of relevant test cases, or that a ‘large’ functional requirement needs to be divided into smaller sub-requirements that will correspond to individual test cases.

#### 3.1. Heuristics

The NP-completeness of the test suite minimization problem encourages the application of heuristics; previous work on test case minimization can be regarded as the development of different heuristics for the minimal hitting set problem [11–14].

Horgan and London applied linear programming to the test case minimization problem in their implementation of a data-flow-based testing tool, ATAC [13, 15]. Harrold *et al.* presented a heuristic for the minimal hitting set problem with the worst-case execution time of  $O(|T| * \max(|T_i|))$  [12]. Here  $|T|$  represents the size of the original test suite, and  $\max(|T_i|)$  represents the cardinality of the largest group of test cases among  $T_1, \dots, T_n$ .

Chen and Lau applied GE and GRE heuristics and compared the results to that of the Harrold–Gupta–Soffa (HGS) heuristic [11]. The GE and GRE heuristics can be thought of as variations of the greedy algorithm that is known to be an effective heuristic for the set cover problem [16]. Chen *et al.* defined *essential* test cases as the opposite of *redundant* test cases. If a test requirement  $r_i$  can be satisfied by one and only one test case, the test case is an essential test case. On the other hand, if a test case satisfies only a subset of the test requirements satisfied by another test case, it is a redundant test case. Based on these concepts, the GE and GRE heuristics can be summarized as follows:

- *GE heuristic*: first select all essential test cases in the test suite; for the remaining test requirements, use the additional greedy algorithm, i.e. select the test case that satisfies the maximum number of unsatisfied test requirements.

Table I. Example test suite taken from Tallam and Gupta [18]. The early selection made by the greedy approach,  $t_1$ , is rendered redundant by subsequent selections,  $\{t_2, t_3, t_4\}$ .

Test case	Testing requirements					
	$r_1$	$r_2$	$r_3$	$r_4$	$r_5$	$r_6$
$t_1$	x	x	x			
$t_2$	x			x		
$t_3$		x			x	
$t_4$			x			x
$t_5$					x	

- *GRE heuristic*: first remove all redundant test cases in the test suite, which may make some test cases essential; then perform the GE heuristic on the reduced test suite.

Their empirical comparison suggested that no single technique is better than the other. This is a natural finding, because the techniques concerned are heuristics rather than precise algorithms.

Offutt *et al.* also treated the test suite minimization problem as the dual of the minimal hitting set problem, i.e. the set cover problem [14]. Their heuristics can also be thought of as variations of the greedy approach to the set cover problem. However, they adopted several different test case orderings, instead of the fixed ordering in the greedy approach. Their empirical study applied their techniques to a mutation-score-based test case minimization, which reduced sizes of test suites by over 30%.

Whereas other minimization approaches primarily considered code-level structural coverage, Marré and Bertolino formulated test suite minimization as a problem of finding a spanning set over a graph [17]. They represented the structure of the SUT using a *decision-to-decision* graph (ddgraph). A ddgraph is a more compact form of the normal CFG since it omits any node that has one entering edge and one exiting edge, making it an ideal representation of the SUT for branch coverage. They also mapped the result of data-flow analysis onto the ddgraph for testing requirements such as *def-use* coverage. Once testing requirements are mapped to entities in the ddgraph, the test suite minimization problem can be reduced to the problem of finding the minimal spanning set.

Tallam and Gupta developed the greedy approach further by introducing the *delayed greedy* approach, which is based on the Formal Concept Analysis of the relation between test cases and testing requirements [18]. A potential weakness of the greedy approach is that the early selection made by the greedy algorithm can eventually be rendered redundant by the test cases subsequently selected. For example, consider the test suite and testing requirements depicted in Table I, taken from Tallam and Gupta [18]. The greedy approach will select  $t_1$  first as it satisfies the maximum number of testing requirements, and then continues to select  $t_2$ ,  $t_3$  and  $t_4$ . However, after the selection of  $t_2$ ,  $t_3$  and  $t_4$ ,  $t_1$  is rendered redundant. Tallam *et al.* tried to overcome this weakness by constructing a concept lattice, a hierarchical clustering based on the relation between test cases and testing requirements. Tallam *et al.* performed two types of reduction on the concept lattice. First, if a set of requirements covered by  $t_i$  is a superset of the set of requirements covered by  $t_j$ , then  $t_j$  is removed from the test suite. Second, if a set of test cases that cover requirement  $r_i$  is a subset of the set of test cases that cover requirement  $r_j$ , requirement  $r_i$  is removed. The concept lattice is a natural representation that supports the identification of these test cases. Finally, the greedy algorithm is applied to the transformed set of test cases and testing requirements. In the empirical evaluation, the test suites minimized by this ‘delayed greedy’ approach were either the same size or smaller than those minimized by the classical greedy approach or by the HGS heuristic.

Jeffrey and Gupta extended the HGS heuristic so that certain test cases are selectively retained [19, 20]. This ‘selective redundancy’ is obtained by introducing a secondary set of testing requirements. When a test case is marked as redundant with respect to the first set of testing requirements, Jeffrey and Gupta considered whether the test case is also redundant with respect to the second set of testing requirements. If it is not, the test case is still selected, resulting in a certain level of redundancy with respect to the first set of testing requirements. The empirical evaluation used

branch coverage as the first set of testing requirements and *all-uses* coverage information obtained by data-flow analysis. The results were compared to two versions of the HGS heuristic, based on branch coverage and *def-use* coverage. The results showed that, while their technique produced larger test suites, the fault-detection capability was better preserved compared to single-criterion versions of the HGS heuristic.

Whereas the selective redundancy approach considers the secondary criterion only when a test case is marked as being redundant by the first criterion, Black *et al.* considered a bi-criteria approach that takes into account both testing criteria [21]. They combined the *def-use* coverage criterion with the past fault-detection history of each test case using a weighted-sum approach and used Integer Linear Programming (ILP) optimization to find subsets. The weighted-sum approach uses weighting factors to combine multiple objectives. For example, given a weighting factor  $\alpha$  and two objectives  $o_1$  and  $o_2$ , the new and combined objective,  $o'$ , is defined as follows:

$$o' = \alpha o_1 + (1 - \alpha) o_2$$

Consideration of a secondary objective using the weighted-sum approach has been used in other minimization approaches [22] and prioritization approaches [23]. Hsu and Orso also considered the use of an ILP solver with multi-criteria test suite minimization [22]. They extended the work of Black *et al.* by comparing different heuristics for a multi-criteria ILP formulation: the weighted-sum approach, the prioritized optimization and a hybrid approach. In prioritized optimization, the human user assigns a priority to each of the given criteria. After optimizing for the first criterion, the result is added as a constraint, while optimizing for the second criterion, and so on. However, one possible weakness shared by these approaches is that they require additional input from the user of the technique in the forms of weighting coefficients or priority assignment, which might be biased, unavailable or costly to provide.

Contrary to these approaches, Yoo and Harman treated the problem of time-aware prioritization as a multi-objective optimization problem [24]. Instead of using a fitness function that combines selection and prioritization, they used a Pareto-efficient multi-objective evolutionary algorithm to simultaneously optimize for multiple objectives. You and Harman argued that the resulting Pareto-frontier not only provides solutions but also allows the tester to observe trade-offs between objectives, providing additional insights.

McMaster and Memon proposed a test suite minimization technique based on *call-stack coverage*. A test suite is represented by a set of unique maximum depth call stacks; its minimized test suite is a subset of the original test suite whose execution generates the same set of unique maximum depth call stacks. Note that their approach is different from simply using function coverage for test suite minimization. Consider two test cases,  $t_1$  and  $t_2$ , respectively, producing call stacks  $c_1 = \langle f_1, f_2, f_3 \rangle$  and  $c_2 = \langle f_1, f_2 \rangle$ . With respect to function coverage,  $t_2$  is rendered redundant by  $t_1$ . However, McMaster and Memon regard  $c_2$  to be unique from  $c_1$ . For example, it may be that  $t_2$  detects a failure that prevents the invocation of function  $f_3$ . Once the call-stack coverage information is collected, the HGS heuristic can be applied. McMaster and Memon later applied the same approach to Graphical User Interface (GUI) testing [25]. It was also implemented for object-oriented systems by Smith *et al.* [26].

While most test suite minimization techniques are based on some kind of coverage criteria, there do exist interesting exceptions. Harder *et al.* approached test suite minimization using operational abstraction [27]. An operational abstraction is a formal mathematical description of program behaviour. While it is identical to formal specifications in form, an operational abstraction expresses dynamically observed behaviour. Harder *et al.* use the widely studied Daikon dynamic invariant detector [28] to obtain operational abstractions. Daikon requires executions of test cases for the detection of possible program invariants. Test suite minimization is proposed as follows: if the removal of a test case does not change the detected program invariant, it is rendered redundant. They compared the operational abstraction approach to branch coverage-based minimization. While their approach resulted in larger test suites, it also maintained higher fault-detection capability. Moreover, Harder *et al.* also showed that test suites minimized for coverage adequacy can often be improved by considering operational abstraction as an additional minimization criterion.

Leitner *et al.* propose a somewhat different version of the minimization problem [29]. They start from the assumption that they already have a *failing test case*, which is too complex and too long for the human tester to understand. Note that this is often the case with randomly generated test data; the test case is often simply too complex for the human to establish the cause of failure. The goal of minimization is to produce a shorter version of the test case; the testing requirement is that the shorter test case should still reproduce the failure. This minimization problem is interesting because there is no uncertainty about the fault-detection capability; it is given as a testing requirement. Leitner *et al.* applied the widely studied *delta-debugging* technique [30] to reduce the size of the failing test case.

Schroeder and Korel proposed an approach of test suite minimization for black-box software testing [31]. They noted that the traditional approach of testing black-box software with combinatorial test suites may result in redundancy, since certain inputs to the software may not affect the outcome of the output being tested. They first identified, for each output variable, the set of input variables that can affect the outcome. Then, for each output variable, an individual combinatorial test suite is generated with respect to only those input variables that may affect the outcome. The overall test suite is a union of all combinatorial test suites for individual output variables. This has a strong connection to the concept of Interaction Testing, which is discussed in detail in Section 5.2.

Other work has focused on model-based test suite minimization [32, 33]. Vaysburg *et al.* introduced a minimization technique for model-based test suites that uses dependence analysis of Extended Finite State Machines (EFSMs) [32]. Each test case for the model is a sequence of transitions. Through dependence analysis of the transition being tested, it is possible to identify the transitions that affect the tested transition. In other words, testing a transition  $T$  can be thought of as testing a set of other transitions that affect  $T$ . If a particular test case tests the same set of transitions as some other test case, then it is considered to be redundant. Korel *et al.* extended this approach by combining the technique with automatic identification of changes in the model [33]. The dependence analysis-based minimization technique was applied to the set of test cases that were identified to execute the modified transitions. Chen *et al.* extended Korel's model-based approach to incorporate more complex representations of model changes [34].

A risk shared by most test suite minimization techniques is that a *discarded* test case may detect a fault. In some domains, however, test suite minimization techniques can enjoy the certainty of guaranteeing that discarding a test case will not reduce the fault-detection capability. Anido *et al.* investigated test suite minimization for testing Finite State Machines (FSMs) in this context [35]. When only some components of the SUT need testing, the system can be represented as a composition of two FSMs: *component* and *context*. The context is assumed to be fault-free. Therefore, certain transitions of the system that concern only the context can be identified to be redundant. Under the 'testing in context' assumption (i.e. the context is fault-free), it follows that it is possible to guarantee that a discarded test case cannot detect faults.

Kaminski and Ammann investigated the use of a logic criterion to reduce test suites while guaranteeing fault detection in testing predicates over Boolean variables [36]. From the formal description of fault classes, it is possible to derive a hierarchy of fault classes [37]. From the hierarchy, it follows that the ability to detect a class of faults may guarantee the detection of another class. Therefore, the size of a test suite can be reduced by executing only those test cases for the class of faults that subsume another class, whenever this is feasible.

### 3.2. Impact on fault-detection capability

Although the techniques discussed so far reported reduced test suites, there has been a persistent concern about the effect that the test suite minimization has on the fault-detection capability of test suites. Several empirical studies were conducted to investigate this effect [8, 38–40].

Wong *et al.* studied 10 common Unix programs using randomly generated test suites; this empirical study is often referred to as the WHLM study [39]. To reduce the size of the test suites, they used the ATAC testing tool developed by Horgan and London [13, 15]. First, a large pool of test cases was created using a random test data generation method. From this pool, several test suites with different total block coverage were generated. After generating test suites randomly,

artificial faults were seeded into the programs. These artificial faults were then categorized into four groups. Faults in Quartile-I can be detected by [0–25)% of the test cases from the original test suite; the percentage for Quartile-II, III and IV is [25–50)%, [50–75)% and [75–100)%, respectively. Intuitively, faults in Quartile-I are harder to detect than those in Quartile-IV. The effectiveness of the minimization itself was calculated as follows:

$$\left( 1 - \frac{\text{number of test cases in the reduced test suite}}{\text{number of test cases in the original test suite}} \right) * 100\%$$

The impact of test suite minimization was measured by calculating the reduction in fault detection effectiveness as follows:

$$\left( 1 - \frac{\text{number of faults detected by the reduced test suite}}{\text{number of faults detected by the original test suite}} \right) * 100\%$$

By categorizing the test suites (by different levels of block coverage) and test cases (by difficulty of detection), they arrived at the following observation. First, the reduction in size is greater in test suites with a higher block coverage in most cases. This is natural considering that test suites with higher block coverage will require more test cases in general. The average size reduction for test suites with (50–55)%,(60–65)%,(70–75)%,(80–85)% and (90–95)% block coverage was 1.19, 4.46, 7.78, 17.44, 44.23%, respectively. Second, the fault-detection effectiveness was decreased by test case reduction, but the overall decrease in fault-detection effectiveness is not excessive and could be regarded as worthwhile for the reduced cost. The average effectiveness reduction for test suites with (50–55)%,(60–65)%,(70–75)%,(80–85)% and (90–95)% block coverage was 0, 0.03, 0.01, 0.38, 1.45%, respectively. Third, test suite minimization did not decrease the fault-detection effectiveness for faults in Quartile-IV at all, meaning that all faults in Quartile-IV had been detected by the reduced test suite. The average decrease in fault-detection effectiveness for Quartile-I, II and III was 0.39, 0.66, and 0.098%, respectively. The WHLM study concluded that, if the cost of testing is directly related to the number of test cases, then the use of the reduction technique is recommended.

Wong *et al.* followed up on the WHLM study by applying the ATAC tool to test suites of another, bigger C program; this empirical study is often referred to as the WHLP study [40]. The studied program, space, was an interpreter for the Array Description Language (ADL) developed by the European Space Agency. In the WHLP study, test cases were generated, not randomly, but from the operational profiles of space. That is, each test case in the test case pool was generated so that it matches an example of real usage of space recorded in an operational profile. From the test case pool, different types of test suites were generated. The first group of test suites was constructed by randomly choosing a fixed number of test cases from the test case pool. The second group of test suites was constructed by choosing test cases from the test case pool until a predetermined block coverage target was met. The faults in the program were not artificial, but real faults that were retrieved from development logs.

The results of the WHLP study confirmed the findings of the WHLM study. As in the WHLM study, test suites with low initial block coverage (50, 55, 60 and 65%) showed no decrease in fault-detection effectiveness, after test suite minimization. For both the fixed size test suites and fixed coverage test suites, the application of the test case reduction technique did not affect the fault-detection effectiveness in any significant way; the average effectiveness reduction due to test suite minimization was less than 7.28%.

While both the WHLM and WHLP studies showed that the impact of test suite minimization on fault-detection capability was insignificant, other empirical studies produced radically different findings. Rothermel *et al.* also studied the impact of test suite minimization on the fault-detection capability [38]. They applied the HGS heuristics to the Siemens suite [41], and later expanded this to include space [8]. The results from these empirical studies contradicted the previous findings of the WHLM and WHMP studies.

For the study of the Siemens suite [38], Rothermel *et al.* constructed test suites from the test case pool provided by the Siemens suite so that the test suites include varying amounts of redundant

test cases that do not contribute to the decision coverage of the test suite. The effectiveness and impact of reduction was measured using the same metrics that were used in the WHLM study.

Rothermel *et al.* reported that the application of the test suite minimization technique produced significant savings in test suite size. The observed tendency in size reduction suggested a logarithmic relation between the original test suite size and the reduction effectiveness. The results of logarithmic regression confirmed this.

However, Rothermel *et al.* also reported that, due to the size reduction, the fault-detection capabilities of test suites were severely compromised. The reduction in fault-detection effectiveness was over 50% for more than half of 1000 test suites considered, with some cases reaching 100%. Rothermel *et al.* also reported that, unlike the size reduction effectiveness, the fault-detection effectiveness did not show any particular correlation with the original test suite size.

This initial empirical study was subsequently extended [8]. For the Siemens suite, the results of the HGS heuristic were compared to random reduction by measuring the fault-detection effectiveness of randomly reduced test suites. Random reduction was performed by randomly selecting, from the original test suite, the same number of test cases as in the reduced version of the test suite. The results showed that random reduction produced larger decreases in fault-detection effectiveness. To summarize the results for the Siemens suite, the test suite minimization technique produced savings in test suite size, but at the cost of decreased fault-detection effectiveness; however, the reduction heuristic showed better fault-detection effectiveness than the random reduction technique.

Rothermel *et al.* also expanded the previous empirical study by including the larger program, space [8]. The reduction in size observed in the test suites of space confirmed the findings of the previous empirical study of the Siemens suite; the size reduction effectiveness formed a logarithmic trend, plotted against the original test suite size, similar to the programs in the Siemens suite. More importantly, the reduction in fault-detection effectiveness was less than those of the Siemens suite programs. The average reduction in fault-detection effectiveness of test suites reduced by the HGS heuristic was 8.9%, whereas that of test suites reduced by random reduction was 18.1%.

Although the average reduction in fault-detection effectiveness is not far from that reported for the WHLP study in the case of space, those of the Siemens suite differed significantly from both the WHLP study and the WHLM study, which reported that the application of the minimization technique did not have significant impact on fault-detection effectiveness. Rothermel *et al.* [8] pointed out the following differences between these empirical studies as candidates for the cause(s) of the contradictory findings, which is paraphrased as follows:

1. *Different subject programs*: the programs in the Siemens suite are generally larger than those studied in both the WHLM and WHLP studies. Difference in program size and structure certainly could have impact on the fault-detection effectiveness.
2. *Different types of test suites*: the WHLM study used test suites that were not coverage-adequate and were much smaller compared to test suites used by Rothermel *et al.* The initial test pools used in the WHLM study also did not necessarily contain any minimum number of test cases per covered item. These differences could have contributed to less redundancy in test suites, which led to reduced likelihood that test suite minimization will exclude fault-revealing test cases.
3. *Different types of test cases*: the test suites used in the WHLM study contained a few test cases that detected all or most of the faults. When such strong test cases are present, reduced versions of the test suites may well show little loss in fault-detection effectiveness.
4. *Different types of faults*: the faults studied by Rothermel *et al.* were all Quartile-I faults according to the definition of the WHLM study, whereas only 41% of the faults studied in the WHLM study belonged to the Quartile-I group. By having more ‘easy-to-detect’ faults, the test suites used in the WHLM study could have shown less reduction in fault-detection effectiveness after test suite minimization.

Considering the two contradicting empirical results, it is natural to conclude that the question of evaluating the effectiveness of the test suite minimization technique is very hard to answer in general and for all testing scenarios. The answer depends on too many factors, such as the structure

of the SUT, the quality of test cases and test suites and the types of faults present. This proliferation of potential contributory factors makes it very difficult to generalize any empirical result.

The empirical studies from WHLM, WHLP and Rothermel *et al.* all evaluated the effectiveness of test suite minimization in terms of two metrics: percentage size reduction and percentage fault-detection reduction. McMaster and Memon noticed that neither metric considers the actual role each testing requirement plays on fault detection [42]. Given a set of test cases,  $TC$ , a set of known faults,  $KF$  and a set of testing requirements,  $CR$ , *fault correlation* for a testing requirement  $i \in CR$  to fault  $k \in KF$  is defined as follows:

$$\frac{|\{j \in TC | j \text{ covers } i\} \cap \{j \in TC | j \text{ detects } k\}|}{|\{j \in TC | j \text{ covers } i\}|}$$

The expected probability of finding a given fault  $k$  after test suite minimization is defined as the maximum fault correlation of all testing requirements with  $k$ . From this, the probability of detecting all known faults in  $KF$  is the product of expected probability of finding all  $k \in KF$ . Since  $CR$  is defined by the choice of minimization criterion, e.g. branch coverage or call-stack coverage, comparing the probability of detecting all known faults provides a systematic method of comparing different minimization criteria, without depending on a specific heuristic for minimization. The empirical evaluation of McMaster and Memon compared five different minimization criteria for the minimization of test suites for GUI-intensive applications: event coverage (i.e. each event is considered as a testing requirement), event interaction coverage (i.e. each pair of events is considered as a testing requirement), function coverage, statement coverage and call-stack coverage proposed in [43]. While call-stack coverage achieved the highest average probability of detecting all known faults, McMaster and Memon also found that different faults correlate more highly with different criteria. This analysis provides valuable insights into the selection of minimization criterion.

Yu *et al.* considered the effect of test suite minimization on fault localization [44]. They applied various test suite minimization techniques to a set of programs, and measured the impact of the size reduction on the effectiveness of coverage-based fault localization techniques. Yu *et al.* reported that higher reduction in test suite size, typically achieved by statement coverage-based minimization, tends to have a negative impact on fault localization, whereas minimization techniques that maintain higher levels of redundancy in test suites have negligible impact.

#### 4. TEST CASE SELECTION

Test case selection, or the RTS problem, is essentially similar to the test suite minimization problem; both problems are about choosing a subset of test cases from the test suite. The key difference between these two approaches in the literature is whether the focus is upon the changes in the SUT. Test suite minimization is often based on metrics such as coverage measured from a single version of the program under test. By contrast, in RTS, test cases are selected because their execution is relevant to the changes between the previous and the current version of the SUT.

To recall Definition 2,  $T'$  ideally should contain all the *faults-revealing* test cases in  $T$ , i.e. the test cases that will reveal faults in  $P'$ . In order to define this formally, Rothermel and Harrold introduced the concept of a *modification-revealing* test case [45]. A test case  $t$  is modification-revealing for  $P$  and  $P'$  if and only if  $P(t) \neq P'(t)$ . Given the following two assumptions, it is possible to identify the fault-revealing test cases for  $P'$  by finding the modification-revealing test cases for  $P$  and  $P'$ .

- ***P*-Correct-for-*T* Assumption:** It is assumed that, for each test case  $t \in T$ , when  $P$  was tested with  $t$ ,  $P$  halted and produced the correct output.
- **Obsolete-Test-Identification Assumption:** It is assumed that there exists an effective procedure for determining, for each test case  $t \in T$ , whether  $t$  is obsolete for  $P'$ .

From these assumptions, it is clear that every test case in  $T$  terminates and produces correct output for  $P$ , and is also supposed to produce the same output for  $P'$ . Therefore, a modification-revealing

test case  $t$  must be also fault-revealing. Unfortunately, it is not possible to determine whether a test case  $t$  is fault-revealing against  $P'$  or not because it is undecidable whether  $P'$  will halt with  $t$ . Rothermel considers a weaker criterion for the selection, which is to select all *modification-traversing* test cases in  $T$ . A test case  $t$  is modification-traversing for  $P$  and  $P'$  if and only if:

1. it executes new or modified code in  $P'$  or
2. it formerly executed code that had been deleted in  $P'$ .

Rothermel also introduced the third assumption, which is the Controlled-Regression-Testing assumption.

- *Controlled-Regression-Testing Assumption:* When  $P'$  is tested with  $t$ , all factors that might influence the output of  $P'$ , except for the code in  $P'$ , are kept constant with respect to their states when  $P$  was tested with  $t$ .

Given that the Controlled-Regression-Testing assumption holds, a non-obsolete test case  $t$  can thereby be modification-revealing only if it is also modification-traversing for  $P$  and  $P'$ . Now, if the  $P$ -Correct-for- $T$  assumption and the Obsolete-Test-Identification assumption hold along with the Controlled-Regression-Testing assumption, then the following relation also holds between the subset of fault-revealing test cases,  $T_{fr}$ , the subset of modification-revealing test cases,  $T_{mr}$ , the subset of modification-traversing test cases,  $T_{mt}$ , and the original test suite,  $T$ :

$$T_{fr} = T_{mr} \subseteq T_{mt} \subseteq T$$

Rothermel and Harrold admitted that the Controlled-Regression-Testing assumption may not be always practical, since certain types of regression testing may make it impossible to control the testing environment, e.g. testing of a system ported to different operating system [1]. Other factors like non-determinism in programs and time dependencies are also difficult to control effectively. However, finding the subset of modification-traversing test cases may still be a useful approach in practice, because  $T_{mt}$  is the closest approximation to  $T_{mr}$  that can be achieved without executing all test cases. In other words, by finding  $T_{mt}$ , it is possible to exclude those test cases that are guaranteed not to reveal any fault in  $P'$ . The widely used term, *safe RTS*, is based on this concept [46]. A safe RTS technique is not safe from all possible faults; however, it is safe in a sense that, if there exists a modification-traversing test case in the test suite, it will definitely be selected.

Based on Rothermel's formulation of the problem, it can be said that test case selection techniques for regression testing focus on identifying the modification-traversing test cases in the given test suite. The details of the selection procedure differ according to how a specific technique defines, seeks and identifies modifications in the program under test. Various approaches have been proposed using different techniques and criteria including Integer Programming [47, 48], data-flow analysis [49–52], symbolic execution [53], dynamic slicing [54], CFG graph-walking [46, 55–57], textual difference in source code, [58, 59] SDG slicing [60], path analysis [61], modification detection [62], firewall [63–66], CFG cluster identification [67] and design-based testing [68, 69]. The following subsections describe these in more detail, highlighting their strengths and weaknesses.

#### 4.1. Integer programming approach

One of the earliest approaches to test case selection was presented by Fischer *et al.*, who used Integer Programming (IP) to represent the selection problem for testing FORTRAN programs [47, 48]. Lee and He implemented a similar technique [70]. Fischer first defined a program segment as a single-entry, single-exit block of code whose statements are executed sequentially. Their selection algorithm relies on two matrices that describe the relation between program segments and test cases, as well as between different program segments.

For a program with  $m$  segments and  $n$  test cases, the IP formulation is given as the problem of finding the decision vector,  $\langle x_1, \dots, x_n \rangle$ , that minimizes the following objective function,  $Z$ :

$$Z = c_1x_1 + c_2x_2 + \dots + c_nx_n$$

subject to:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &\geq b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &\geq b_2 \\ &\vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n &\geq b_m \end{aligned}$$

The decision vector,  $\langle x_1, \dots, x_n \rangle$ , represents the subset of selected test cases;  $x_i$  is equal to 1 if the  $i$ th test case is included; 0 otherwise. The coefficients,  $c_1, \dots, c_n$ , represent the cost of executing each corresponding test case; Fischer *et al.* used the constant value of 1 for all coefficients, treating all test cases as being equally expensive. The test case dependency matrix,  $a_{11}, \dots, a_{mn}$  represents the relations between test cases and the program segments. The element  $a_{ij}$  is equal to 1 if the  $i$ th program segment is executed by the test case  $j$ ; 0 otherwise.

After deriving the series of inequalities, the set of  $b_k$  values are determined by using a reachability matrix that describes the program segments that are reachable from other segments. Using this approach, when the modified segments are known, it is possible to calculate all the segments that are reachable from the modified segments, which thus need to be tested at least once. The integer programming formulation is completed by assigning 1 to the  $b$  values for all the segments that need to be tested. The inequality,  $a_{11}x_1 + \cdots + a_{in}x_n \geq b_i$ , thus ensures that at least one included test case covers the program element reachable from a change.

Hartmann and Robson implemented and extended a version of Fischer's algorithm in order to apply the technique to C [71–73]. They treat subroutines as segments, achieving subroutine coverage rather than statement coverage.

One weakness in Fischer's approach is its inability to deal with control-flow changes in  $P'$ . The test case dependency matrix,  $a_{11}, \dots, a_{mn}$ , depends on the control-flow structure of the program under test. If the control-flow structure changes, the test case dependency matrix can be updated only by executing all the test cases, which negates the point of applying the selection technique.

#### 4.2. Data-flow analysis approach

Several test case selection techniques have been proposed based on data-flow analysis [49–52]. Data-flow analysis-based selection techniques seek to identify new, modified or deleted definition-use pairs in  $P'$ , and select test cases that exercise these pairs.

Harrold and Soffa presented data-flow analysis as the testing criterion for an incremental approach to unit testing during the maintenance phase [50]. Taha *et al.* built upon this idea and presented a test case selection framework based on an incremental data-flow analysis algorithm [52]. Harrold and Soffa developed both intra-procedural and inter-procedural selection techniques [51, 74]. Gupta *et al.* applied program slicing techniques to identify definition-use pairs that are affected by a code modification [49]. The use of slicing techniques enabled identification of definition-use pairs that need to be tested without performing a complete data-flow analysis, which is often very costly. Wong *et al.* combined a data-flow selection approach with a coverage-based minimization and prioritization to further reduce the effort [75].

One weakness shared by all data-flow analysis-based test case selection techniques is the fact that they are unable to detect modifications that are unrelated to data-flow change. For example, if  $P'$  contains new procedure calls without any parameter, or modified output statements that contain no variable uses, data-flow techniques may not select test cases that execute these.

Fisher II *et al.* applied the Data-flow-based RTS approach for test re-use in spreadsheet programs [76]. Fisher II *et al.* proposed an approach called What-You-See-Is-What-You-Test (WYSIWYT) to provide incremental, responsive and visual feedback about the *testedness* of cells in spreadsheets. The WYSIWYT framework collects and updates data-flow information incrementally as the user of the spreadsheet makes modifications to cells, using Cell Relation Graph.

Interestingly, the data-flow analysis approach to re-test spreadsheets is largely free from the difficulties associated with data-flow testing of procedural code, because spreadsheet programs lack tricky semantic features such as aliasing and unstructured control flow. This makes spreadsheet programs better suited to a data-flow analysis approach.

#### 4.3. Symbolic execution approach

Yau and Kishimoto presented a test case selection technique based on symbolic execution of the SUT [53]. In symbolic execution of a program, the variables' values are treated as symbols, rather than concrete values [77]. Yau and Kishimoto's approach can be thought of as an application of symbolic execution and input partitioning to the test case selection problem. First, the technique statically analyses the code and specifications to determine the input partitions. Next, it produces test cases so that each input partition can be executed at least once. Given information on where the code has been modified, the technique then identifies the edges in the control-flow graph (CFG) that lead to the modified code. While symbolically executing all test cases, the technique determines test cases that traverse edges that do not reach any modification. The technique then selects all test cases that reach new or modified code. For the symbolic test cases that reach modifications, the technique completes the execution; the real test cases that match these symbolic test cases should be retested.

While it is theoretically powerful, the most important drawback of the symbolic execution approach is the algorithmic complexity of the symbolic execution. Yau and Kishimoto acknowledge that symbolic execution can be very expensive. Pointer arithmetic can also present challenging problems for symbolic execution-based approaches.

#### 4.4. Dynamic slicing-based approach

Agrawal *et al.* introduced a family of test case selection techniques based on different program slicing techniques [54]. An *execution slice* of a program with respect to a test case is what is usually referred to as an execution trace; it is the set of statements executed by the given test case. A *dynamic slice* of a program with respect to a test case is the set of statements in the execution slice that have an influence on an output statement. Since an execution slice may contain statements that do not affect the program output, a dynamic slice is a subset of an execution slice. For example, consider the program shown in Figure 1. It contains two faults in line S3 and S11, respectively. The execution slice of the program with respect to test case  $T_3$  in Table II is shown in column ES of Figure 1. The dynamic slice of the program with respect to test case  $T_1$  in Table II is shown in column DS of Figure 1.

In order to make the selection more precise, Agrawal *et al.* proposed two additional slicing criteria: a *relevant slice* and an *approximate relevant slice*. A relevant slice of a program with respect to a test case is the dynamic slice with respect to the same test case together with all the predicate statements in the program that, if evaluated differently, could have caused the program to produce a different output. An approximated relevant slice is a more conservative approach to include predicates that could have caused a different output; it is the dynamic slice with all the predicate statements in the execution slice. By including all the predicates in the execution slice, an approximated relevant slice caters for the indirect references via pointers. For example, consider the correction of S3 in the program shown in Figure 1. The dynamic slice of  $T_4$  does not include S3 because the class value of  $T_4$  is not affected by any of the lines between S3 and S8. However, the relevant slice of  $T_4$ , shown in column DS of Figure 1, does include S3 because it could have affected the output when evaluated differently.

The test suite and the previous version of the program under test are preprocessed using these slicing criteria; each test case is connected to a slice,  $sl$ , constructed by one of the four slicing criteria. After the program is modified, test cases for which  $sl$  contains the modified statement should be executed again. For example, assume that the fault in S11, detected by  $T_5$ , is corrected. The program should be retested with  $T_5$ . However,  $T_3$  need not be executed because the execution slice of  $T_3$ , shown in column ES of Figure 1, does not contain S11. Similarly, assume that the fault in S3, detected by  $T_6$ , is corrected. The program should be retested with  $T_6$ . The execution

```

Slices
ES DS RS
x x x   S1: read(a,b,c);
x x   S2: class := scalene;
x x   S3: if a = b or b = a
          S4:      class := isosceles;
x x   S5: if a * a = b * b + c * c
x   S6:      class := right
x x x   S7: if a = b and b = c
x   S8:      class := equilateral
x x x   S9: case class of:
x   S10:      right       : area = b * c / 2;
x   S11:      equilateral : area = a * 2 * sqrt(3) / 4;
x   S12:      otherwise    : s := (a + b + c) / 2;
x   S13:                  area := sqrt(s * (s-a) * (s-b) * (s-c));
x   S14: end;
x   S15: write(class, area);

```

Figure 1. Example triangle classification program taken from Agrawal *et al.* [54]. Note that it is assumed that the input vector is sorted in descending order. It contains two faults. In S3,  $b = a$  should be  $b = c$ . In S11,  $a * 2$  should be  $a * a$ . The column ES represents the statements that belong to the execution slice with respect to test case  $T_3$  in Table II. Similarly, column DS represents the dynamic slice with respect to test case  $T_1$  in Table II and column RS the relevant slice with respect to test case  $T_4$  in Table II.

Table II. Test cases used with the program shown in Figure 1, taken from Agrawal *et al.* [54]. Note that  $T_5$  detects the fault in S11, because the value for area should be 3.90. Similarly,  $T_6$  detects the fault in S3, because the value for class should be isosceles.

Test case	Input			Output	
	a	b	c	Class	Area
$T_1$	2	2	2	equilateral	1.73
$T_2$	4	4	3	isosceles	5.56
$T_3$	5	4	3	right	6.00
$T_4$	6	5	4	scalene	9.92
$T_5$	3	3	3	equilateral	2.60
$T_6$	4	3	3	scalene	4.47

slice technique selects all six test cases,  $T_1-T_6$ , after the correction of the fault in S3 because the execution slices of all six test cases include S3. However, it is clear that  $T_1$  and  $T_3$  are not affected by the correction of S3; their class values are overwritten after the execution of S3. The dynamic slicing technique overcomes this weakness. The dynamic slice of  $T_1$  is shown in column DS of Figure 1. Since S3 does not affect the output of  $T_1$ , it is not included in the dynamic slice. Therefore, the modification of S3 does not necessitate the execution of  $T_1$ .

Agrawal *et al.* first built their technique on cases in which modifications are restricted to those that do not alter the CFG of the program under test. As long as the CFG of the program remains the same, their technique is safe and can be regarded as an improvement over Fischer's integer programming approach. Slicing removes the need to formulate the linear programming problem, reducing the effort required from the tester. Agrawal *et al.* later relaxed the assumption about static CFGs in order to cater for modifications in the CFGs of the SUT. If a statement  $s$  is added to  $P$ , now the slice  $sl$  contains all the statements in  $P$  that uses the variables defined in  $s$ . Similarly, if a predicate  $p$  is added to  $P$ , the slice  $sl$  contains all the statements in  $P$  that are control-dependent on  $p$ . This does cater for the changes in the CFG to some degree, but it is not complete. For example, if the added statement is a simple output statement that does not define or use any variable, then this statement can still be modification-revealing. However, since the new statement does not contain any variable, its addition will not affect any of the existing slices, resulting in an empty selection.

#### 4.5. Graph-walk approach

Rothermel and Harrold presented regression test case selection techniques based on graph-walking of Control Dependence Graphs (CDGs), Program Dependence Graphs (PDGs), System Dependence Graphs (SDGs) and CFGs [46, 56, 57, 78]. The CDG is similar to PDG but lacks data dependence relations. By performing a depth-first traversal of the CDGs of both  $P$  and  $P'$ , it is possible to identify points in a program through which the execution trace reaches the modifications [46]. If a node in the CDG of  $P$  is not lexically equivalent to the corresponding node in the CDG of  $P'$ , the algorithm selects all the test cases that execute the control-dependence predecessors of the mismatching node. The CDG-based selection technique does not cater for inter-procedural regression test case selection; Rothermel and Harrold recommend application of the technique at the individual procedural level.

Rothermel and Harrold later extended the graph-walking approach to use PDGs for intra-procedural selection, and SDGs for inter-procedural selection [56]. A weakness of the CDG-based technique is that, due to the lack of data dependence, the technique will select test cases that execute modified definitions but not the actual uses of a variable. If the modified definition of a variable is never used, it cannot contribute to any different output, and therefore its inclusion is not necessary for safe regression testing. PDGs contain data dependence for a single procedure; SDGs extend this to a complete program with multiple procedures. By using these graphs, Rothermel and Harrold's algorithm is able to check whether a modified definition of a variable is actually used later.

Rothermel and Harrold later presented the graph-walking approach based on CFGs [57]. The CFG-based technique essentially follows the approach introduced for the CDG-based technique, but on CFGs rather than on CDGs. Since CFG is a much simpler representation of the structure of a program, the CFG-based technique may be more efficient. However, the CFG lacks data dependence information, so the CFG-based technique may select test cases that are not capable of producing different outputs from the original programs, as explained above. The technique has been evaluated against various combinations of subject programs and test suites [79]. Ball improved the precision of the graph-walk approach with respect to branch coverage [80].

Rothermel *et al.* extended the CFG-based graph-walk approach for object-oriented software using the Inter-procedural Control-Flow Graph (ICFG) [81]. The ICFG connects methods using *call* and *return* edges. Harrold *et al.* adopted a similar approach for test case selection for Java software, using the Java Interclass Graph as representation (JIG) [82]. Xu and Rountev later extended this technique to consider AspectJ programs by incorporating the interactions between methods and advices at certain join points into the CFG [83]. Zhao *et al.* also considered a graph representation of AspectJ programs to apply a graph-walk approach for RTS [84]. Beydeda and Gruhn extended the graph-walk approach by adding black-box data-flow information to the Class Control-Flow Graph (CCFG) to test object-oriented software [85].

Orso *et al.* considered using different types of graph representation of the system to improve the scalability of graph-walk approach [86]. Their approach initially relies on a high-level graph representation of SUT to identify the parts of the system to be further analysed. Subsequently, the technique uses more detailed graph representation to perform more precise selection.

One strength of the graph-walk approach is its generic applicability. For example, it has been successfully used in black-box testing of re-usable classes [87]. Martins and Vieira captured the behaviours of a re-usable class by constructing a directed graph called the Behavioural Control-Flow Graph (BCFG) from the Activity Diagram (AD) of the class. The BCFG is a directed graph,  $G = (V, E, s, x)$ , with vertices  $V$ , edges  $E$ , a unique entry vertex  $s$  and an exit vertex  $x$ . Each vertex contains a label that specifies the signature of a method; each edge is also labelled according to the corresponding guards in AD. A path in  $G$  from  $s$  to  $x$  represents a possible life history of an object. By mapping changes made to the object to its BCFG and applying the graph-walking algorithm, it is possible to select test cases based on the behavioural difference between two versions of the same object. This approach requires traceability between the behavioural model and the actual test cases, because test cases are selected, not based on their structural coverage, but based on their behavioural coverage measured on BCFG. ADs have also been directly used for RTS by Chen *et al.* [88].

Orso *et al.* used a variation of the graph-walk approach to consider an RTS technique based on meta-data and specifications obtained from software components [89, 90]. They presented two different techniques based on meta-data: code-based RTS using component meta-data and specification-based RTS using component meta-data. For code-based RTS, it was assumed that each software component was capable of providing structural coverage information, which was fed into the graph-walk algorithm. For specification-based RTS, the component specification was represented in UML state-chart diagrams, which were used by the graph-walk algorithm.

The graph-walk algorithm has also been applied to test web services, despite the challenges that arise from the distributed nature of web services [91–95]. Several different approaches have been introduced to overcome these challenges. Lin *et al.* adopted the JIG-based approach after transforming the web services to a single-JVM local application [91]. Ruth *et al.* collected a coarse-grained CFG from developers of each web service that forms a part of the entire application [92, 93, 95]. Finally, Tarhini *et al.* utilized Timed Labelled Transition System (TLTS), which is a coarse-grained representation of web services that resemble a labelled state machine [94].

#### 4.6. Textual difference approach

Volkolos and Frankl proposed a selection technique based on the textual difference between the source code of two versions of SUT [58, 59]. They identified modified parts of SUT by applying the `diff` Unix tool to the source code of different versions. The source code was pre-processed into canonical forms to remove the impact of cosmetic differences. Although their technique operates on a different representation of SUT, its behaviour is essentially very similar to that of the CFG-based graph-walk approach.

#### 4.7. SDG slicing approach

Bates and Horwitz proposed test case selection techniques based on program slices from Program Dependency Graphs (PDGs) [60]. Bates and Horwitz approach the RTS problem in two stages. First, all the test cases that can be reused for  $P'$  need to be identified. Bates and Horwitz introduce the definition of an equivalent execution pattern. If statements  $s$  and  $s'$  belong to  $P$  and  $P'$ , respectively,  $s$  and  $s'$  have *equivalent execution patterns* if and only if all of the following hold:

1. For any input file on which both  $P$  and  $P'$  terminate normally,  $s$  and  $s'$  are exercised the same number of times.
2. For any input file on which  $P$  terminates normally but  $P'$  does not,  $s'$  is exercised at most as many times as  $s$  is exercised.
3. For any input file on which  $P'$  terminates normally but  $P$  does not,  $s$  is exercised at most as many times as  $s'$  is exercised.

Using program slicing, Bates and Horwitz categorize statements into execution classes. Statement  $s$  from  $P$  and  $s'$  from  $P'$  belong to the same execution class if and only if any test that exercises  $s$  will also exercise  $s'$ .

Now, a statement  $s'$  in  $P'$  is *affected* by the modification if and only if one of the following holds:

1. There is no corresponding statement  $s$  in  $P$ .
2. The behaviour of  $s'$  is not equivalent to the corresponding statement  $s$  in  $P$ .

Equivalent behaviour is determined by PDG slice isomorphism; if the PDG slices of two statements are isomorphic, then those statements share an equivalent behaviour. For each affected statement in  $P'$ , reusable test cases are selected based on the information retrieved from the identification stage.

While Bates and Horwitz's technique selects test cases for modified or newly added statements in  $P'$ , it does not select tests that exercise statements that are deleted from  $P$ , and therefore is not safe.

Binkley [96, 97] presented a technique based on SDG slicing, which extends Bates and Horwitz's intra-procedural selection technique to inter-procedural test case selection. Binkley introduced the

concept of *common execution patterns*, which corresponds to the equivalent execution patterns of Bates and Horwitz, to capture the multiple invocations of a procedure.

#### 4.8. Path analysis

Benedusi *et al.* applied path analysis for test case selection [61]. They construct *exemplar paths* from  $P$  and  $P'$  expressed in an algebraic expression. By comparing two sets of exemplar paths, they classified paths in  $P'$  as new, modified, cancelled or unmodified. Test cases and the paths they execute in  $P$  are known; therefore, they selected all the test cases that will traverse modified paths in  $P'$ .

One potential weakness of the path analysis approach of Benedusi *et al.* lies not in path analysis itself, but in the potentially over-specific definition of ‘modification’ used in the post-analysis selection phase. No test cases are selected for the paths that are classified as new or cancelled. However, new or cancelled paths denote modifications that represent differences between  $P$  and  $P'$ ; test cases that execute new or cancelled paths in  $P'$  may be modification-revealing. As presented, therefore, the path analysis approach is not safe.

#### 4.9. Modification-based technique

Chen *et al.* introduced a testing framework called TestTube, which utilizes a modification-based technique to select test cases [62]. TestTube partitions the SUT into *program entities*, and monitors the execution of test cases to establish connections between test cases and the program entities that they execute. TestTube also partitions  $P'$  into program entities, and identifies program entities that are modified from  $P$ . All the test cases that execute the modified program entities in  $P$  should be re-executed.

TestTube can be thought of as an extended version of the graph-walk approach. Both techniques identify modifications by examining the program source code, and select test cases that will execute the modified parts. TestTube extends the CDG-based graph-walk technique by introducing program entities that include both functions and entities that are not functions, i.e. variables, data types and pre-processor macros. Any test case that executes modified functions will be selected. Therefore, TestTube is a safe test case selection technique.

One weakness of TestTube is pointer handling. By including variable and data types as program entities, TestTube requires that all value creations and manipulations in a program can be inferred from source code analysis. This is only valid for languages without pointer arithmetic and type coercion. As a result, TestTube makes assumptions; for example, it assumes that all pointer arithmetics are well-bounded. If these assumptions do not hold then safety cannot be guaranteed.

#### 4.10. Firewall approach

Leung and White introduced and later implemented what they called a *firewall* technique for regression testing of system integration [63–66]. The main concept is to draw a firewall around the modules of the system that need to be retested. They categorize modules into the following categories:

- *No Change*: Module  $A$  has not been modified,  $NoCh(A)$ .
- *Only Code Change*: Module  $A$  has the same specification but its code has been modified,  $CodeCh(A)$ .
- *Spec Change*: module  $A$  has modified specifications,  $SpecCh(A)$ .

If a module  $A$  calls a module  $B$ , there exist 9 possible pairings between the states of  $A$  and  $B$ . The integration between  $A$  and  $B$  can be ignored for regression testing if  $NoCh(A) \wedge NoCh(B)$ , leaving 8 pairings. If both  $A$  and  $B$  are modified either in code or in specifications, the integration tests between  $A$  and  $B$  should be executed again as well as the unit tests of  $A$  and  $B$ ; this accounts for 4 of the remaining 8 pairings. The other 4 pairings are cases in which an unchanged module calls a changed module, or vice versa; these pairs form the boundary for integration testing, i.e. the so-called firewall.

By considering modules as the atomic entities, Leung and White maintained a very conservative approach to test case selection. If a module has been modified, any test case that tests the integration of the modified module should be selected. Therefore, all modification-traversing test cases will be selected. However, their technique may also select other test cases that execute the modified module, but are not modification-traversing in any way. Leung and White also noted that, in practice, the test suite for system integration is often not very reliable. The low reliability means that it is more likely that there may still exist a fault-revealing test case that does not belong to the test suite, and therefore cannot be selected. Note that it is always a risk that a fault-revealing test case exists outside the given test suite in any type of testing, not only in integration testing. What Leung and White pointed out was that such a risk can be higher in system integration testing due to the generally low quality of test suites.

The Firewall approach has been applied to Object-oriented programs [98–100] and GUIs [101]. Firewall approach has also been successfully applied to RTS for black-box Commercial Off-the-Shelf components. Zheng *et al.* applied the firewall technique of Leung and White based on the information extracted from the deployed binary code [102–105]. Skoglund and Runeson applied the firewall approach to a large-scale banking system [106].

#### 4.11. Cluster identification

Laski and Szemerédi presented a test case selection technique based on analysis of the CFG of the program under test [67]. Their technique identifies single-entry, single-exit subgraphs of CFG called clusters. Given a program  $P$  and its modified version  $P'$ ,

- each cluster in  $P$  encapsulates some modifications to  $P$ ,
- there is a unique cluster in  $P'$  that corresponds to the cluster in  $P$ , and
- when clusters in each graph are replaced by single nodes, there is a one-to-one correspondence between nodes in both graphs.

The CFGs of the original program and the modified program are reduced using a set of operators such as node collapse and node removal. During the process, if the counterpart of a node from the CFG of the original program cannot be found in the CFG of the modified program, this node is labelled as ‘MOD’, indicating a modification at the node. Eventually, all the modifications will be enclosed in one or more MOD cluster nodes. As with other test case selection techniques, their technique requires that the tester records the execution history of each test case in the test suite. Once clustering is completed, test case selection is performed by selecting all the test cases for which the corresponding execution path enters any of the MOD clusters.

The strength of the cluster identification technique is that it guarantees to select all modification-traversing test cases regardless of the type of the modification, i.e. addition or deletion of statements and control structures. However, since the clusters can encapsulate much larger areas of the SUT than the scope of actual modification, the technique may also select test cases that are not modification-traversing. In this sense the approach sacrifices precision in order to achieve safety.

#### 4.12. Design-based approach

Briand *et al.* presented a black-box, design level RTS approach for UML-based designs [68, 69]. Assuming that there is traceability between the design and regression test cases, it is possible to perform RTS of code-level test cases from the impact analysis of UML design models. Briand *et al.* formalized possible changes in UML models, and classified the relevant test cases into the categories defined by Leung and White [10]: obsolete, retestable and reusable. They implemented an automated impact analysis tool for UML and empirically evaluated it using both student projects and industrial case studies.

The results showed that the changes made to a model can have a widely variable impact on the resulting system, which, in turn, yields varying degrees of reduction of effort in terms of the number of selected test cases. However, Briand *et al.* noted that the automated impact analysis itself can be valuable, especially for very large systems, such as the cruise control and monitoring system they studied. The UML use-cases of the model of the system had 323 614 corresponding

test cases. UML-based models also have been considered by Dent *et al.* [107], Pilskalns *et al.* [108] and Farooq *et al.* [109] for RTS; Le Traon *et al.* [110] and Wu and Offutt [111] considered the use of UML models in the wider context of regression testing in general. Muccini *et al.* considered the RTS problem at the software architecture level, although they did not use UML for the representation [112, 113].

## 5. TEST CASE PRIORITIZATION

Test case prioritization seeks to find the ideal ordering of test cases for testing, so that the tester obtains maximum benefit, even if the testing is prematurely halted at some arbitrary point. The approach was first mentioned by Wong *et al.* [39]. However, in that work it was only applied to test cases that were already selected by a test case selection technique. Harrold and Rothermel [114, 115] proposed and evaluated the approach in a more general context.

For example, consider the test suite described in Table III. Note that the example depicts an ideal situation in which fault-detection information is known. The goal of prioritization is to maximize early fault detection. It is obvious that the ordering A-B-C-D-E is inferior to B-A-C-D-E. In fact, any ordering that starts with the execution of C-E is superior to those that do not, because the subsequence C-E detects faults as early as possible; should testing be stopped prematurely, this ensures that the maximum possible fault coverage will have been achieved.

Note that the problem definition concerns neither versions of the program under test, nor exact knowledge of modifications. Ideally, the test cases should be executed in the order that maximizes early fault detection. However, fault-detection information is typically not known until the testing is finished. In order to overcome the difficulty of knowing which tests reveal faults, test case prioritization techniques depend on surrogates, hoping that early maximization of a certain chosen surrogate property will result in maximization of earlier fault detection. In a controlled-regression-testing environment, the result of prioritization can be evaluated by executing test cases according to the fault-detection rate.

### 5.1. Coverage-based prioritization

Structural coverage is a metric that is often used as the prioritization criterion [115–121]. The intuition behind the idea is that early maximization of structural coverage will also increase the chance of early maximization of fault detection. Therefore, while the goal of test case prioritization remains that of achieving a higher fault-detection rate, prioritization techniques actually aim to maximize early coverage.

Rothermel *et al.* reported empirical studies of several prioritization techniques [115, 121]. They applied the same algorithm with different fault-detection rate surrogates. The considered surrogates were: branch-total, branch-additional, statement-total, statement-additional, Fault Exposing Potential (FEP)-total and FEP-additional.

The branch-total approach prioritizes test cases according to the number of branches covered by individual test cases, while branch-additional prioritizes test cases according to the additional

Table III. Example test suite with fault-detection information, taken from Elbaum *et al.* [116]. It is clearly beneficial to execute test case C first, followed by E.

Test case	Fault revealed by test case									
	1	2	3	4	5	6	7	8	9	10
A	x				x					
B	x				x	x	x			
C	x	x	x	x	x	x	x			
D					x					
E							x	x	x	

number of branches covered by individual test cases. The statement-total and statement-additional approaches apply the same idea to program statements, rather than branches. Algorithmically, ‘total’ approaches are essentially instances of greedy algorithms, whereas ‘additional’ approaches are essentially instances of additional greedy algorithms.

The FEP of a test case is measured using program mutation. Program mutation introduces a simple syntactic modification to the program source, producing a mutant version of the program [122]. This mutant is said to be *killed* by a test case if the test case reveals the difference between the original program and the mutant. Given a set of mutants, the mutation score of a test case is the ratio of mutants that are killed by the test case to the total killable mutants. The FEP-total approach prioritizes test cases according to the mutation score of individual test cases, while the FEP-additional approach prioritizes test cases according to the additional increase in mutation score provided by individual test cases. Note that FEP criterion can be constructed to be at least as *strong* as structural coverage; to kill a mutant, a test case not only needs to achieve the coverage of the location of mutation but also to execute the mutated part with a set of test inputs that can kill the mutant. In other words, coverage is necessary but not sufficient to kill the mutants.

It is important to note that all the ‘additional’ approaches may reach 100% realization of the utilized surrogate before every test case is prioritized. For example, achieving 100% branch coverage may not require all the test cases in the test suite, in which case none of the remaining test cases can increase the branch coverage. Rothermel *et al.* reverted to the ‘total’ approach once such a condition is met.

The results were evaluated using the Average Percentage of Fault-Detection (APFD) metric. Higher APFD values denote faster fault-detection rates. When plotting the percentage of detected faults against the number of executed test cases, APFD can be calculated as the area below the plotted line. More formally, let  $T$  be the test suite containing  $n$  test cases and let  $F$  be the set of  $m$  faults revealed by  $T$ . For ordering  $T'$ , let  $TF_i$  be the order of the first test case that reveals the  $i$ th fault. The APFD value for  $T'$  is calculated as follows [123]:

$$APFD = 1 - \frac{TF_1 + \dots + TF_m}{nm} + \frac{1}{2n}$$

Note that, while APFD is commonly used to evaluate test case prioritization techniques, it is not the aim of test case prioritization techniques to maximize APFD. Maximization of APFD would be possible only when every fault that can be detected by the given test suite is already known. This would imply that all test cases have been already executed, which would nullify the need to prioritize. APFD is computed after the prioritization only to evaluate the performance of the prioritization technique.

Rothermel *et al.* compared the proposed prioritization techniques to random prioritization, optimal prioritization and no prioritization, using the Siemens suite programs. Optimal prioritization is possible because the experiment was performed in a controlled environment, i.e. the faults were already known. The results show that all the proposed techniques produce higher APFD values than random or no prioritization. The surrogate with the highest APFD value differed between programs, suggesting that there is no single best surrogate. However, on average across the programs, the FEP-additional approach performed most effectively, producing APFD value of 74.5% compared to the 88.5% of the optimal approach. It should still be noted that these results are dependent on many factors, including the types of faults used for evaluation and types of mutation used for FEP, limiting the scope for generalization.

Elbaum *et al.* extended the empirical study of Rothermel *et al.* by including more programs and prioritization surrogates [116]. Among the newly introduced prioritization surrogates, function-coverage and function-level FEP enabled Elbaum *et al.* to study the effects of granularity on prioritization. Function-coverage of a test case is calculated by counting the number of functions that the test case executes. Function-level FEP is calculated, for each function  $f$  and each test case  $t$ , by summing the ratio of mutants in  $f$  killed by  $t$ . Elbaum *et al.* hypothesized that approaches with coarser granularity would produce lower APFD values, which was confirmed statistically.

Jones and Harrold applied the greedy-based prioritization approach to Modified Condition/Decision Coverage (MC/DC) criterion [124]. MC/DC is a ‘stricter form’ of branch coverage; it

requires execution coverage at condition level. A condition is a Boolean expression that cannot be factored into simpler Boolean expressions. By checking each condition in decision predicates, MC/DC examines whether each condition independently affects the outcome of the decision [125]. They presented an empirical study that contained only an execution time analysis of the prioritization technique and not an evaluation based on fault-detection rate.

Srivastava and Thiagarajan combined the greedy-based prioritization approach with RTS [126]. They first identified the modified code blocks in the new version of the SUT by comparing its binary code to that of the previous version. Once the modified blocks are identified, test case prioritization is performed using greedy-based prioritization, but only with respect to the coverage of modified blocks.

Do and Rothermel applied coverage-based prioritization techniques to the JUnit testing environment, a popular unit testing framework [127]. The results showed that prioritized execution of JUnit test cases improved the fault-detection rate. One interesting finding was that the random prioritization sometimes resulted in an APFD value higher than the *untreated* ordering, i.e. the order of creation. When executed in the order of creation, newer unit tests are executed later. However, it is the newer unit tests that have a higher chance of detecting faults. The empirical results showed that random prioritization could exploit this weakness of untreated ordering in some cases.

Li *et al.* applied various meta-heuristics for test case prioritization [128]. They compared random prioritization, a hill climbing algorithm, a genetic algorithm, a greedy algorithm, the additional greedy algorithm and a two-optimal greedy algorithm. The greedy algorithm corresponds to the *total* approaches outlined above, whereas the additional greedy algorithm corresponds to the *additional* approaches outlined above. The two-optimal greedy is similar to the greedy algorithm except that it considers two candidates at the same time rather than a single candidate for the next order. They considered the Siemens suite programs and the program space, and evaluated each technique based on Average Percentage of Block Coverage (APBC) instead of APFD. The results showed that the additional greedy algorithm is the most efficient in general.

### 5.2. Interaction testing

Interaction testing is required when the SUT involves multiple combinations of different components. A common example would be configuration testing, which is required to ensure that the SUT executes correctly on different combinations of environment, such as different operating systems or hardware options. Each component that can be changed is called a *factor*; the number of choices for each factor is called the *level* of the corresponding factor. As the number of factors and levels of each factor increase, exhaustive testing of all possible combinations of factors becomes infeasible as it requires an exponentially large test suite.

Instead of testing exhaustively, *pairwise* interaction testing requires only that every individual pair of interactions between different factors are included at least once in the testing process. The reduction grows larger as more factors and levels are involved. More formally, the problem of obtaining interaction testing combinations can be expressed as the problem of obtaining a *covering array*,  $CA(N; t, k, v)$ , which is an array with  $N$  rows and  $k$  columns;  $v$  is the number of levels associated with each factor, and  $t$  is the *strength* of the interaction coverage (2 in the case of pairwise interaction testing).

One important research direction in interaction testing is how to efficiently generate an interaction test suite with high interaction coverage. This shares the same basic principles of test case prioritization. For example, the greedy approach aims to find, one by one, the ‘next’ test case that will most increase the  $k$ -way interaction coverage [129, 130], which resembles the greedy approach to test case prioritization. However, the similarities are not just limited to the generation of interaction test suites. Bryce and Colbourn assume that testers may value certain interactions higher than others [131, 132]. For example, an operating system with a larger user base may be more important than one with a smaller user base. After weighting each level value for each factor, they calculate the combined benefit of a given test by adding the weights of each level value selected for the test. They present a Deterministic Density Algorithm (DDA) that prioritizes interaction

tests according to their combined benefit. Qu *et al.* compared different weighting schemes used for prioritizing covering arrays [133, 134].

Bryce and Memon also applied the principles of interaction coverage to the test case prioritization of Event-Driven Software (EDS) [135]. EDS takes sequences of events as input, changes state and outputs new event sequences. A common example would be GUI-based programs. Bryce and Memon interpreted  $t$ -way interaction coverage as sequences that contain different combinations of events over  $t$  unique GUI windows. Interaction coverage-based prioritization of test suites was compared to different prioritization techniques such as unique event coverage (the aim is to cover as many unique events as possible, as early as possible), longest to shortest (execute the test case with the longest event sequence first) and shortest to longest (execute the test case with the shortest event sequence first). The empirical evaluation showed that interaction coverage-based testing of EDS can be more efficient than the other techniques, provided that the original test suite contains higher interaction coverage. Note that Bryce and Memon did not try to generate additional test cases to improve interaction coverage; they only considered permutations of existing test cases.

### 5.3. Prioritization approaches based on other criteria

While the majority of existing prioritization literature concerns structural coverage in some form or another, there are prioritization techniques based on other criteria [136–139].

*Distribution-based Approach:* Leon and Podgurski introduced distribution-based filtering and prioritization [136]. Distribution-based techniques minimize and prioritize test cases based on the distribution of the profiles of test cases in the multi-dimensional profile space. Test case profiles are produced by the dissimilarity metric, a function that produces a real number representing the degree of dissimilarity between two input profiles. Using this metric, test cases can be clustered according to their similarities. The clustering can reveal some interesting information. For example:

- Clusters of similar profiles may indicate a group of redundant test cases
- Isolated clusters may contain test cases inducing unusual conditions that are perhaps more likely to cause failures
- Low density regions of the profile space may indicate uncommon usage behaviours

The first point is related to reduction of effort; if test cases in a cluster are indeed very similar, it may be sufficient to execute only one of them. The second and third points are related to fault-proneness. Certain unusual conditions and uncommon behaviours may tend to be harder to reproduce than more common conditions and behaviours. Therefore, the corresponding parts of the program are likely to be tested less than other, more frequently used parts of the program. Assigning a high priority to test cases that execute these unusual behaviours may increase the chance of early fault detection. A good example might be exception handling code.

Leon and Podgurski developed new prioritization techniques that combine coverage-based prioritization with distribution-based prioritization. This hybrid approach is based on the observation that *basic coverage maximization* performs reasonably well compared to *repeated coverage maximization*. Repeated coverage maximization refers to the prioritization technique of Elbaum *et al.* [116], which, after realizing 100% coverage, repeatedly prioritizes test cases starting from 0% coverage again. In contrast, basic coverage maximization stops prioritizing when 100% coverage is achieved. Leon and Podgurski observed that the fault-detection rate of repeated coverage maximization is not as high as that of basic coverage maximization. This motivated them to consider a hybrid approach that first prioritizes test cases based on coverage, then switches to distribution-based prioritization once the basic coverage maximization is achieved. They considered two different distribution-based techniques. The one-per-cluster approach samples one test case from each cluster, and prioritizes them according to the order of cluster creation during the clustering. The failure-pursuit approach behaves similarly, but it adds the  $k$  closest neighbours of any test case that finds a fault. The results showed that the distribution-based prioritization techniques could outperform repeated coverage maximization.

*Human-based Approach:* Tonella *et al.* combined Case-Based Reasoning (CBR) with test case prioritization [137]. They utilized a machine learning technique called *boosting*, which is a

framework to combine simple learners into a single, more general and effective learner [140]. They adopted a boosting algorithm for ranking learning called *Rankboost* [141]. The algorithm takes a test suite,  $T$ , an initial prioritization index,  $f$ , and a set of pairwise priority relations between test cases,  $\Phi$ , as input. The pairwise priority relation is obtained from comparisons of test cases made by the human tester. The output is a ranking function  $H : T \rightarrow \mathbb{R}$  such that, with test cases  $t_1$  and  $t_2$ ,  $t_1 \prec t_2$  if  $H(t_1) > H(t_2)$ . The ranking function  $H$  is then used to prioritize test cases.

They used the statement coverage metric and the cyclomatic complexity computed for the functions executed by test cases as the initial prioritization index. The test suite of the space program was considered. In order to measure the human effort required for the learning process, different test suite sizes were adopted, ranging from 10 to 100 test cases. The results were compared to other prioritization techniques including optimal ordering, random prioritization, statement coverage prioritization and additional statement coverage prioritization (the latter two correspond to statement-total and statement-additional respectively).

The results showed that, for all test suite sizes, the CBR approach was outperformed only by the optimal ordering. The number of pairwise relations entered manually showed a linear growth against the size of test suites. Tonella *et al.* reported that for test suites of space with fewer than 60 test cases, the CBR approach can be more efficient than other prioritization techniques with limited human effort. Note that empirical evaluation was performed based on an *ideal* user model, i.e. it was assumed that the human tester always makes the correct decision when comparing test cases. One notable weakness of this approach was that it did not scale well. The input from the human tester becomes inconsistent beyond a certain number of comparisons, which in turn limits the size of the learning samples for CBR.

Yoo *et al.* tried to improve the scalability of human-based prioritization approaches by combining pairwise comparisons of test cases with a clustering technique [138]. While the prioritization is still based on the comparisons made by the human tester, the tester is presented with clusters of similar test cases instead of individual test cases. The prioritization between clusters (*inter-cluster* prioritization) is, therefore, performed by the tester. However, the prioritization within each cluster (*intra-cluster* prioritization) is performed based on coverage. After both layers of prioritization are complete, the final ordering of test cases is determined by selecting the test case with the highest priority, determined by intra-cluster prioritization, from the next cluster in the order determined by inter-cluster prioritization, until all the clusters are empty. This is called the Interleaved Clusters Prioritization (ICP) technique.

With the use of clustering, Yoo *et al.* were able to reduce the size of the prioritization problem so that they could apply a more expensive pairwise approach called Analytic Hierarchy Process (AHP). AHP is a pairwise comparison technique developed by the Operations Research community [142] and has been successfully applied to Requirements Engineering [143]. The combination of AHP and ICP has been empirically evaluated for programs and test suites of various sizes, using a more realistic user model (with errors). The results showed that this combination of techniques can be much more effective than coverage-based prioritization. One surprising finding was that sometimes, an error rate higher than 50%, i.e. the human tester making wrong comparisons half the time, did not prevent this technique from achieving higher APFD than coverage-based prioritization. Yoo *et al.* explained this unexpected finding by showing that a certain amount of improvement derived from the effect of clustering. This confirms the argument of Leon and Podgurski about the benefits of distribution-based approach [136]; the clustering sometimes enables the early execution of a fault-revealing test case that would have been assigned low priority due to its low contribution to code coverage.

*Probabilistic Approach:* Kim and Porter proposed a history-based approach to prioritize test cases that are already selected by RTS [144]. If the number of test cases selected by an RTS technique is still too large, or if the execution costs are too high, then the selected test cases may have to be further prioritized. Since the relevance to the recent change in SUT is assumed by the use of an RTS technique, Kim *et al.* focus on the execution history of each test case, borrowing from statistical quality control. They define the probabilities of each test case  $tc$  to be selected at time  $t$  as  $P_{tc,t}(H_{tc}, \alpha)$ , where  $H_{tc}$  is a set of  $t$  timed observations  $\{h_1, \dots, h_t\}$  drawn from previous

runs of  $tc$  and  $\alpha$  is a smoothing constant. Then the probability  $P_{tc,t}(H_{tc}, \alpha)$  is defined as follows:

$$\begin{aligned} P_0 &= h_1 \\ P_k &= \alpha h_k + (1 - \alpha) P_{k-1} \quad (0 \leq \alpha \leq 1, k \geq 1) \end{aligned}$$

Different definitions of  $H_{tc}$  result in different prioritization approaches. For example, Kim *et al.* define Least Recently Used (LRU) prioritization by using test case execution history as  $H_{tc}$  with  $\alpha$  value that is as close to 0 as possible. The empirical evaluation showed that the LRU prioritization approach can be competitive in a severely constrained testing environment, i.e. when it is not possible to execute all test cases selected by an RTS technique.

Mirarab and Tahvildari took a different probabilistic approach to test case prioritization using Bayesian Networks [145]. The Bayesian Network model is built upon changes in program elements, fault proneness of program elements and probability of each test case to detect faults. Mirarab and Tahvildari extended the approach by adding a feedback route to update the Bayesian Network as prioritization progresses [146]. For example, if a test case covers a set of program elements, the probability of selecting other test cases that cover the same elements will be lowered. Note that this corresponds to the ‘additional’ approach described by Rothermel *et al.* [115, 121].

*History-based Approach:* Sherriff *et al.* presented a prioritization technique based on association clusters of software artefacts obtained by a matrix analysis called singular value decomposition [147]. The prioritization approach depends on three elements: association clusters, relationship between test cases and files and a modification vector. Association clusters are generated from a change matrix using SVD; if two files are often modified together as a part of a bug fix, they will be clustered into the same association cluster. Each file is also associated with test cases that affect or execute it. Finally, a new system modification is represented as a vector in which the value indicates whether a specific file has been modified. Using the association clusters and the modification vector, it is then possible to assign each file with a priority that corresponds to how closely the new modification matches each test case. One novel aspect of this approach is that any software artefact can be considered for prioritization. Sherriff *et al.* noted that the faults that are found in non-source files, such as media files or documentation, can be as severe as those found in source code.

*Requirement-based Approach:* Srikanth *et al.* presented requirement-based test case prioritization [139]. Test cases are mapped to software requirements that are tested by them, and then prioritized by various properties of the mapped requirements, including customer-assigned priority and implementation complexity. One potential weakness of this approach is the fact that requirement properties are often estimated and subjective values. Krishnamoorthi and Sahaaya developed a similar approach with additional metrics [148].

*Model-based Approach:* Korel *et al.* introduced a model-based prioritization approach [149–151]. Their initial approach was called *selective* prioritization, which was strongly connected to RTS [149]. Test cases were classified into a high priority set,  $TS_H$ , and a low priority set,  $TS_L$ . They defined and compared different definitions of high and low priority test cases, but essentially a test case is assigned high priority if it is relevant to the modification made to the model. The initial selective prioritization process consists of the random prioritization of  $TS_H$  followed by the random prioritization of  $TS_L$ . Korel *et al.* developed more sophisticated heuristics based on the dependence analysis of the models [150, 151].

*Other Approaches:* The use of mutation score for test case prioritization has been analysed by Rothermel *et al.* along with other structural coverage criteria [115, 121]. Hou *et al.* considered interface-contract mutation for the regression testing of component-based software and evaluated it with the *additional* prioritization technique [152].

Sampath *et al.* presented the prioritization of test cases for web applications [153]. The test cases are, in this case, recorded user sessions from the previous version of the SUT. Session-based test cases are thought to be ideal for testing web applications because they tend to reflect the actual usage patterns of real users, thereby making for realistic test cases. They compared different criteria for prioritization such as the number of HTTP requests per test case, coverage of parameter values,

frequency of visits for the pages recorded in sessions and the number of parameter values. The empirical evaluations showed that prioritized test suites performed better than randomly ordered test suites, but also that there is not a single prioritization criterion that is always best. However, the 2-way parameter-value criterion, the prioritization criterion that orders tests to cover all pairwise combinations of parameter-values between pages as soon as possible, showed the highest APFD value for 2 out of 3 web applications that were studied.

Fraser and Wotawa introduced a model-based prioritization approach [154]. Their prioritization technique is based on the concept of *property relevance* [155]. A test case is relevant to a model property if it is theoretically possible for the test case to violate the property. The relevance relation is obtained by the use of a model-checker, which is used as the input to the greedy algorithm. While they showed that property-based prioritization can outperform coverage-based prioritization, they noted that the performance of property-based prioritization is heavily dependent on the quality of the model specification.

A few techniques and analyses used for test suite minimization or RTS problem have also been applied to test case prioritization. Rummel *et al.* introduced a prioritization technique based on data-flow analysis by treating each *du* pair as a testing requirement to be covered [156]. Smith *et al.* introduced a prioritization technique based on a call-tree model, which they also used for test suite minimization [26]. They prioritized test cases according to the number of call-tree paths covered by each test case. Jeffrey and Gupta prioritized test cases using relevant slices [157], which was also used for RTS [54]. Each test case was associated with output statements, from which relevant slices were calculated. Then test cases were prioritized according to the sum of two elements: the size of the corresponding relevant slice and the number of statements that are executed by the test case but do not belong to the relevant slice. Both elements were considered to correlate to the chance of revealing a fault introduced by a recent change.

#### 5.4. Cost-aware test case prioritization

Unlike test suite minimization and RTS, the basic definition of test case prioritization does not involve filtering out test cases, i.e. it is assumed that the tester executes the entire test suite following the order given by the prioritization technique. This may not be feasible in practice due to limited resources. A number of prioritization techniques addressed this problem of the need to be cost-aware [23, 24, 118, 158].

With respect to cost-awareness, the basic APFD metric has two limitations. First, it considers all faults to be equally severe. Second, it assumes that every test case costs the same in resources. Elbaum *et al.* extended the basic APFD metric to APFD<sub>c</sub> so that the metric incorporates not just the rate of fault detection but also the severity of detected faults and the expense of executing test cases [118]. An ordering of test cases according to the APFD<sub>c</sub> metric detects more severe faults at a lower cost. More formally, let  $T$  be the set of  $n$  test cases with costs  $t_1, \dots, t_n$ , and let  $F$  be the set of  $m$  faults with severity values  $f_1, \dots, f_m$ . For ordering  $T'$ , let  $TF_i$  be the order of the first test case that reveals the  $i$ th fault. APFD<sub>c</sub> of  $T'$  is calculated as following:

$$APFD_c = \frac{\sum_{i=1}^m (f_i \times (\sum_{j=TF_i}^n t_j - \frac{1}{2}t_{TF_i}))}{\sum_{i=1}^n t_i \times \sum_{i=1}^m f_i}$$

Elbaum *et al.* applied random ordering, additional statement coverage prioritization, additional function coverage prioritization and additional fault index prioritization techniques to space, which contains faults discovered during the development stage. They adopted several different models of test case cost and fault severity, including uniform values, random values, normally distributed values and models taken from the Mozilla open source project. The empirical results were achieved by synthetically adding cost severity models to space. This enabled them to observe the impact of different severity and cost models. They claimed two practical implications. With respect to test case cost, they proposed the use of many small test cases rather than a few large test cases. Clearly the number of possible prioritizations is higher with a test suite that contains many small test cases, compared to one with a small number of large test cases. It was also claimed that having different models of fault severity distribution can also impact the efficiency of testing.

This is true only when the prioritization technique considers the fault-detection history of previous tests.

Elbaum *et al.* compared two different severity distribution models: linear and exponential. In the linear model, the severity values grow linearly as the severity of faults increases, whereas they grow exponentially in the exponential model. If the previous fault-detection history correlates to the fault detection capability of the current iteration of testing, the exponential model ensures that test cases with a history of detecting more severe faults are executed earlier.

Walcott *et al.* presented a time-aware prioritization technique [23]. Time-aware prioritization does not prioritize the entire test suite; it aims to produce a subset of test cases that are prioritized and can be executed within the given time budget. More formally, it is defined as follows:

*Given:* A test suite,  $T$ , the collection of all permutations of elements of the power set of permutations of  $T$ ,  $\text{perms}(2^T)$ , the time budget,  $t_{\max}$ , a time function  $\text{time} : \text{perms}(2^T) \rightarrow \mathbb{R}$  and a fitness function  $\text{fit} : \text{perms}(2^T) \rightarrow \mathbb{R}$ :

*Problem:* Find the test tuple  $\sigma_{\max} \in \text{perms}(2^T)$  such that  $\text{time}(\sigma_{\max}) \leq t_{\max}$  and  $\forall \sigma' \in \text{perms}(2^T)$ , where  $\sigma_{\max} \neq \sigma'$  and  $\text{time}(\sigma') \leq t_{\max}$ ,  $\text{fit}(\sigma_{\max}) > \text{fit}(\sigma')$ .

Intuitively, a time-aware prioritization technique selects and prioritizes test cases at the same time so that the produced ordered subset yields higher fault-detection rates within the given time budget. Walcott *et al.* utilized a genetic algorithm, combining selection and prioritization into a single fitness function. The selection component of the fitness function is given higher weighting so that it dominates the overall fitness value produced. The results of the genetic algorithm were compared to random ordering, reverse ordering and the optimal ordering. The results showed that time-aware prioritization produces higher rates of fault detection compared to random, initial and reverse ordering. However, Walcott *et al.* did not compare the time-aware prioritization to the existing, non-time-aware prioritization techniques. Note that non-time-aware prioritization techniques can also be executed in ‘time-aware’ manner by stopping the test when the given time budget is exhausted.

While Yoo and Harman studied test suite minimization [24], their multi-objective optimization approach is also relevant to the cross-cutting concern of cost-awareness. By using multi-objective optimization heuristics, they obtained a Pareto-frontier which represents the trade-offs between the different criteria, including cost. When there is a constraint on cost, the knowledge of Pareto-frontier can provide the tester with more information to achieve higher coverage. The tester can then prioritize the subset selected by observing the Pareto-frontier.

The cost-constraint problem has also been analysed using ILP [159, 160]. Hou *et al.* considered the cost-constraint in web service testing [159]. Users of web services are typically assigned with a usage quota; testing a system that uses web services, therefore, has to consider the remaining quota for each web service. The ILP approach was later analysed in more generic context using execution time of each test as a cost factor [160].

Do and Rothermel studied the impact of time constraints on the cost-effectiveness of existing prioritization techniques [158]. In total, six different prioritization approaches were evaluated: original order, random order, total block coverage, additional block coverage, Bayesian Network approach without feedback, Bayesian Network approach with feedback. They considered four different time constraints, each of which allows {25%, 50%, 75%, 100%} of time required for the execution of all test cases. Each prioritization approach was evaluated under these constraints using a cost-benefit model. The results showed that, although time constraints affect techniques differently, it is always beneficial to adopt some prioritization when under time constraints. The original ordering was always affected the most severely.

## 6. META-EMPIRICAL STUDIES

Recently, the meta-empirical study of regression testing techniques has emerged as a separate subject in its own right. It addresses cross-cutting concerns such as cost-benefit analysis of regression testing techniques and the studies of evaluation methodology for these techniques. Both studies

seek to provide more confidence in efficiency and effectiveness of regression testing techniques. Work in these directions is still in the early stages compared to the bodies of work available for minimization, selection or prioritization techniques. However, it is believed that these studies will make significant contributions towards the technology transfer.

Empirical evaluation of any regression testing technique is inherently a *post-hoc* process that assumes the knowledge of a set of known faults. Without the *a priori* knowledge of faults, it would not be possible to perform a controlled experiment comparing different regression testing techniques. This poses a challenge to the empirical evaluation of techniques, since the availability of fault data tends to be limited [161].

Andrews *et al.* performed an extensive comparison between real faults and those seeded by mutation [161]. One concern when using mutation faults instead of real faults is that there is no guarantee that the detection of mutation faults can be an accurate predictor of the detection of real faults. After considering various statistical data such as the ratio and distribution of fault detection, Andrews *et al.* concluded that mutation faults can indeed provide a good indication of the fault detection capability of the test suite, assuming that mutation operators are carefully selected and equivalent mutants are removed. However, they also note that, while mutation faults were not easier to detect than real faults, they were also not harder to detect. Do and Rothermel extended this study by focusing the comparison on the result of test case prioritization techniques [162, 163]. Here, they considered whether evaluating prioritization techniques against mutation faults and seeded faults differs. Based on the comparison of these two evaluation methods, it was concluded that mutation faults can be safely used in place of real or hand-seeded faults.

Although it was not their main aim, Korel *et al.* made an important contribution to the empirical evaluation methodology of regression testing techniques through the empirical evaluation of their prioritization techniques [149–151]. They noted that, in order to compare different prioritization techniques in terms of their rate of fault detection, they need to be evaluated using all possible prioritized sequences of test cases that may be generated by each technique. Even deterministic prioritization algorithms, such as the greedy algorithm, can produce different results for the same test suite if some external factors change; for example, if the ordering of the initial test suite changes, there is a chance that the greedy algorithm will produce a different prioritization result. Korel *et al.* argued, therefore, that the rate of fault detection should be measured in average across all possible prioritized sequences. They introduced a new metric called Most Likely average Position, which measures the average relative position of the first test case that detects a specific fault.

Elbaum *et al.* extended the empirical studies of prioritization techniques with the Siemens suite and space [116] by performing statistical analysis of the variance in APFD [117]. The APFD values were analysed against various program, change and test metrics. Program metrics included mean number of executable statements, mean function size across all functions, etc. Change metrics included number of functions with at least one changed statement, number of statements inserted or deleted, etc. Test metrics included number of tests in the test suite, percentage of tests reaching a changed function, mean number of changed functions executed by a test over a test suite, etc. The aim was to identify the source of variations in results. Elbaum *et al.* reported that the metrics that reflected normalized program characteristics (such as mean function size across the program) and characteristics of test suites in relation to programs (such as mean percentage of functions executed by a test over a test suite) were the primary contributors to the variances in prioritization. While they reported that this finding was not the anticipated one, it showed that the prioritization results are the product of closely coupled interactions between programs under test, changes and test suites.

Empirical evaluation of different techniques can benefit from a shared evaluation framework. Rothermel and Harrold presented a comparison framework for RTS techniques [45], which was used to compare different RTS techniques [1]. While minimization and prioritization techniques lack such a framework, certain metrics have been used as a *de facto* standard evaluation framework. Rate of reduction in size and rate of reduction in fault-detection capability have been widely used to evaluate test suite minimization techniques [12, 18–21, 25, 26, 38–40, 43]. Similarly, APFD [116] has been widely used to evaluate prioritization techniques [23, 116, 117, 120, 121, 123, 127, 136–138, 145, 146, 154, 156, 157, 159, 160, 162–165].

Rothermel *et al.* studied the impact of test suite granularity and test input grouping on the cost-effectiveness of regression testing [120, 165]. They first introduced the concept of *test grains*, which is the smallest unit of test input that is executable and checkable. Test cases are constructed by grouping test grains. Based on this, they defined test suite granularity as the number of test grains in a test case, and test input grouping as the way test grains are added to each test case, e.g. randomly or grouped by their functionality. They reported that having a coarse-grained test suite did not significantly compromise the fault-detection capability of the test suite, but resulted in decreased total execution time. The savings in execution time can be explained by the fact that a coarse-grained test suite contains fewer test cases, thereby reducing the set-up time and other overheads that occur between execution of different test cases. However, they did not consider the cost of the test oracle. It is not immediately obvious whether the cost of a test oracle would increase or decrease as the test suite granularity increases. This oracle cost could affect the overall cost-effectiveness.

Kim *et al.* studied the impact of test application frequency on the cost-effectiveness of RTS techniques [166, 167]. Their empirical studies showed that the frequency of regression test application has a significant impact on the cost-effectiveness of RTS techniques. They reported that RTS techniques tend to be more cost-effective when the frequency of test application is high. It implies that only a small amount of changes are made between tests, which makes RTS more effective. However, as intervals between tests grow, changes are accumulated and RTS techniques tend to select more and more test cases, resulting in low cost-effectiveness. One interesting finding is that, as intervals between tests grow, random re-testing tends to work very well. With small testing intervals, the random approach fails to focus on the modification. As testing intervals increase, more parts of SUT need to be re-tested, improving the effectiveness of the random approach. Elbaum *et al.* studied the impacts of changes in terms of the quantitative nature of modifications [164]. They investigated how the cost-effectiveness of selection and prioritization techniques is affected by various change metrics such as percentage of changed lines of code (LOC), average number of LOC changed per function, etc. Their empirical analysis confirmed that the differences in these metrics can make a significant impact on the cost-effectiveness of techniques. However, they also reported that simple size of change, measured in LOC, was not a predominant factor in determining the cost-effectiveness of techniques. Rather, it was the distribution of changes and the ability of test cases to reach these changes.

Elbaum *et al.* also presented a technique for selecting the most cost-effective prioritization technique [168]. They applied a set of prioritization techniques to the same set of programs, and analysed the resulting APFD metric values. Different techniques perform best for different programs; they applied the classification tree technique to predict the best-suited technique for a program. Note that the term ‘cost-effectiveness’ in this work means the efficiency of a prioritization technique measured by the APFD metric; the computational cost of applying these techniques was not considered.

Rosenblum and Weyuker introduced a coverage-based cost-effective predictor for RTS techniques [169]. Their analysis is based on the coverage relation between test cases and program entities. If each program entity has a uniformly distributed probability of being changed in the next version, it is possible to predict the average number of test cases to be selected by a safe RTS technique using coverage relation information. They evaluated their predictor with the TestTube RTS tool [62], using multiple versions of the KornShell [170] and an I/O library for Unix, SFIO [171], as subjects. Their predictor was reasonably accurate; for example, it predicted an average of 87.3% of the test suite to be selected for KornShell, when TestTube selected 88.1%. However, according to the cost model of Leung and White [172], the cost of coverage analysis for RTS per test case was greater than the cost of execution per test case, indicating that TestTube was not cost-effective. Harrold *et al.* introduced an improved version of the cost-effective predictor of Rosenblum *et al.* for more accurate cost-effectiveness prediction of version-specific RTS [173]. They evaluated their predictor using TestTube and another RTS tool, DejaVu [57].

Modelling the cost-effectiveness of regression testing techniques has emerged as a research topic. This is motivated by the observation that any analysis of cost-effectiveness should depend

on some model. Leung and White introduced an early cost-model for regression testing strategies and compared the cost models of the *retest-all* strategy and the selective retesting strategy [172]. Malishevsky *et al.* presented detailed models of cost-benefit trade-offs for regression testing techniques [119]. They applied their models to the regression testing of bash, a popular Unix shell [174], with different ratio values of  $f/(e+c)$ , where  $f$  is the cost of omitting one fault,  $e$  is the additional cost per test and  $c$  is the result-validation cost per test. The results implied that if a regression testing technique does not consider  $f$ , it may overestimate the cost-effectiveness of a given technique. The cost model of Malishevsky *et al.* has been extended and evaluated against the prioritization of JUnit test cases [175]. Smith and Kapfhammer studied the impact of the incorporation of cost into test suite minimization [176]. Existing minimization heuristics including HGS [12], delayed greedy [18] and 2-optimal greedy algorithm [128] were extended to incorporate the execution cost of each test case. Do and Rothermel considered the impact of time constraints on selection and prioritization techniques across multiple consecutive versions of subject programs to incorporate software life-cycle factors into the study [177].

Reflecting the complexity of regression testing process, cost-effectiveness models often need to be sophisticated in order to incorporate multiple variables [119, 175–177]. However, complexity can be a barrier to uptake. Do and Rothermel introduced an approach based on statistical sensitivity analysis to simplify complicated cost models [178]. Their approach fixed certain cost factors that were deemed to be the least significant by the sensitivity analysis. The empirical evaluation showed that, while certain levels of simplification can still preserve the accuracy of the model, over-simplification may be risky.

## 7. SUMMARY & DISCUSSION

### 7.1. Analysis of current global trends in the literature

This paper has produced a survey of 159 papers on test suite minimization, RTS and test case prioritization. This number includes papers on methodologies of empirical evaluation and comparative studies. Data summarizing the results in these papers are shown in Tables AI–AIV in the Appendix. Note that the last category consists of papers on cross-cutting concerns for empirical studies, such as methodologies of empirical evaluation and analyses of cost-effectiveness, as well as purely comparative studies and surveys. Figure 2 plots the number of surveyed papers for each year since 1977, when Fischer published his paper on RTS using a linear programming approach [47]. The observed trend in the number of publications shows that the field continues to grow.

Figure 3 shows the chronological trend in the number of studies for each of the topics in this paper. In this figure, the papers have been classified into four different categories. The first three

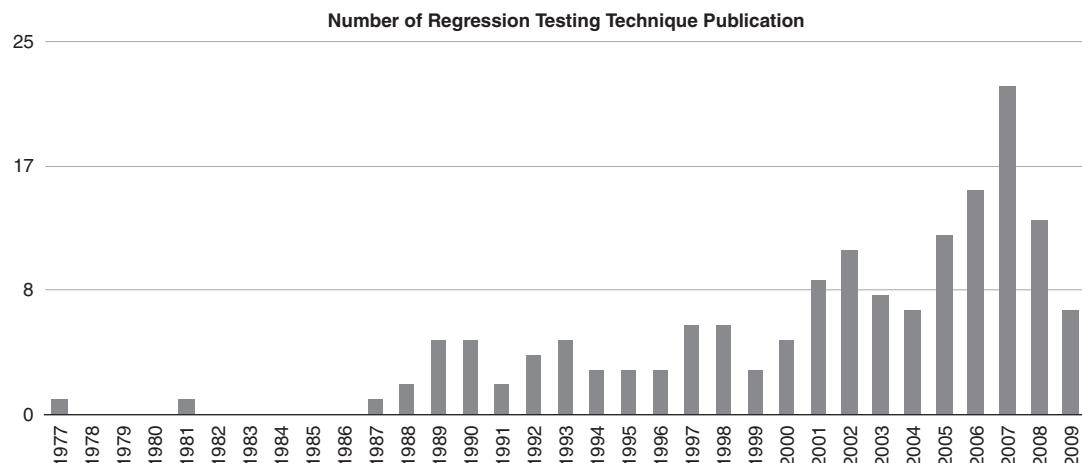


Figure 2. Number of surveyed papers in each year since 1977. The field is still growing.

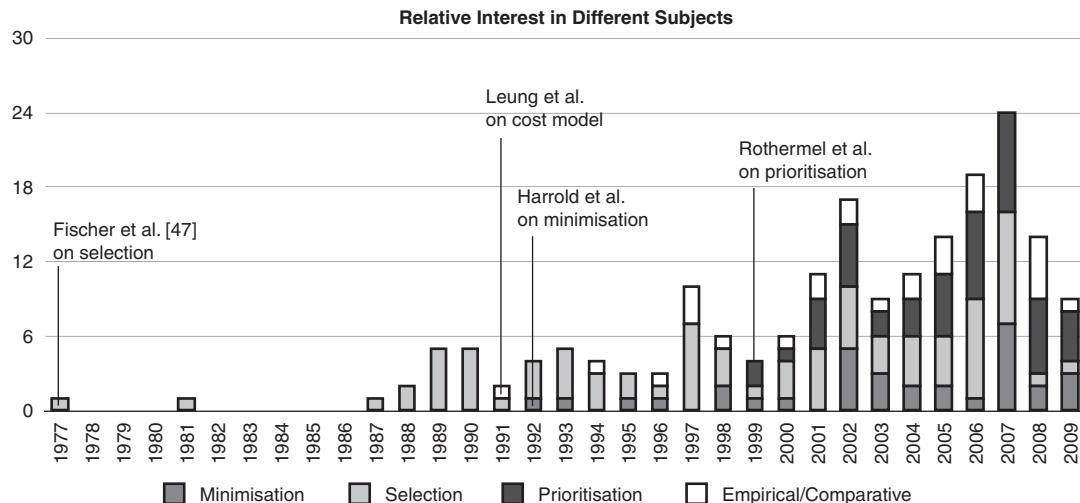


Figure 3. Relative research interest in each subject. Papers that consider more than one subject were counted multiple times.

categories contain papers on minimization, selection and prioritization, respectively. The fourth category contains papers on empirical evaluation and comparative studies, including previous surveys. Papers that consider more than one subject are represented in each category for which they are relevant; for example, a survey on RTS [1] is counted in both the selection category and the comparative studies category. Therefore, while the graph closely resembles Figure 2, it is not a representation of the number of publications. Rather, the figure should be read as a guide to the trends in study topics over time.

Considering that many different RTS approaches were introduced in the late 80s and 90s, recent research on RTS techniques has been mostly concerned with the application and evaluation of the graph-walk approach [83, 84, 87, 90–95]. As the figure reveals, the interest in test case prioritization has been steadily growing since the late 90s. In Figures 2 and 3, the data for 2009 is, of course, partial.

Whereas most of the early papers on RTS were theoretical (Table AII), empirical evaluation of regression testing techniques has recently received a burgeoning interest. Not only are there more publications on pure empirical/comparative studies (as can be observed in Figure 3), but recent studies of regression testing techniques tend to evaluate the suggested techniques empirically, as can be observed in Tables AI–AIII.

However, the scale of empirical studies seems to remain limited. Figure 4 shows the maximum size of SUTs (measured in LOC) and test suites (measured as the number of test cases) studied empirically in the literature. For both data, only the empirical studies that explicitly note the size of subject SUTs and test suites have been included. When only the average size of test suites is given, the maximum average size of studied test suites has been used. For the maximum size of SUTs, only the empirical studies that use source code as test subjects have been included; for example, studies of regression testing of UML models are not included. For about 60% of empirical studies, the largest SUT studied is smaller than 10 000 LoC. For about 70% of empirical studies, the largest test suite studied contains fewer than 1000 test cases.

Figure 5 shows the origins of subject SUTs studied in the literature. For detailed information about the classification, refer to the Appendix. Programs available from the Software Infrastructure Repository (SIR) [179] account for over 50% of subjects of empirical studies of regression testing techniques. The predominant programming language is C, followed by Java. Considering that the first paper appeared in 2002, model-based techniques have shown significant growth. Although there are several papers on RTS for web applications [91–95], empirical evaluation of these techniques remains limited.

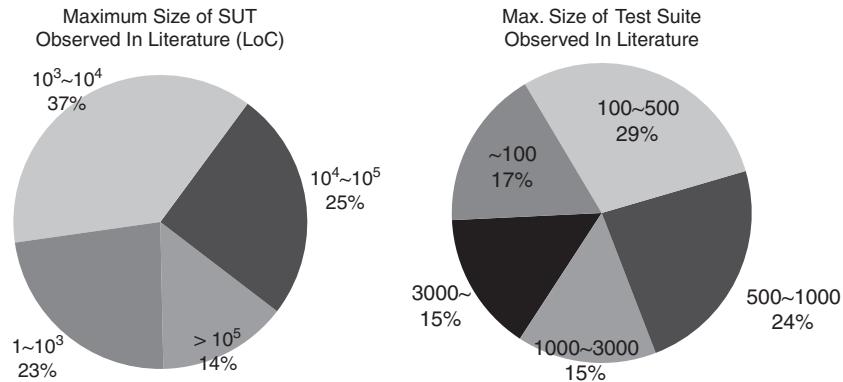


Figure 4. Maximum size of SUT and test suites studied in the literature.

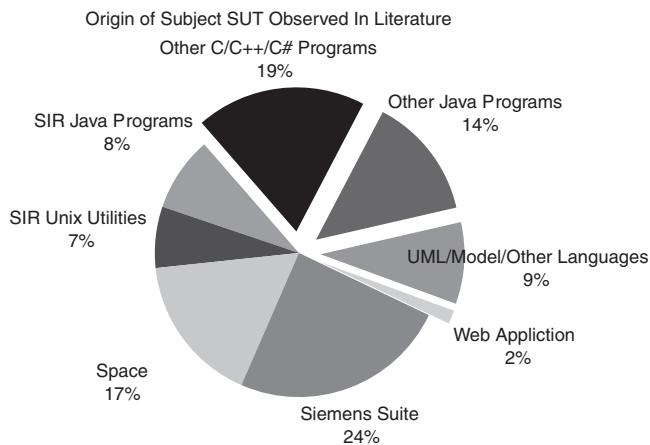


Figure 5. Origins of subject SUTs observed in the literature.

## 7.2. State-of-the-art, trends and issues

**7.2.1. State-of-the-art** Among the class of RTS techniques, the graph-walk approach seems to be the predominant technique in the literature. Although it was originally conceived for CDGs and PDGs [46, 56], the most widely used form of graph-walk approach works on CFGs [57, 79]. Its popularity can be observed from the fact that the approach has been applied to many forms of graph representation of SUT that are not CFGs [82–84, 87]. Indeed, the strength of the graph-walk approach lies not only in the fact that it is intuitive but also in the generic applicability of the technique to any graph representation of systems.

By studying the literature, it becomes clear that two ideas played essential roles in the development of RTS techniques: Leung and White's early idea of regression testing and test case classification [10], and Rothermel and Harrold's definition of a *safe* RTS [45]. Collectively, these two ideas provided a profound theoretical framework that can be used to evaluate RTS techniques.

The existence of such a theoretical framework is what differentiate RTS techniques from test suite minimization and test case prioritization. For RTS techniques, it is possible to define what a *safe* technique should do because the RTS problem is specifically focused on the modifications between two versions of SUT. Minimization and prioritization techniques, on the other hand, are forced to rely on surrogate metrics for real fault-detection capability. Therefore, it may not be clear what 'safe' minimization or prioritization techniques would mean. A minimization technique would not be safe unless the surrogate metric perfectly captures the fault-detection capability of the test suite; however, the empirical studies so far have shown that there is no such single metric.

For prioritization, the concept of ‘safe’ and ‘non-safe’ does not apply directly since the aim of the technique is to permute the test suite and not to select a subset out of it.

Naturally, the history of minimization and prioritization literature is an on-going exploration of different heuristics and surrogate metrics. It is interesting to note that the greedy algorithm, a good approximation for the set cover problem and, therefore, test suite minimization problem, is also an efficient heuristic for test case prioritization precisely because of its *greedy* nature. In other words, *as much as possible, as soon as possible*. As a result of this, the greedy approach and its variations have a strong presence in the literature on both test suite minimization [11, 14, 18] and test case prioritization [115, 116, 121]. Recently, there are other approaches to test suite minimization and test case prioritization that aim to overcome the uncertainty of surrogates, e.g. the use of multiple minimization criteria [21, 22, 24] and the use of expert knowledge for test case prioritization [137, 138].

**7.2.2. Trends Emphasis on Models:** Whereas most of the early regression testing techniques concerned code-based, white-box regression testing, the model-based regression testing approach has been of more recent growing interest [32–34, 68, 69, 88, 107–111]. UML models and EFSMs are often used.

**New Domains:** While the majority of existing literature on regression testing techniques concerns stand-alone programs written in C and Java, there are a growing number of other domains that are being considered for regression testing. For example, spreadsheets [76], GUIs [25, 180] and web applications [91–95, 153] have been considered.

**Multi-criteria Regression Testing:** In minimization and prioritization, it is known that there is no single surrogate metric that correlates to fault prediction capability for all programs [8, 38–40, 115, 121]. One potential way of overcoming this limitation is to consider multiple surrogates simultaneously [21, 22] using classical multi-objective optimization techniques, such as the weighted-sum approach [21] or the prioritized approach [22]. Expert domain knowledge has also been used in addition to software metrics for test case prioritization [137, 138]. Finally, Multi-Objective Evolutionary Algorithms (MOEAs) have been used to deal with multiple objectives [24].

**Cost-awareness:** The cost of regression testing is a cross-cutting concern for all three classes of regression testing techniques. Studies of cost-aware regression testing can be classified into two categories. First, there is work that aims to incorporate the cost of testing directly into regression testing techniques at technical level; for example, cost-aware test case prioritization [23, 118] or minimization techniques that provide the tester with a series of alternative subsets of the original test suites that can be executed in different amounts of time [24]. Second, there are empirical evaluations of regression testing techniques that consider whether the application of these techniques is indeed cost-effective in the wider context of the overall software life-cycle [120, 158, 165, 169, 177].

**7.2.3. Issues Limited Subjects:** In Figure 5, subject programs from SIR [179] account for almost 60% of the subjects of empirical studies observed in the regression testing technique literature. While this is certainly evidence that SIR has been of tremendous value to the research community, it also means that many regression testing techniques are being evaluated against a limited set of programs and test suites. By no means is this a criticism of SIR itself or work that is based on subjects from SIR; rather, the dependency on SIR shows how time consuming and difficult it is to collect multiple versions of program source code, their test suites and associated fault information, let alone to make it publicly available. The SIR commonality also supports and facilitates comparisons. However, the community faces a possible risk of ‘over-fitting’ the research of these techniques to those programs that are easily available. Open source software projects are often suggested as an alternative source of data, but from the results of this survey analysis it seems that their use for research on regression testing techniques is limited.

Two potential approaches to the issue of limited subjects can be envisioned. The first approach is to design a method that will allow a realistic simulation of real software faults. There is early work in this direction. Andrews *et al.* considered whether mutation faults are similar to real faults [161];

Do and Rothermel studied the same question especially in the context of regression testing [163]. Jia and Harman used a search-based approach to obtain higher-order mutation faults that are more ‘subtle’ and, therefore, potentially harder to detect than first-order mutation faults [181].

The second approach is to engage more actively with industry and open source communities. This is not an easy task, because the information about testing and software faults tends to be sensitive, particularly to commercial organizations. The research community faces the challenge of convincing the wider practitioner community of the cost-effectiveness and usefulness of these techniques. This is closely related to the issue of technology transfer, to which the paper now turns.

*Technology Transfer:* A detailed observation of the literature suggests that the community may have reached a stage of maturity that, in order to progress to the next level of achievement, technology transfer to industry will play an important role. While the research community may not have found the ‘silver bullet’, most empirical evaluation of the proposed techniques suggests that application of minimization, selection and prioritization techniques does make a difference from (1) uncontrolled regression testing, i.e. retest-all approach and un-prioritized regression testing, and (2) random approaches.

However, empirical evaluation and application of regression testing techniques at industrial level seems to remain limited [182]. Out of the 159 papers listed in Tables AI–AIV, only 31 papers list a member of industry as an author or a co-author. More importantly, only 12 papers consider industrial software artefacts as a subject of the associated empirical studies [68, 69, 88, 99–102, 104–106, 126, 147]. This suggests that a large-scale industrial uptake of these techniques has yet to occur.

## 8. FUTURE DIRECTIONS

This section discusses some of the possible future directions in the field of regression testing techniques. While it is not possible to predict the future direction that a field of study will follow, it was possible to identify some trends in literature, which may suggest and guide the direction of future research.

### 8.1. Orchestrating regression testing techniques with test data generation

Automatic test data generation has made advances in both functional and non-functional testing [183]. Superficially test data generation is the counterpart to regression testing techniques; it creates test cases while regression testing seeks to mange them. However, *because* these two activities are typically located at the opposite ends of the testing process, they may become close when the testing process repeats.

Orchestrating regression testing with test data generation has a long heritage. From the classification of test cases by Leung and White [10], it follows that regression testing involves two different needs that are closely related to test data generation: repairing obsolete test cases for corrective regression testing and generating additional test data for progressive regression testing. The second problem, in particular, has been referred to as the test suite augmentation problem [184, 185]. There was early work on both problems. Memon and Soffa considered automatic repair of GUI test cases that were made obsolete by the changes in the GUI [180], which was later extended by Memon [186]. Similarly, Alshahwan and Harman focused on the use of user session data for regression testing of web applications, and how the test cases can be automatically repaired for the next version [187]. Apiwattanapong *et al.* identified the testing requirements that are needed to test the new and modified parts of an SUT so that additional test data can be generated automatically [184, 185].

However, it is believed that there are additional areas that may be synergetic. For example, test data generation might possibly refer to the test cases selected and unselected during the last iteration in order to identify the part of the SUT to focus on. Similarly, regression testing techniques can use the additional information provided by test data generation techniques in order to make regression testing more efficient and effective. For example, there are test data generation

techniques that target a specific concern in the SUT, such as detection of the presence of a memory leak. The additional information about the *intention* behind each test case could be used to enrich the minimization, selection and prioritization process.

### 8.2. Multi-objective regression testing

Regression testing is a complex and costly process that may involve multiple objectives and constraints. For example, the cost of executing a test case is usually measured as the time taken to execute the test case. However, there may be a series of different costs involved in executing a test case, such as setting up the environment or preparing a test input, each of which may be subject to a different constraint. Existing techniques also assume that test cases can be executed in any given order without any change to the cost of execution, which seems unrealistic. Test cases may have dependency relations between them. It may also be possible to lower the cost of execution by grouping test cases that share the same test environment, thereby saving set-up time.

Considering the complexity of real-world regression testing, existing representations of problems in regression testing may be over-simplistic. Indeed, most of the published empirical studies rely on relatively small-scale academic examples. Even when real-world programs are studied, they tend to be individual programs, not a software system as a whole. Larger software systems do not simply entail larger problem size; they may denote a different level of complexity.

It is believed that, in order to cope with the complexity of regression testing, regression testing techniques may need to become multi-objective. There is existing, preliminary work that seeks to represent the cost of test case execution as an additional constraint using evolutionary algorithms [23, 24]. Multi-objective optimization heuristics may provide the much-needed flexibility that is required for the representation of problems with high complexity.

Another benefit of moving to a multi-objective paradigm is the fact that it provides additional insight into the regression testing problem by allowing the testers to observe the inherent trade-offs between multiple constraints. This is not possible with the so-called *classical* approaches to multi-objective problems that either consider one objective at a time [22] or conflate multiple objectives into a single objective using weighting [21, 23]; these approaches may be based on *multiple criteria*, but they are not truly multi-objective in the sense that they all produce a single solution. The result of a multi-objective optimization is often a set of solutions that do not dominate each other, thereby forming the trade-offs between constraints. The insight into the trade-offs may provide additional information that is hard to obtain manually.

### 8.3. Problem of test oracle and its cost

Test oracles present a set of challenging problems for software testing. It is difficult to generate them automatically, they often require human efforts to verify and the cost of this effort is hard to estimate and measure. The oracle cost has been considered as a part of cost models [178], but has not been considered as a part of the process of minimization, selection and prioritization itself. Since regression testing techniques seek to efficiently re-use existing test cases, information about the cost of verifying the output observed with the existing test suite may be collected across versions. This can be incorporated into the existing regression testing techniques.

While a test oracle and its cost may be seen as yet another additional objective that can be considered using a multi-objective approach, it can be argued that the test oracle will present many interesting and exciting research questions in the context of regression testing and, thus, deserves a special treatment in its own right. This is because, compared to other testing cost such as the physical execution time of test cases, the test oracle cost is closely related to the *quality* of testing. Moreover, unlike some costs that can be reduced by using more advanced hardware, the cost of oracle verification derives from human effort and is, therefore, harder to reduce. These characteristics make the issues related to test oracles challenging but interesting research subjects.

#### 8.4. Consideration of other domains

The majority of regression testing techniques studied in this survey concern white-box structural regression testing of code-level software artefacts. However, other domains are emerging as new and exciting research subjects.

Recently, Service-Oriented Architectures (SOAs) have been of keen interest both for academic researchers and industrialists. In the SOA paradigm, software system is built, or *composed*, by orchestrating a set of web services, each of which takes charge of a specific task. Several approaches have been introduced to address the issue of regression testing of web services, most of which seek to apply the same technique developed for traditional applications to web services [91–95]. However, the inherently distributed nature of an SOA system presents several challenges that are alien to traditional regression testing techniques.

Web services often reside in remote locations and are developed by a third-party, making it hard to apply the traditional white-box regression testing techniques that require analysis of source code. Modifications can happen across multiple services, which can make fault localization difficult. High interactivity in web applications may result in complex test cases that may involve human interaction. Finally, distributed systems often contain concurrency issues. Traditional regression testing techniques assume that the program produces deterministic output. This may not be adequate for testing applications with concurrency. Answers to these specific issues in regression testing of web applications are still in the early stage of development.

Model-based regression testing techniques have also received growing interests [68, 69, 107–111]. It is believed that the model-based regression testing techniques will be of crucial importance in the future for the following reasons:

- *Higher level regression testing*: These techniques can act as a medium between requirement/specification and testing activities, bringing regression testing from the structural level to functional level.
- *Scalability*: in dealing with software systems of industrial scale, model-based techniques will scale up better than code-based techniques.

However, there are a few open research questions. First, there is the well-known issue of traceability. Unless the traceability from requirements and specifications to code-level artefacts and test cases is provided, the role of model-based regression testing techniques will be severely limited. Second, there is the issue of test adequacy: if a test adequacy  $A$  is appropriate for a model  $M$ , which test adequacy should be used to test the program  $P$  that has been automatically generated from  $M$ ? Does  $A$  still apply to  $P$ ? If so, does it follow that  $M$  being adequate for  $A$  means  $P$  will be adequate for  $A$  as well?

There are also other interesting domains to consider. Testing of GUIs has received growing interest, not only in the context of regression testing [25, 135, 180], but also in the context of testing in general [188, 189]. Regression testing GUIs present a different set of challenges to code-based structural regression testing since GUIs are usually generated in a visual programming environment; they are often subject to frequent changes and, not being well-typed, do not readily facilitate static analysis.

#### 8.5. Non-functional testing

A majority of existing regression testing techniques rely upon structural information about the SUT, such as data-flow analysis, CFG analysis, program slices and structural coverage. The impact that non-functional property testing will have on regression testing techniques has not been fully studied. Existing techniques were able to map the problems in the regression testing to well-formed abstract problems using the properties of structural information. For example, test suite minimization could be mapped to the minimal hitting set problem or the set coverage problem, precisely because the techniques were based on the concept of ‘coverage’. Similarly, graph-walking approaches to test case selection were made possible because the changes between different versions were defined by structural differences in CFGs.

Imagine regression testing techniques for non-functional properties. What would be the minimized test suite that can test the power consumption of an embedded system? How would test cases be prioritized to achieve an efficient and effective stress testing of a web application? These questions remain largely unanswered and may require approaches that are significantly different from existing paradigms.

### 8.6. Tool support

Closely related to the issue of technology transfer is the issue of tool support. Without readily available tools that implement regression testing techniques, practical adoption will remain limited. One potential difficulty of providing tool support is the fact that, unlike unit testing for which there exists a series of frameworks based on the xUnit architecture, there is not a common framework for the regression testing process in general. The closest to a common ground for regression testing would be an Integrated Development Environment (IDE), such as Eclipse, with which the xUnit architecture is already integrated successfully. A good starting point for regression testing techniques may be the management framework of unit test cases, built upon xUnit architecture and IDEs.

## 9. CONCLUSION

This paper provides both a survey and a detailed analysis of trends in regression test case selection, minimization and prioritization. The paper shows how the work on these three topics is closely related and provides a survey of the landscape of work on the development of these ideas, their applications, empirical evaluation and open problems for future work.

The analysis of trends reported in the paper reveals some interesting properties. There is evidence to suggest that the topic of test case prioritization is of increasing importance, judging by the shift in emphasis towards it that is evident in the literature. It is also clear that the research community is moving towards assessment of the complex trade-offs and balances between different concerns, with an increase in work that considers the best way in which to incorporate multiple concerns (cost and value for instance) and to fully evaluate regression testing improvement techniques.

This focus on empirical methodology is one tentative sign that the field is beginning to mature. The trend analysis also indicates a rising profile of publication, providing evidence to support the claim that the field continues to attract growing attention from the wider research community, which is a positive finding for those working on regression test case selection and minimisation and, in particular those working on prioritization problems.

Our survey also provides evidence to indicate that there is a preponderance of empirical work that draws upon a comparatively small set of subjects (notably those available through the SIR repository). This is a testament to the importance of this source of case study material. It is valuable because it allows for cross comparison of results and replication, which is essential for the development of any science. However, it may potentially suggest a risk of over-fitting.

## APPENDIX A

The following tables contain detailed information about publications on each class of technique: test suite minimization in Table AII, RTS in Table AIII, test case prioritization in Table AIV and empirical/comparative studies in Table AIV. Note that publications that consider more than one class of techniques appear in multiple tables for ease of reference. For example, a survey of Regression Test Selection [1] appears in both Tables AII and AIV.

Information on the maximum size of SUT (measured in LOC) and the maximum size of test suites were collected from papers only when they were explicitly noted. Some papers that considered multiple test suites for a single program contained only the average, in which case the size of the test suite with largest average size was recorded. When the studied SUT cannot be measured in

Table A1. Summary of publications on test suite minimization.

Reference	Year	Max. SUT size (LoC)	Max. Test suite size	Siemens suite space	SIR Unix utilities	SIR Java programs	Other C/C# programs	Java languages	Other other languages	Models and Web applications
Horgan <i>et al.</i> [13]	1992	1000	26							
Harrold <i>et al.</i> [12]	1993	N/A	19							
Offutt <i>et al.</i> [14]	1995	48	26.8							
Chen <i>et al.</i> [11]	1996	N/A*	—							
Rothermel <i>et al.</i> [38]	1998	516	260	•						
Wong <i>et al.</i> [39]	1998	842	33	•						
Wong <i>et al.</i> [40]	1999	6218	200							
Schroeder and Korel [31]	2000	Theory	—							
Korel <i>et al.</i> [33]	2002	Theory	—							
Malishevsky <i>et al.</i> [119]	2002	65 632	1168							
Rothermel <i>et al.</i> [120]	2002	68 782	1985							
Rothermel <i>et al.</i> [8]	2002	516	260	•						
Vaysburg <i>et al.</i> [32]	2002	Theory	—							
Anido <i>et al.</i> [35]	2003	Theory	—							
Harder <i>et al.</i> [27]	2003	6218	169	•						
Marré and Bertolini [17]	2003	516	5542	•						
Black <i>et al.</i> [21]	2004	512	5542	•						
Rothermel <i>et al.</i> [165]	2004	68 782	1985	•						
Jeffrey and Gupta [19]	2005	516	135	•						
McMaster and Memon [43]	2005	6218	4712	•						
Tallam and Gupta [18]	2006	6218	539	•						
Chen <i>et al.</i> [34]	2007	Theory	—							
Hou <i>et al.</i> [152]	2007	5500	183	•						
Jeffrey and Gupta [20]	2007	6218	1560	•						
Leitner <i>et al.</i> [29]	2007	N/A†	—							
McMaster and Memon [42]	2007	11 803	1500	•						
Smith <i>et al.</i> [26]	2007	1455	N/A	•						
Yoo <i>et al.</i> [24]	2007	6218	169	•						

Table AI. *Continued.*

Reference	Year	Max. SUT size (LoC)	Max. Test suite size	Siemens suite	space	SIR Unix utilities	SIR Java Programs	Other C/ C++/C#	Other Java programs	Models and other languages	Web applications
McMaster and Memon [25]	2008	11 803	1500							•	
Yu <i>et al.</i> [44]	2008	6218	13 585	•	•	•					
Zhong <i>et al.</i> [6]	2008	26 824	N/A	•	•	•					
Hsu <i>et al.</i> [22]	2009	1 892 226	5542	•	•	•					
Kaminski <i>et al.</i> [36]	2009	N/A <sup>†</sup>	—								
Smith and Kapfhammer [176]	2009	6822	110							•	

\*Chen *et al.* [11]<sup>1</sup> evaluated their heuristics using simulation rather than real data.

<sup>1</sup>Leitner *et al.* [29]<sup>2</sup> minimized the length of a unit test case, not a test suite.

<sup>‡</sup>Kaminski *et al.* [36]<sup>3</sup> applied logical reduction to a set of 19 boolean predicates taken from the avionic software.

Table AII. Summary of publications on Regression Test Selection (RTS).

Reference	Year	Max. SUT size (LoC)	Max. Test suite size	Siemens suite	SIR Unix utilities	SIR Java Programs	Other C/C++/C#	Other Java programs	Models and other languages	Web applications
Fischer [47]	1977	Theory	—							
Fischer <i>et al.</i> [48]	1981	Theory	—							
Yau and Kishimoto [53]	1987	Theory	—							
Benedusi <i>et al.</i> [61]	1988	Theory	—							
Harrold and Soffa [50]	1988	Theory	—							
Harrold and Soffa [74]	1989	Theory	—							
Harrold and Soffa [51]	1989	Theory	—							
Hartmann and Robson [71]	1989	Theory	—							
Leung and White [10]	1989	Theory	—							
Taha <i>et al.</i> [52]	1989	Theory	—							
Hartmann and Robson [73]	1990	Theory	—							
Hartmann and Robson [72]	1990	Theory	—							
Lee and He [70]	1990	Theory	—							
Leung and White [63]	1990	Theory	—							
Leung and White [64]	1990	550	235							
Horgan and London [15]	1991	1000	26							
Gupta <i>et al.</i> [49]	1992	Theory	—							
Laski and Zemmer [67]	1992	Theory	—							
White and Leung [65]	1992	Theory	—							
Agrawal <i>et al.</i> [54]	1993	Theory	—							
Bates Horwitz [60]	1993	Theory	—							
Rothermel and Harrold [46]	1993	Theory	—							
White <i>et al.</i> [66]	1993	Theory	—							
Chen <i>et al.</i> [62]	1994	11 000	39							
Rothermel and Harrold [56]	1994	Theory	—							
Rothermel and Harrold [45]	1994	Theory	—							
Binkley [96]	1995	Theory	—							
Kung <i>et al.</i> [98]	1995	N/A*	—							
Rothermel and Harrold [1]	1996	Survey	—							
Rothermel [55]	1996	516	5542	•						

Table AII. *Continued.*

Reference	Year	Max. SUT size (LoC)	Max. Test suite size	Siemens suite	SIR space	Unix utilities	SIR Java Programs	Other C/C#	Other Java programs	Models and other languages	Web applications
Binkley [97]	1997	623	— <sup>†</sup>								
Rosenblum and Rothermel [5]	1997	49 316	1033	•							
Rosenblum and Weyuker [169]	1997	N/A <sup>‡</sup>	N/A								
Rothermel and Harrold [57]	1997	516	5542	•							
Rothermel and Harrold [78]	1997	512	5542	•							
Wong <i>et al.</i> [75]	1997	6218	1000								
Vokolos and Frankl [58]	1997	N/A <sup>§</sup>	N/A								
Ball [80]	1998	Theory	—								
Graves <i>et al.</i> [7]	1998	516	398	•							
Rothermel and Harrold [79]	1998	516	5542	•							
Vokolos and Frankl [59]	1998	6218	100	•							
Harrold [114]	1999	Theory	—								
Kim <i>et al.</i> [166]	2000	6218	4361	•							
Le Traon <i>et al.</i> [110]	2000	N/A <sup>¶</sup>	N/A								
Rothermel <i>et al.</i> [81]	2000	24 849	317								
Beydeda and Gruhn [85]	2001	Theory	—								
Bible <i>et al.</i> [4]	2001	49 316	1033	•							
Harrold <i>et al.</i> [82]	2001	N/A	189								
Harrold <i>et al.</i> [173]	2001	516	19	•							
Jones and Harrold [124]	2001	6218	4712	•							
Orso <i>et al.</i> [89]	2001	6035	138								
Briand <i>et al.</i> [68]	2002	N/A <sup>  </sup>	596								
Chen <i>et al.</i> [88]	2002	N/A	306								
Fisher II <i>et al.</i> [76]	2002	N/A <sup>**</sup>	493								
Malishevsky <i>et al.</i> [119]	2002	65 632	1168								
Rothermel <i>et al.</i> [120]	2002	68 782	1985								
Elbaum <i>et al.</i> [164]	2003	65 632	1168								
Wu and Offutt [111]	2003	Theory	—								
White <i>et al.</i> [101]	2003	N/A <sup>††</sup>	N/A								
Deng <i>et al.</i> [107]	2004	Theory	—								
Rothermel <i>et al.</i> [165]	2004	68 782	1985								

Table AII. *Continued.*

Reference	Year	Max. SUT size (LoC)	Max. Test suite size	Siemens suite space	SIR Unix utilities	SIR Java Programs	Other C/C++/C#	Other Java programs	Models and other languages	Web applications
Orso <i>et al.</i> [86]	2004	532 000	707							
White and Robinson [99]	2004	N/A††	N/A							
Kim <i>et al.</i> [167]	2005	6218	4361	•						
Martins and Vieira [87]	2005	902	N/A							
Muccini <i>et al.</i> [112]	2005	N/A‡‡								
Skoglund and Runeson [106]	2005	1 200 000	N/A							
Do and Rothermel [177]	2006	80 400	1533							
Lin <i>et al.</i> [91]	2006	N/A##	N/A							
Pilskalns <i>et al.</i> [108]	2006	N/A	52							
Tarhini <i>et al.</i> [94]	2006	Theory	—							
Zhao <i>et al.</i> [84]	2006	Theory	—							
Zheng <i>et al.</i> [102]	2006	757 000	592							
Zheng <i>et al.</i> [103]	2006	757 000	592							
Muccini <i>et al.</i> [113]	2006	N/A‡‡	N/A							
Faroq <i>et al.</i> [109]	2007	Theory	—							
Orso <i>et al.</i> [90]	2007	6035	567	•						
Ruth and Tu [92]	2007	Theory	—							
Ruth <i>et al.</i> [95]	2007	Theory	—							
Ruth and Tu [93]	2007	Theory	—							
Sherriff <i>et al.</i> [147]	2007	Theory	—							
Xu and Rountev [83]	2007	3423	63	•						

Table AII. *Continued.*

Reference	Year	Max. SUT size (LoC)	Max. Test suite size	Siemens suite space	SIR Unix utilities	SIR Java Programs	Other C/C++/C#	Other Java programs	Models and other languages	Web applications
Zheng <i>et al.</i> [104]	2007	757 000	592							
Zheng <i>et al.</i> [105]	2007	757 000	31							
Fahad and Nadeem [3]	2008	Survey	—							
White <i>et al.</i> [100]	2008	Over 1MLOC	N/A							
Briand <i>et al.</i> [69]	2009	N/A <sup>  </sup>	323 614							
				•						

\* Kung *et al.* applied their technique to InterView C++ library, which contains 147 files and over 140 classes [98].

† Binkley [97] applied his technique to five C programs (the largest of which had 623 statements), but only identified the statements that require retesting without considering test suites.

‡ Roseblum and Weyuker [169] evaluated their cost-effectiveness predictor using 31 versions of the KornShell and a single version of the SFIO (Unix library), but exact versions, sizes of SUT and sizes of test suites were not specified.

§ Vokolos and Frankl evaluated their textual difference selection technique using a small C function in addmon family of tools, power [58].

¶ Le Traon *et al.* [110] presented a case study of a model of packet-switched data transport service, the size of which was not specified.

|| Briand *et al.* studied UML models rather than real systems, the biggest of which contained either 9 classes with 70 methods [68] or 32 classes with 64 methods [69]. Muccini *et al.* studied an RTS technique at software architecture level and presented case studies for the architecture model of an elevator system and a cargo router system [112, 113].

\*\* Fisher II *et al.* [76] evaluated their retesting strategy for spreadsheets with a spreadsheet containing 48 cells, 248 expressions and 100 predicates.

†† White *et al.* applied the firewall approach to GUI objects [101]. White and Robinson applied the firewall approach to a real time system developed by ABB [99].

‡‡ Lin *et al.* [91] applied their technique to a Java Interclass Graph (JIG) with over 100 nodes.

Table AIII. Summary of publications on test case prioritization

Reference	Year	Max. SUT size (LoC)	Max. Test suite size	Siemens suite	Space	SIR utilities	SIR Unix	SIR Java	Other C/C#	Other Java programs	Java	Other languages	Models and Web applications
Harrold [114]	1999	Theory	—										
Rothermel <i>et al.</i> [115]	1999	516	19	•									
Elbaum <i>et al.</i> [116]	2000	6218	169	•	•								
Elbaum <i>et al.</i> [117]	2001	6218	169	•	•								
Elbaum <i>et al.</i> [118]	2001	6218	166	•	•								
Jones <i>et al.</i> [124]	2001	6218	4712	•	•								
Rothermel <i>et al.</i> [121]	2001	6218	169	•	•								
Elbaum <i>et al.</i> [123]	2002	6218	169	•	•								
Kim <i>et al.</i> [144]	2002	6218	226	•	•								
Malishhevsky <i>et al.</i> [119]	2002	65632	1168	•	•								
Rothermel <i>et al.</i> [120]	2002	68782	1985	•	•								
Srivastava <i>et al.</i> [126]	2002	18 000 000*	3128	•	•								
Elbaum <i>et al.</i> [164]	2003	65632	1168	•	•								
Leon <i>et al.</i> [136]	2003	N/A <sup>†</sup>	3333	•	•								
Do <i>et al.</i> [127]	2004	80 400	877	•	•								
Elbaum <i>et al.</i> [168]	2004	68 000	1985.32	•	•								
Rothermel <i>et al.</i> [165]	2004	68782	1985	•	•								
Bryce <i>et al.</i> [131]	2005	N/A <sup>‡</sup>	—	•	•								
Do <i>et al.</i> [162]	2005	80 400	877	•	•								
Korel <i>et al.</i> [149]	2005	800	980	•	•								
Rummel <i>et al.</i> [156]	2005	N/A <sup>§</sup>	21	•	•								
Srikanth <i>et al.</i> [139]	2005	2500	50	•	•								
Bryce <i>et al.</i> [132]	2006	N/A <sup>‡</sup>	—	•	•								
Do <i>et al.</i> [177]	2006	80 400	1533	•	•								
Do <i>et al.</i> [175]	2006	80 400	877	•	•								
Do <i>et al.</i> [163]	2006	80 400	877	•	•								
Jeffrey <i>et al.</i> [157]	2006	516	N/A	•	•								
Tonella <i>et al.</i> [137]	2006	6218	169	•	•								

Table AIII. *Continued.*

Reference	Year	Max. SUT size (LoC)	Max. Test suite size	Siemens suite	SIR Unix utilities	SIR Java Programs	Other C/C#	Other Java programs	Models and other languages	Web applications
Walcott <i>et al.</i> [23]	2006	1808	53							•
Bryce <i>et al.</i> [135]	2007	N/A <sup>†</sup>	—							
Fraser <i>et al.</i> [154]	2007	N/A <sup>§</sup>	246							
Hou <i>et al.</i> [152]	2007	5500	183							•
Korel <i>et al.</i> [150]	2007	1416	1000							•
Li <i>et al.</i> [128]	2007	11148	4350	•						•
Mirarab <i>et al.</i> [145]	2007	124 000	105							
Qu <i>et al.</i> [133]	2007	17 155	796							
Smith <i>et al.</i> [26]	2007	1455	N/A							
Do <i>et al.</i> [158]	2008	80 400	912							
Hou <i>et al.</i> [159]	2008	N/A <sup>¶</sup>	1000							
Korel <i>et al.</i> [151]	2008	1416	1439	•						
Mirarab <i>et al.</i> [146]	2008	80 400	912							
Qu <i>et al.</i> [134]	2008	107 992	975							
Sampath <i>et al.</i> [153]	2008	9401	890							
Krishnamoorthi <i>et al.</i> [148]	2009	6000	N/A							
Smith <i>et al.</i> [176]	2009	6822	110							
Yoo <i>et al.</i> [138]	2009	122 169	1061	•						
Zhang <i>et al.</i> [160]	2009	5361	209							•

\*Srivastava and Thiagarajan [126] considered the biggest software system so far, an office productivity application with over 18 million LoC. However, the technique took the executable binary as input, not the source code. The compiled application was 8.8 Mb in size, with a 22 Mb symbol table.

<sup>†</sup>Leon *et al.* [136] considered three compilers: javac, jikes and gcc. The sizes of the source code were not specified.

<sup>‡</sup>Bryce *et al.* [131, 132, 135] studied interaction coverage prioritization, for which the LoC metric is not appropriate.

<sup>§</sup>These papers considered models rather than real software systems. Rummel *et al.* [156] applied their technique to a model with 3 classes and 21 methods. Fraser *et al.* [154] did not specify the size of the studied models and test suites.

<sup>¶</sup>Hou *et al.* [159] evaluated their technique using a web application composed of 12 web services.

Table AIV. Summary of publications on empirical evaluation and comparative studies.

Reference	Year	Max. SUT size (LoC)	Max. Test suite size	Siemens suite	SIR Unix utilities	SIR Java Programs	Other C/C# programs	Other Java programs	Models and other languages	Web applications
Leung <i>et al.</i> [172]	1991	Theory	—							
Rothermel <i>et al.</i> [45]	1994	Theory	—							
Rothermel <i>et al.</i> [1]	1996	Survey	—							
Rosenblum <i>et al.</i> [5]	1997	49 316	1033	•						
Rosenblum <i>et al.</i> [169]	1997	N/A*	N/A							
Rothermel <i>et al.</i> [78]	1997	512	5542	•						
Graves <i>et al.</i> [7]	1998	516	398	•	•					
Kim <i>et al.</i> [166]	2000	6218	4361							
Bible <i>et al.</i> [4]	2001	49 316	1033	•						
Harrold <i>et al.</i> [173]	2001	516	19	•						
Malishevsky <i>et al.</i> [119]	2002	65 632	1168	•	•	•				
Rothermel <i>et al.</i> [120]	2002	68 782	1985							
Elbaum <i>et al.</i> [164]	2003	65 632	1168							
Elbaum <i>et al.</i> [168]	2004	68 000	1985.32	•	•	•	•	•		
Rothermel <i>et al.</i> [165]	2004	68 782	1985	•	•	•	•	•		
Do <i>et al.</i> [162]	2005	80 400	877							
Do <i>et al.</i> [179]	2005	Theory	—							
Kim <i>et al.</i> [167]	2005	6218	4361	•						
Do <i>et al.</i> [177]	2006	80 400	1533							
Do <i>et al.</i> [175]	2006	80 400	877							
Do <i>et al.</i> [163]	2006	80 400	877							
Do <i>et al.</i> [158]	2008	80 400	912							
Do <i>et al.</i> [178]	2008	80 400	912							
Engström <i>et al.</i> [2]	2008	Systematic Review	—							
Fahad <i>et al.</i> [3]	2008	Survey	—							
Zhong <i>et al.</i> [6]	2008	26 824	N/A	•						
Smith <i>et al.</i> [176]	2009	6822	110	•						

\*Rosenblum *et al.* [169] evaluated their cost-effectiveness predictor using 31 versions of the KornShell and a single version of the SFIO (Unix library), but exact versions, sizes of SUT and sizes of test suites were not specified.

LOC, the detailed information was provided in footnotes. Tables also contain information about the origins of the studied SUT, which are classified as follows:

- *Siemens suite* [41]: All or part of the following set of C programs—printtokens, printtokens2, schedule, schedule2, replace, tcas, totinfo, available from SIR [179].
- *space*: An interpreter for ADL, developed by European Space Agency. Available from SIR.
- *Unix utilities in SIR*: All or part of the following set of C programs—flex, grep, gzip, sed, vim, bash, available from SIR.
- *Java programs in SIR*: All or part of the following set of Java programs—siena, ant, jmeter, jtopas, xml-security, nanoxml, available from SIR.
- *Other C/C++/C# programs*: Programs written in C/C++/C# that are not available from SIR.
- *Other Java programs*: Programs written in Java that are not available from SIR.
- *Models and programs in other languages*: models including state machines and UML diagrams. There are also a very few empirical studies that consider programs written in other languages, e.g. Pascal.
- *Web applications*: web applications and web services.

#### ACKNOWLEDGEMENTS

Shin Yoo is supported by the EPSRC SEBASE project (EP/D050863). Mark Harman is supported by EPSRC Grants EP/D050863, GR/S93684 & GR/T22872, by EU grant IST-33472 (EvoTest) and also by the kind support of DaimlerChrysler Berlin and Vizuri Ltd., London.

#### REFERENCES

1. Rothermel G, Harrold MJ. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering* 1996; **22**(8):529–551.
2. Emelie EE, Skoglund M, Runeson P. Empirical evaluations of regression test selection techniques: A systematic review. *ESEM '08: Proceedings of the Second ACM—IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM: New York, NY, U.S.A., 2008; 22–31.
3. Fahad M, Nadeem A. A survey of UML based regression testing. *Intelligent Information Processing* 2008; **288**:200–210.
4. Bible J, Rothermel G, Rosenblum DS. A comparative study of coarse- and fine-grained safe regression test-selection techniques. *ACM Transactions on Software Engineering and Methodology* 2001; **10**(2):149–183.
5. Rosenblum D, Rothermel G. A comparative study of regression test selection techniques. *Proceedings of the 2nd International Workshop on Empirical Studies of Software Maintenance*. IEEE Computer Society Press: Silver Spring, MD, 1997; 89–94.
6. Zhong H, Zhang L, Mei H. An experimental study of four typical test suite reduction techniques. *Information and Software Technology* 2008; **50**(6):534–546.
7. Graves T, Harrold MJ, Kim JM, Porter A, Rothermel G. An empirical study of regression test selection techniques. *Proceedings of the 20th International Conference on Software Engineering (ICSE 1998)*. IEEE Computer Society Press: Silver Spring, MD, 1998; 188–197.
8. Rothermel G, Harrold M, Ronne J, Hong C. Empirical studies of test suite reduction. *Software Testing, Verification, and Reliability* 2002; **4**(2):219–249.
9. Garey MR, Johnson DS. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company: New York, NY, 1979.
10. Leung HKN, White L. Insight into regression testing. *Proceedings of the International Conference on Software Maintenance (ICSM 1989)*. IEEE Computer Society Press: Silver Spring, MD, 1989; 60–69.
11. Chen TY, Lau MF. Dividing strategies for the optimization of a test suite. *Information Processing Letters* 1996; **60**(3):135–141.
12. Harrold MJ, Gupta R, Soffa ML. A methodology for controlling the size of a test suite. *ACM Transactions on Software Engineering and Methodology* 1993; **2**(3):270–285.
13. Horgan J, London S. ATAC: A data flow coverage testing tool for c. *Proceedings of the Symposium on Assessment of Quality Software Development Tools*. IEEE Computer Society Press: Silver Spring, MD, 1992; 2–10.
14. Offutt J, Pan J, Voas J. Procedures for reducing the size of coverage-based test sets. *Proceedings of the 12th International Conference on Testing Computer Software*. ACM Press: New York, 1995; 111–123.

15. Horgan JR, London S. Data flow coverage and the C language. *Proceedings of the Symposium on Testing, Analysis, and Verification (TAV4)*. ACM Press: New York, 1991; 87–97.
16. Papadimitriou CH, Steiglitz K. *Combinatorial Optimization*. Courier Dover Publications: Mineola, NY, 1998.
17. Marré M, Bertolino A. Using spanning sets for coverage testing. *IEEE Transactions on Software Engineering* 2003; **29**(11):974–984.
18. Tallam S, Gupta N. A concept analysis inspired greedy algorithm for test suite minimization. *SIGSOFT Software Engineering Notes* 2006; **31**(1):35–42.
19. Jeffrey D, Gupta N. Test suite reduction with selective redundancy. *Proceedings of the 21st IEEE International Conference on Software Maintenance 2005 (ICSM'05)*. IEEE Computer Society Press: Silver Spring, MD, 2005; 549–558.
20. Jeffrey D, Gupta N. Improving fault detection capability by selectively retaining test cases during test suite reduction. *IEEE Transactions on Software Engineering* 2007; **33**(2):108–123.
21. Black J, Melachrinoudis E, Kaeli D. Bi-criteria models for all-uses test suite reduction. *Proceedings of the 26th International Conference on Software Engineering (ICSE 2004)*. ACM Press: New York, 2004; 106–115.
22. Hsu HY, Orso A. MINTS: A general framework and tool for supporting test-suite minimization. *Proceedings of the 31st International Conference on Software Engineering (ICSE 2009)*. IEEE Computer Society: Silver Spring, MD, 2009; 419–429.
23. Walcott KR, Soffa ML, Kapfhammer GM, Roos RS. Time aware test suite prioritization. *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2006)*. ACM Press: New York, 2006; 1–12.
24. Yoo S, Harman M. Pareto efficient multi-objective test case selection. *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2007)*. ACM Press: New York, 2007; 140–150.
25. McMaster S, Memon A. Call-stack coverage for gui test suite reduction. *IEEE Transactions on Software Engineering* 2008; **34**(1):99–115.
26. Smith A, Geiger J, Kapfhammer GM, Soffa ML. Test suite reduction and prioritization with call trees. *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*. ACM Press: New York, 2007.
27. Harder M, Mellen J, Ernst MD. Improving test suites via operational abstraction. *Proceedings of the 25th International Conference on Software Engineering (ICSE 2003)*. IEEE Computer Society: Silver Spring, MD, 2003; 60–71.
28. Ernst MD, Cockrell J, Griswold WG, Notkin D. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering* 2001; **27**(2):99–123.
29. Leitner A, Oriol M, Zeller A, Ciupa I, Meyer B. Efficient unit test case minimization. *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*. ACM Press: New York, 2007; 417–420.
30. Zeller A. Yesterday, my program worked. Today, it does not why? *SIGSOFT Software Engineering Notes* 1999; **24**(6):253–267.
31. Schroeder PJ, Korel B. Black-box test reduction using input-output analysis. *SIGSOFT Software Engineering Notes* 2000; **25**(5):173–177.
32. Vaysburg B, Tahat LH, Korel B. Dependence analysis in reduction of requirement based test suites. *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2002)*. ACM Press: New York, 2002; 107–111.
33. Korel B, Tahat L, Vaysburg B. Model based regression test reduction using dependence analysis. *Proceedings of the IEEE International Conference on Software Maintenance (ICSM 2002)*. IEEE Computer Society: Silver Spring, MD, 2002; 214–225.
34. Chen Y, Probert RL, Ural H. Regression test suite reduction using extended dependence analysis. *Proceedings of the 4th International Workshop on Software Quality Assurance (SOQUA 2007)*. ACM Press: New York, 2007; 62–69.
35. Anido R, Cavalli AR, Lima Jr LP, Yevtushenko N. Test suite minimization for testing in context. *Software Testing, Verification and Reliability* 2003; **13**(3):141–155.
36. Kaminski GK, Ammann P. Using logic criterion feasibility to reduce test set size while guaranteeing fault detection. *Proceedings of the International Conference on Software Testing, Verification, and Validation 2009 (ICST 2009)*. IEEE Computer Society: Silver Spring, MD, 2009; 356–365.
37. Lau MF, Yu YT. An extended fault class hierarchy for specification-based testing. *ACM Transactions on Software Engineering Methodology* 2005; **14**(3):247–276.
38. Rothermel G, Harrold MJ, Ostrin J, Hong C. An empirical study of the effects of minimization on the fault detection capabilities of test suites. *Proceedings of the International Conference on Software Maintenance (ICSM 1998)*. IEEE Computer Press: Silver Spring, MD, 1998; 34–43.
39. Wong WE, Horgan JR, London S, Mathur AP. Effect of test set minimization on fault detection effectiveness. *Software Practice and Experience* 1998; **28**(4):347–369.
40. Wong WE, Horgan JR, Mathur AP, Pasquini A. Test set size minimization and fault detection effectiveness: A case study in a space application. *The Journal of Systems and Software* 1999; **48**(2):79–89.
41. Hutchins M, Foster H, Goradia T, Ostrand T. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. *Proceedings of the 16th International Conference on Software Engineering (ICSE 1994)*. IEEE Computer Press: Silver Spring, MD, 1994; 191–200.

42. McMaster S, Memon AM. Fault detection probability analysis for coverage-based test suite reduction. *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'07)*. IEEE Computer Society: Silver Spring, MD, 2007.
43. McMaster S, Memon AM. Call stack coverage for test suite reduction. *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*. IEEE Computer Society: Washington, DC, U.S.A., 2005; 539–548.
44. Yu Y, Jones JA, Harrold MJ. An empirical study of the effects of test-suite reduction on fault localization. *Proceedings of the International Conference on Software Engineering (ICSE 2008)*. ACM Press: New York, 2008; 201–210.
45. Rothermel G, Harrold MJ. A framework for evaluating regression test selection techniques. *Proceedings of the 16th International Conference on Software Engineering (ICSE 1994)*. IEEE Computer Press: Silver Spring, MD, 1994; 201–210.
46. Rothermel G, Harrold MJ. A safe, efficient algorithm for regression test selection. *Proceedings of the International Conference on Software Maintenance (ICSM 2003)*. IEEE Computer Press: Silver Spring, MD, 1993; 358–367.
47. Fischer K. A test case selection method for the validation of software maintenance modifications. *Proceedings of the International Computer Software and Applications Conference*. IEEE Computer Press: Silver Spring, MD, 1977; 421–426.
48. Fischer K, Raji F, Chruscicki A. A methodology for retesting modified software. *Proceedings of the National Telecommunications Conference*. IEEE Computer Society Press: Silver Spring, MD, 1981; 1–6.
49. Gupta R, Harrold MJ, Sofya ML. An approach to regression testing using slicing. *Proceedings of the International Conference on Software Maintenance (ICSM 1992)*. IEEE Computer Press: Silver Spring, MD, 1992; 299–308.
50. Harrold MJ, Sofya ML. An incremental approach to unit testing during maintenance. *Proceedings of the International Conference on Software Maintenance (ICSM 1998)*. IEEE Computer Press: Silver Spring, MD, 1988; 362–367.
51. Harrold MJ, Sofya ML. Interprocedual data flow testing. *Proceedings of the 3rd ACM SIGSOFT Symposium on Software Testing, Analysis, and Verification (TAV3)*. ACM Press: New York, 1989; 158–167.
52. Taha AB, Thebaut SM, Liu SS. An approach to software fault localization and revalidation based on incremental data flow analysis. *Proceedings of the International Computer Software and Applications Conference (COMPSAC 1989)*. IEEE Computer Press: Silver Spring, MD, 1989; 527–534.
53. Yau SS, Kishimoto Z. A method for revalidating modified programs in the maintenance phase. *Proceedings of the International Computer Software and Applications Conference (COMPSAC 1987)*. IEEE Computer Press: Silver Spring, MD, 1987; 272–277.
54. Agrawal H, Horgan JR, Krauser EW, London SA. Incremental regression testing. *Proceedings of the International Conference on Software Maintenance (ICSM 1993)*. IEEE Computer Society: Silver Spring, MD, 1993; 348–357.
55. Rothermel G. Efficient, effective regression testing using safe test selection techniques. *PhD Thesis*, University of Clemson, May 1996.
56. Rothermel G, Harrold MJ. Selecting tests and identifying test coverage requirements for modified software. *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 1994)*. ACM Press: New York, 1994; 169–184.
57. Rothermel G, Harrold MJ. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology* 1997; 6(2):173–210.
58. Vokolos F, Frankl P. Pythia: A regression test selection tool based on text differencing. *Proceedings of the International Conference on Reliability Quality and Safety of Software Intensive Systems*. Chapman & Hall, 1997.
59. Vokolos F, Frankl P. Empirical evaluation of the textual differencing regression testing technique. *Proceedings of the IEEE International Conference on Software Maintenance (ICSM 1998)*. IEEE Computer Press: Silver Spring, MD, 1998; 44–53.
60. Bates S, Horwitz S. Incremental program testing using program dependence graphs. *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press: New York, 1993; 384–396.
61. Benedusi P, Cmitle A, De Carlini U. Post-maintenance testing based on path change analysis. *Proceedings of the International Conference on Software Maintenance (ICSM 1988)*. IEEE Computer Press: Silver Spring, MD, 1988; 352–361.
62. Chen YF, Rosenblum D, Vo KP. Testtube: A system for selective regression testing. *Proceedings of the 16th International Conference on Software Engineering (ICSE 1994)*. ACM Press: New York, 1994; 211–220.
63. Leung HKN, White L. Insights into testing and regression testing global variables. *Journal of Software Maintenance* 1990; 2(4):209–222.
64. Leung HKN, White L. A study of integration testing and software regression at the integration level. *Proceedings of the International Conference on Software Maintenance (ICSM 1990)*. IEEE Computer Press: Silver Spring, MD, 1990; 290–301.
65. White LJ, Leung HKN. A firewall concept for both control-flow and data-flow in regression integration testing. *Proceedings of the International Conference on Software Maintenance (ICSM 1992)*. IEEE Computer Press: Silver Spring, MD, 1992; 262–271.

66. White LJ, Narayanswamy V, Friedman T, Kirschenbaum M, Piwowarski P, Oha M. Test manager: A regression testing tool. *Proceedings of the International Conference on Software Maintenance (ICSM 1993)*. IEEE Computer Press: Silver Spring, MD, 338–347.
67. Laski J, Szermer W. Identification of program modifications and its applications in software maintenance. *Proceedings of the International Conference on Software Maintenance (ICSM 1992)*. IEEE Computer Press: Silver Spring, MD, 1992; 282–290.
68. Briand LC, Labiche Y, Buijt K, Soccar G. Automating impact analysis and regression test selection based on UML designs. *Proceedings of the International Conference on Software Maintenance (ICSM 2002)*. IEEE Computer Society: Silver Spring, MD, 2002; 252–261.
69. Briand LC, Labiche Y, He S. Automating regression test selection based on UML designs. *Journal of Information and Software Technology* 2009; **51**(1):16–30.
70. Lee JAN, He X. A methodology for test selection. *The Journal of Systems and Software* 1990; **13**(3):177–185.
71. Hartmann J, Robson DJ. Revalidation during the software maintenance phase. *Proceedings of the International Conference on Software Maintenance (ICSM 1989)*. IEEE Computer Press: Silver Spring, MD, 1989; 70–80.
72. Hartmann J, Robson DJ. Retest-development of a selective revalidation prototype environment for use in software maintenance. *Proceedings of the International Conference on System Sciences*, vol. 2. IEEE Computer Press: Silver Spring, MD, 1990; 92–101.
73. Hartmann J, Robson DJ. Techniques for selective revalidation. *IEEE Software* 1990; **7**(1):31–36.
74. Harrold MJ, Soffa ML. An incremental data flow testing tool. *Proceedings of the 6th International Conference on Testing Computer Software (ICTCS 1989)*. ACM Press: New York, 1989.
75. Wong WE, Horgan JR, London S, Bellcore HA. A study of effective regression testing in practice. *Proceedings of the 8th International Symposium on Software Reliability Engineering (ISSRE 1997)*. IEEE Computer Society: Silver Spring, MD, 1997; 264–275.
76. Fisher II M, Jin D, Rothermel G, Burnett M. Test reuse in the spreadsheet paradigm. *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE 2002)*. IEEE Computer Society: Silver Spring, MD, 2002; 257–268.
77. IEEE Standard Glossary of Software Engineering Terminology. IEEE Press, 10 December 1990.
78. Rothermel G, Harrold MJ. Experience with regression test selection. *Empirical Software Engineering: An International Journal* 1997; **2**(2):178–188.
79. Rothermel G, Harrold MJ. Empirical studies of a safe regression test selection technique. *IEEE Transactions on Software Engineering* 1998; **24**(6):401–419.
80. Ball T. On the limit of control flow analysis for regression test selection. *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 1998)*. ACM Press: New York, 1998; 134–142.
81. Rothermel G, Harrold MJ, Dedhia J. Regression test selection for C++ software. *Software Testing, Verification and Reliability* 2000; **10**(2):77–109.
82. Harrold MJ, Jones JA, Li T, Liang D, Orso A, Pennings M, Sinha S, Spoon S. Regression test selection for Java software. *ACM Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA 2001)*. ACM Press: New York, 2001; 312–326.
83. Xu G, Rountev A. Regression test selection for AspectJ software. *Proceedings of the 29th International Conference on Software Engineering (ICSE 2007)*. IEEE Computer Society: Silver Spring, MD, 2007; 65–74.
84. Zhao J, Xie T, Li N. Towards regression test selection for aspect-oriented programs. *Proceedings of the 2nd Workshop on Testing Aspect-oriented Programs (WTAOP 2006)*. ACM Press: New York, 2006; 21–26.
85. Beydeda S, Gruhn V. Integrating white- and black-box techniques for class-level regression testing. *Proceedings of the 25th IEEE International Computer Software and Applications Conference (COMPSAC 2001)*. IEEE Computer Society Press: Silver Spring, MD, 2001; 357–362.
86. Orso A, Shi N, Harrold MJ. Scaling regression testing to large software systems. *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2004)*. ACM Press: New York, 2004; 241–251.
87. Martins E, Vieira VG. Regression test selection for testable classes. *Dependable Computing—EDCC 2005 (Lecture Notes in Computer Science, vol. 3463/2005)*. 2005; 453–470.
88. Chen Y, Probert RL, Sims DP. Specification-based regression test selection with risk analysis. *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research (CASCON 2002)*. IBM Press, 2002; 1–14.
89. Orso A, Harrold MJ, Rosenblum DS, Rothermel G, Soffa ML, Do H. Using component metadata to support the regression testing of component-based software. *Proceedings of the IEEE International Conference on Software Maintenance (ICSM 2001)*. IEEE Computer Society Press: Silver Spring, MD, 2001.
90. Orso A, Do H, Rothermel G, Harrold MJ, Rosenblum DS. Using component metadata to regression test component-based software: Research articles. *Software Testing, Verification, and Reliability* 2007; **17**(2):61–94.
91. Lin F, Ruth M, Tu S. Applying safe regression test selection techniques to java web services. *Proceedings of the International Conference on Next Generation Web Services Practices (NWESP 2006)*. IEEE Computer Society: Silver Spring, MD, 2006; 133–142.
92. Ruth M, Tu S. A safe regression test selection technique for web services. *Proceedings of the 2nd International Conference on Internet and Web Applications and Services (ICIW 2007)*. IEEE Computer Press: Silver Spring, MD, 2007; 47–47.

93. Ruth M, Tu S. Concurrency issues in automating rts for web services. *Proceedings of the IEEE International Conference on Web Services (ICWS 2007)*. IEEE Computer Press: Silver Spring, MD, 1142–1143.
94. Tarhini A, Fouchal H, Mansour N. Regression testing web services-based applications. *Proceedings of the ACS/IEEE International Conference on Computer Systems and Applications (AICCSA 2006)*. IEEE Computer Press: Silver Spring, MD, 2006; 163–170.
95. Ruth M, Oh S, Loup A, Horton B, Gallet O, Mata M, Tu S. Towards automatic regression test selection for web services. *Proceedings of the 31st International Computer Software and Applications Conference (COMPSAC 2007)*. IEEE Computer Press: Silver Spring, MD, 2007; 729–736.
96. Binkley D. Reducing the cost of regression testing by semantics guided test case selection. *Proceedings of the International Conference on Software Maintenance (ICSM 1995)*. IEEE Computer Society: Silver Spring, MD, 1995; 251–260.
97. Binkley  
D. Semantics guided regression test cost reduction. *IEEE Transactions on Software Engineering* 1997; **23**(8): 498–516.
98. Kung DC, Gao J, Hsia P, Lin J, Toyoshima Y. Class firewall, test order, and regression testing of object-oriented programs. *Journal of Object-oriented Programming* 1995; **8**(2):51–65.
99. White L, Robinson B. Industrial real-time regression testing and analysis using firewalls. *Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM 2004)*. IEEE Computer Press: Silver Spring, MD, 2004; 18–27.
100. White L, Jaber K, Robinson B, Rajlich V. Extended firewall for regression testing: An experience report. *Journal of Software Maintenance and Evolution* 2008; **20**(6):419–433.
101. White L, Almezen H, Sastry S. Firewall regression testing of gui sequences and their interactions. *Proceedings of the IEEE International Conference on Software Maintenance (ICSM 2003)*. IEEE Computer Press: Silver Spring, MD, 2003; 398–409.
102. Zheng J, Robinson B, Williams L, Smiley K. A lightweight process for change identification and regression test selection in using cots components. *Proceedings of the 5th International Conference on Commercial-off-the-shelf (COTS)-based Software Systems (ICCBSS 2006)*. IEEE Computer Society: Silver Spring, MD, 2006; 137–146. DOI: <http://dx.doi.org/10.1109/ICCBSS.2006.1>.
103. Zheng J, Robinson B, Williams L, Smiley K. Applying regression test selection for COTS-based applications. *Proceedings of the 28th International Conference on Software Engineering (ICSE 2006)*. ACM Press: New York, 2006; 512–522.
104. Zheng J, Williams L, Robinson B. Pallino: Automation to support regression test selection for COTS-based applications. *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*. ACM Press: New York, 2007; 224–233.
105. Zheng J, Williams L, Robinson B, Smiley K. Regression test selection for black-box dynamic link library components. *Proceedings of the 2nd International Workshop on Incorporating COTS Software into Software Systems: Tools and Techniques (IWICSS 2007)*. IEEE Computer Society: Silver Spring, MD, 2007; 9–14.
106. Skoglund M, Runeson P. A case study of the class firewall regression test selection technique on a large scale distributed software system. *Proceedings of the International Symposium on Empirical Software Engineering (ISESE 2005)*. IEEE Computer Society Press: Silver Spring, MD, 2005; 74–83.
107. Deng D, Sheu PY, Wang T. Model-based testing and maintenance. *Proceedings of the 6th IEEE International Symposium on Multimedia Software Engineering (MMSE 2004)*. IEEE Computer Society Press: Silver Spring, MD, 2004; 278–285.
108. Pilskalns O, Uyan G, Andrews A. Regression testing uml designs. *Proceedings of the 22nd IEEE International Conference on Software Maintenance (ICSM 2006)*. IEEE Computer Society Press: Silver Spring, MD, 2006; 254–264.
109. Farooq Qua, Iqbal MZZ, Malik ZI, Nadeem A. An approach for selective state machine based regression testing. *Proceedings of the 3rd International Workshop on Advances in Model-based Testing (A-MOST 2007)*. ACM: New York, NY, U.S.A., 2007; 44–52.
110. Le Traon Y, Jeron T, Jezequel JM, Morel P. Efficient object-oriented integration and regression testing. *IEEE Transactions on Reliability* 2000; **49**(1):12–25.
111. Wu Y, Offutt J. Maintaining evolving component-based software with UML. *Proceedings of the 7th European Conference on Software Maintenance and Reengineering (CSMR 2003)*. IEEE Computer Society Press: Silver Spring, MD, 2003; 133–142.
112. Muccini H, Dias M, Richardson DJ. Reasoning about software architecture-based regression testing through a case study. *Proceedings of the 29th International Computer Software and Applications Conference (COMPSAC 2005)*, vol. 2. IEEE Computer Society: Silver Spring, MD, 2005; 189–195.
113. Muccini H, Dias M, Richardson DJ. Software-architecture based regression testing. *The Journal of Systems and Software* 2006; **79**(10):1379–1396.
114. Harrold MJ. Testing evolving software. *The Journal of Systems and Software* 1999; **47**(2–3):173–181.
115. Rothermel G, Untch RH, Chu C, Harrold MJ. Test case prioritization: An empirical study. *Proceedings of the International Conference on Software Maintenance (ICSM 1999)*. IEEE Computer Press: Silver Spring, MD, 1999; 179–188.

116. Elbaum SG, Malishevsky AG, Rothermel G. Prioritizing test cases for regression testing. *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2000)*. ACM Press: New York, 2000; 102–112.
117. Elbaum S, Gable D, Rothermel G. Understanding and measuring the sources of variation in the prioritization of regression test suites. *Proceedings of the Seventh International Software Metrics Symposium (METRICS 2001)*. IEEE Computer Press: Silver Spring, MD, 2001; 169–179.
118. Elbaum SG, Malishevsky AG, Rothermel G. Incorporating varying test costs and fault severities into test case prioritization. *Proceedings of the International Conference on Software Engineering (ICSE 2001)*. ACM Press: New York, 2001; 329–338.
119. Malishevsky A, Rothermel G, Elbaum S. Modeling the cost-benefits tradeoffs for regression testing techniques. *Proceedings of the International Conference on Software Maintenance (ICSM 2002)*. IEEE Computer Press: Silver Spring, MD, 2002; 230–240.
120. Rothermel G, Elbaum S, Malishevsky A, Kallakuri P, Davia B. The impact of test suite granularity on the cost-effectiveness of regression testing. *Proceedings of the 24th International Conference on Software Engineering (ICSE 2002)*. ACM Press: New York, 2002; 130–140.
121. Rothermel G, Untch RJ, Chu C. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering* 2001; **27**(10):929–948.
122. Budd TA. Mutation analysis of program test data. *PhD Thesis*, Yale University, New Haven, CT, U.S.A., 1980.
123. Elbaum S, Malishevsky A, Rothermel G. Test case prioritization: A family of empirical studies. *IEEE Transactions on Software Engineering* 2002; **28**(2):159–182.
124. Jones JA, Harrold MJ. Test-suite reduction and prioritization for modified condition/decision coverage. *Proceedings of the International Conference on Software Maintenance (ICSM 2001)*. IEEE Computer Press: Silver Spring, MD, 2001; 92–101.
125. Chilenski J, Miller S. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal* 1994; **9**(5):193–200.
126. Srivastava A, Thiagarajan J. Effectively prioritizing tests in development environment. *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2002)*. ACM Press: New York, 2002; 97–106.
127. Do H, Rothermel G, Kinneer A. Empirical studies of test case prioritization in a junit testing environment. *Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE 2004)*. IEEE Computer Press: Silver Spring, MD, 2004; 113–124.
128. Li Z, Harman M, Hierons RM. Search algorithms for regression test case prioritization. *IEEE Transactions on Software Engineering* 2007; **33**(4):225–237.
129. Bryce RC, Colbourn CJ, Cohen MB. A framework of greedy methods for constructing interaction test suites. *Proceedings of the 27th International Conference on Software Engineering (ICSE 2005)*. ACM Press: New York, 2005; 146–155.
130. Cohen MB, Dwyer MB, Shi J. Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach. *IEEE Transactions on Software Engineering* 2008; **34**(5):633–650.
131. Bryce RC, Colbourn CJ. Test prioritization for pairwise interaction coverage. *Proceedings of the ACM Workshop on Advances in Model-based Testing (A-MOST 2005)*. ACM Press: New York, 2005; 1–7.
132. Bryce RC, Colbourn CJ. Prioritized interaction testing for pair-wise coverage with seeding and constraints. *Journal of Information and Software Technology* 2006; **48**(10):960–970.
133. Qu X, Cohen MB, Woolf KM. Combinatorial interaction regression testing: A study of test case generation and prioritization. *Proceedings of the IEEE International Conference on Software Maintenance (ICSM 2007)*. IEEE Computer Press: Silver Spring, MD, 2007; 255–264.
134. Qu X, Cohen MB, Rothermel G. Configuration-aware regression testing: An empirical study of sampling and prioritization. *Proceedings of the ACM International Symposium on Software Testing and Analysis (ISSTA 2008)*. ACM Press: New York, 2008; 75–86.
135. Bryce RC, Memon AM. Test suite prioritization by interaction coverage. *Proceedings of the Workshop on Domain Specific Approaches to Software Test Automation (DOSTA 2007)*. ACM: New York, 2007; 1–7.
136. Leon D, Podgurski A. A comparison of coverage-based and distribution-based techniques for filtering and prioritizing test cases. *Proceedings of the IEEE International Symposium on Software Reliability Engineering (ISSRE 2003)*. IEEE Computer Press: Silver Spring, MD, 2003; 442–456.
137. Tonella P, Avesani P, Susi A. Using the case-based ranking methodology for test case prioritization. *Proceedings of the 22nd International Conference on Software Maintenance (ICSM 2006)*. IEEE Computer Society: Silver Spring, MD, 2006; 123–133.
138. Yoo S, Harman M, Tonella P, Susi A. Clustering test cases to achieve effective & scalable prioritisation incorporating expert knowledge. *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2009)*. ACM Press: New York, 2009; 201–211.
139. Srikanth H, Williams L, Osborne J. System test case prioritization of new and regression test cases. *Proceedings of the International Symposium on Empirical Software Engineering*. IEEE Computer Society Press: Silver Spring, MD, 2005; 64–73.
140. Freund Y, Schapire R. A short introduction to boosting. *Journal of Japanese Society for Artificial Intelligence* 1999; **14**(5):771–780.

141. Freund Y, Iyer R, Schapire RE, Singer Y. An efficient boosting algorithm for combining preferences. *Proceedings of the 15th International Conference on Machine Learning (ICML 1998)*, Shavlik JW (ed.). Morgan Kaufmann Publishers: Los Altos, CA, 1998; 170–178.
142. Saaty T. *The Analytic Hierarchy Process, Planning, Priority Setting, Resource Allocation*. McGraw-Hill: New York, NY, U.S.A., 1980.
143. Karlsson J, Wohlin C, Regnell B. An evaluation of methods for prioritizing software requirements. *Information and Software Technology* 1998; **39**(14–15):939–947.
144. Kim JM, Porter A. A history-based test prioritization technique for regression testing in resource constrained environments. *Proceedings of the 24th International Conference on Software Engineering (ICSE 2002)*. ACM Press: New York, 2002; 119–129.
145. Mirarab S, Tahvildari L. A prioritization approach for software test cases based on Bayesian networks. *Proceedings of the 10th International Conference on Fundamental Approaches to Software Engineering*. Springer: Berlin, 2007; 276–290.
146. Mirarab S, Tahvildari L. An empirical study on Bayesian network-based approach for test case prioritization. *Proceedings of the International Conference on Software Testing, Verification and Validation*. IEEE Computer Society: Silver Spring, MD, 2008; 278–287.
147. Sherriff M, Lake M, Williams L. Prioritization of regression tests using singular value decomposition with empirical change records. *Proceedings of the 18th IEEE International Symposium on Software Reliability (ISSRE 2007)*. IEEE Computer Society: Washington, DC, U.S.A., 2007; 81–90.
148. Krishnamoorthi R, Sahaaya Arul Mary SA. Factor oriented requirement coverage based system test case prioritization of new and regression test cases. *Information and Software Technology* 2009; **51**(4):799–808.
149. Korel B, Tahat L, Harman M. Test prioritization using system models. *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM 2005)*, 2005; 559–568.
150. Korel B, Koutsogiannakis G, Tahat LH. Model-based test prioritization heuristic methods and their evaluation. *Proceedings of the 3rd International Workshop on Advances in Model-based Testing (A-MOST 2007)*. ACM Press: New York, 2007; 34–43.
151. Korel B, Koutsogiannakis G, Tahat L. Application of system models in regression test suite prioritization. *Proceedings of the IEEE International Conference on Software Maintenance 2008 (ICSM 2008)*. IEEE Computer Society Press: Silver Spring, MD, 2008; 247–256.
152. Hou SS, Zhang L, Xie T, Mei H, Sun JS. Applying interface-contract mutation in regression testing of component-based software. *Proceedings of the 23rd IEEE International Conference on Software Maintenance (ICSM 2007)*. IEEE Computer Society Press: Silver Spring, MD, 2007; 174–183.
153. Sampath S, Bryce RC, Viswanath G, Kandimalla V, Koru AG. Prioritizing user-session-based test cases for web applications testing. *Proceedings of the 1st International Conference on Software Testing Verification and Validation (ICST 2008)*. IEEE Computer Society: Silver Spring, MD, 2008; 141–150.
154. Fraser G, Wotawa F. Test-case prioritization with model-checkers. *SE'07: Proceedings of the 25th Conference on IASTED International Multi-conference*. ACTA Press: Anaheim, CA, U.S.A., 2007; 267–272.
155. Fraser G, Wotawa F. Property relevant software testing with model-checkers. *SIGSOFT Software Engineering Notes* 2006; **31**(6):1–10.
156. Rummel M, Kapfhammer GM, Thall A. Towards the prioritization of regression test suites with data flow information. *Proceedings of the 20th Symposium on Applied Computing (SAC 2005)*. ACM Press: New York, 2005.
157. Jeffrey D, Gupta N. Test case prioritization using relevant slices. *Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC 2006)*. IEEE Computer Society: Washington, DC, U.S.A., 2006; 411–420.
158. Do H, Mirarab SM, Tahvildari L, Rothermel G. An empirical study of the effect of time constraints on the cost-benefits of regression testing. *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM Press: New York, 2008; 71–82.
159. Hou SS, Zhang L, Xie T, Sun JS. Quota-constrained test-case prioritization for regression testing of service-centric systems. *Proceedings of the IEEE International Conference on Software Maintenance (ICSM 2008)*. IEEE Computer Society Press: Silver Spring, MD, 2008.
160. Zhang L, Hou SS, Guo C, Xie T, Mei H. Time-aware test-case prioritization using Integer Linear Programming. *Proceedings of the International Conference on Software Testing and Analysis (ISSTA 2009)*. ACM Press: New York, 2009; 212–222.
161. Andrews JH, Briand LC, Labiche Y. *Proceedings of the 27th International Conference on Software Engineering (ICSE 2005)*. ACM Press: New York, 2005; 402–411.
162. Do H, Rothermel G. A controlled experiment assessing test case prioritization techniques via mutation faults. *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM 2005)*. IEEE Computer Society Press: Silver Spring, MD, 2005; 411–420.
163. Do H, Rothermel G. On the use of mutation faults in empirical assessments of test case prioritization techniques. *IEEE Transactions on Software Engineering* 2006; **32**(9):733–752.
164. Elbaum S, Kallakuri P, Malishevsky A, Rothermel G, Kanduri S. Understanding the effects of changes on the cost-effectiveness of regression testing techniques. *Software Testing, Verification, and Reliability* 2003; **13**(2):65–83.

165. Rothermel G, Elbaum S, Malishevsky A, Kallakuri P, Qiu X. On test suite composition and cost-effective regression testing. *ACM Transactions on Software Engineering and Methodology* 2004; **13**(3):277–331.
166. Kim JM, Porter A, Rothermel G. An empirical study of regression test application frequency. *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000)*. ACM Press: New York, 2000; 126–135.
167. Kim JM, Porter A, Rothermel G. An empirical study of regression test application frequency. *Software Testing, Verification, and Reliability* 2005; **15**(4):257–279.
168. Elbaum S, Rothermel G, Kanduri S, Malishevsky AG. Selecting a cost-effective test case prioritization technique. *Software Quality Control* 2004; **12**(3):185–210.
169. Rosenblum D, Weyuker E. Using coverage information to predict the cost-effectiveness of regression testing strategies. *IEEE Transactions on Software Engineering* 1997; **23**(3):146–156.
170. Bolksy MI, Korn DG. *The New KornShell Command and Programming Language*. Prentice-Hall PTR: Upper Saddle River, NJ, U.S.A., 1995.
171. Korn D, phong Vo K. SFIO: Safe/Fast String/File IO. *Proceedings of the Summer Usenix Conference 1991*, 1991; 235–256.
172. Leung HKN, White L. A cost model to compare regression test strategies. *Proceedings of the International Conference on Software Maintenance (ICSM 1991)*. IEEE Computer Press: Silver Spring, MD, 1991; 201–208.
173. Harrold MJ, Rosenblum DS, Rothermel G, Weyuker EJ. Empirical studies of a prediction model for regression test selection. *IEEE Transactions on Software Engineering* 2001; **27**(3):248–263.
174. Ramey C, Fox B. *Bash Reference Manual* (2.2 edn). O'Reilly and Associates: Sebastopol, CA, 1998.
175. Do H, Rothermel G, Kinneer A. Prioritizing junit test cases: An empirical assessment and cost-benefits analysis. *Empirical Software Engineering* 2006; **11**(1):33–70.
176. Smith AM, Kapfhammer GM. An empirical study of incorporating cost into test suite reduction and prioritization. *Proceedings of the 24th Symposium on Applied Computing (SAC 2009)*. ACM Press: New York, 2009.
177. Do H, Rothermel G. An empirical study of regression testing techniques incorporating context and lifetime factors and improved cost-benefit models. *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM Press: New York, 2006; 141–151.
178. Do H, Rothermel G. Using sensitivity analysis to create simplified economic models for regression testing. *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008)*. ACM Press: New York, 2008; 51–61.
179. Do H, Elbaum SG, Rothermel G. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering* 2005; **10**(4):405–435.
180. Memon AM, Soffa ML. Regression testing of GUIs. *ESEC/FSE-11: Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM: New York, NY, U.S.A., 2003; 118–127.
181. Jia Y, Harman M. Constructing subtle faults using higher order mutation testing. *Proceedings of the IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2008)*. IEEE Computer Society Press: Silver Spring, MD, 2008; 249–258.
182. Harrold M, Orso A. Retesting software during development and maintenance. *Frontiers of Software Maintenance (FoSOM 2008)*. IEEE Computer Society Press: Silver Spring, MD, 2008; 99–108.
183. McMinn P. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability* 2004; **14**(2):105–156.
184. Apiwattanapong T, Santelices R, Chittimalli PK, Orso A, Harrold MJ. Matrix: Maintenance-oriented testing requirements identifier and examiner. *Proceedings of the Testing: Academic and Industrial Conference on Practice And Research Techniques*. IEEE Computer Society: Los Alamitos, CA, U.S.A., 2006; 137–146.
185. Santelices RA, Chittimalli PK, Apiwattanapong T, Orso A, Harrold MJ. Test-suite augmentation for evolving software. *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*. IEEE: New York, 2008; 218–227.
186. Memon AM. Automatically repairing event sequence-based gui test suites for regression testing. *ACM Transactions on Software Engineering Methodology* 2008; **18**(2):1–36.
187. Alshahwan N, Harman M. Automated session data repair for web application regression testing. *Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*. IEEE Computer Society: Los Alamitos, CA, U.S.A., 2008; 298–307.
188. Bertolini C, Peres G, d'Amorim M, Mota A. An empirical evaluation of automated black box testing techniques for crashing guis. *Proceedings of the 2nd International Conference on Software Testing Verification and Validation (ICST 2009)*. IEEE Computer Press: Silver Spring, MD, 2009; 21–30.
189. Fu C, Grechanik M, Xie Q. Inferring types of references to gui objects in test scripts. *Proceedings of the 2nd International Conference on Software Testing Verification and Validation (ICST 2009)*. IEEE Computer Press: Silver Spring, MD, 2009; 1–10.



**University of  
Zurich<sup>UZH</sup>**

**Zurich Open Repository and  
Archive**  
University of Zurich  
Main Library  
Strickhofstrasse 39  
CH-8057 Zurich  
[www.zora.uzh.ch](http://www.zora.uzh.ch)

---

Year: 2018

---

## **Modern code review: a case study at google**

Sadowski, Caitlin ; Söderberg, Emma ; Church, Luke ; Sipko, Michal ; Bacchelli, Alberto

DOI: <https://doi.org/10.1145/3183519.3183525>

Posted at the Zurich Open Repository and Archive, University of Zurich

ZORA URL: <https://doi.org/10.5167/uzh-152982>

Conference or Workshop Item

Published Version



The following work is licensed under a Creative Commons: Attribution-NonCommercial-NoDerivatives 4.0 International (CC BY-NC-ND 4.0) License.

Originally published at:

Sadowski, Caitlin; Söderberg, Emma; Church, Luke; Sipko, Michal; Bacchelli, Alberto (2018). Modern code review: a case study at google. In: the 40th International Conference Conference on Software Engineering: Software Engineering in Practice (ICSE SEiP), Gothenburg, Sweden, 27 June 2018 - 3 July 2018, 181-190.

DOI: <https://doi.org/10.1145/3183519.3183525>

# Modern Code Review: A Case Study at Google

Caitlin Sadowski, Emma Söderberg,  
Luke Church, Michal Sipko  
Google, Inc.  
[{supertri,emso,lukechurch,sipkom}@google.com](mailto:{supertri,emso,lukechurch,sipkom}@google.com)

Alberto Bacchelli  
University of Zurich  
[bacchelli@ifi.uzh.ch](mailto:bacchelli@ifi.uzh.ch)

## ABSTRACT

Employing lightweight, tool-based code review of code changes (aka *modern code review*) has become the norm for a wide variety of open-source and industrial systems. In this paper, we make an exploratory investigation of modern code review at Google. Google introduced code review early on and evolved it over the years; our study sheds light on why Google introduced this practice and analyzes its current status, after the process has been refined through decades of code changes and millions of code reviews. By means of 12 interviews, a survey with 44 respondents, and the analysis of review logs for 9 million reviewed changes, we investigate motivations behind code review at Google, current practices, and developers' satisfaction and challenges.

## CCS CONCEPTS

- Software and its engineering → Software maintenance tools;

### ACM Reference format:

Caitlin Sadowski, Emma Söderberg,  
Luke Church, Michal Sipko and Alberto Bacchelli. 2018. Modern Code Review: A Case Study at Google. In *Proceedings of 40th International Conference on Software Engineering: Software Engineering in Practice Track, Gothenburg, Sweden, May 27-June 3, 2018 (ICSE-SEIP '18)*, 10 pages.

DOI: [10.1145/3183519.3183525](https://doi.org/10.1145/3183519.3183525)

## 1 INTRODUCTION

Peer code review, a manual inspection of source code by developers other than the author, is recognized as a valuable tool for improving the quality of software projects [2, 3]. In 1976, Fagan formalized a highly structured process for code reviewing—code inspections [16]. Over the years, researchers provided evidence on the benefits of code inspection, especially for defect finding, but the cumbersome, time-consuming, and synchronous nature of this approach hindered its universal adoption in practice [37]. Nowadays, most organizations adopt more lightweight code review practices to limit the inefficiencies of inspections [33]. Modern code review is (1) informal (in contrast to Fagan-style), (2) tool-based [32], (3) asynchronous, and (4) focused on reviewing code changes.

An open research challenge is understanding which practices represent valuable and effective methods of review in this

novel context. Rigby and Bird quantitatively analyzed code review data from software projects spanning varying domains as well as organizations and found five strongly convergent aspects [33], which they conjectured can be prescriptive to other projects. The analysis of Rigby and Bird is based on the value of a *broad* perspective (that analyzes multiple projects from different contexts). For the development of an empirical body of knowledge, championed by Basili [7], it is essential to also consider a *focused and longitudinal* perspective that analyzes a single case. This paper expands on work by Rigby and Bird to focus on the review practices and characteristics established at Google, *i.e.*, a company with a multi-decade history of code review and a high-volume of daily reviews to learn from. This paper can be (1) prescriptive to practitioners performing code review and (2) compelling for researchers who want to understand and support this novel process.

Code review has been a required part of software development at Google since very early on in the company's history; because it was introduced so early on, it has become a core part of Google culture. The process and tooling for code review at Google have been iteratively refined for more than a decade and is applied by more than 25,000 developers making more than 20,000 source code changes each workday, in dozens of offices around the world [30].

We conduct our analysis in the form of an exploratory investigation focusing on three aspects of code review, in line with and expanding on the work by Rigby and Bird [33]: (1) The motivations driving code review, (2) the current practices, and (3) the perception of developers on code review, focusing on challenges encountered with a specific review (*breakdowns* in the review process) and satisfaction. Our research method combines input from multiple data sources: 12 semi-structured interviews with Google developers, an internal survey sent to engineers who recently sent changes to review with 44 responses, and log data from Google's code review tool pertaining to 9 million reviews over two years.

We find that the process at Google is markedly lighter weight than in other contexts, based on a single reviewer, quick iterations, small changes, and a tight integration with the code review tool. Breakdowns still exist, however, due to the complexity of the interactions that occur around code review. Nevertheless, developers consider this process valuable, confirm that it works well at scale, and conduct it for several reasons that also depend on the relationship between author and reviewers. Finally, we find evidence on the use of the code review tool beyond collaborative review and corroboration for the importance of code review as an educational tool.

*ICSE-SEIP '18, Gothenburg, Sweden*

© 2018 Copyright held by the owner/author(s). This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *Proceedings of 40th International Conference on Software Engineering: Software Engineering in Practice Track, May 27-June 3, 2018*, <http://dx.doi.org/10.1145/3183519.3183525>.

## 2 BACKGROUND AND RELATED WORK

We describe the review processes investigated in literature, then we detail convergent code review practices across these processes [33].

### 2.1 Code Review Processes and Contexts

**Code Inspections.** Software inspections are one of the first formalized processes for code review. This highly structured process involves planning, overview, preparation, inspection meeting, reworking, and follow-up [16]. The goal of code inspections is to find defects during a synchronized inspection meeting, with authors and reviewers sitting in the same room to examine code changes. Kollanus and Koskinen compiled the most recent literature survey on code inspection research [25]. They found that the vast majority of studies on code inspections are empirical in nature. There is a consensus about the overall value of code inspection as a defect finding technique and the value of reading techniques to engage the inspectors. Overall, research on code inspection has declined from 2005, in line with the spread of the internet and the growth of asynchronous code review processes.

**Asynchronous review via email.** Until the late 2000s, most large OSS projects adopted a form of remote, asynchronous reviews, relying on patches sent to communication channels such as mailing lists and issue tracking systems. In this setting, project members evaluate contributed patches and ask for modifications through these channels. When a patch is deemed of high enough quality, core developers commit it to the codebase. Trusted committers may have a commit-then-review process instead of doing pre-commit reviews [33]. Rigby *et al.* were among the first to do extensive work in this setting; they found that this type of review “*has little in common* [with code inspections] *beyond a belief that peers will effectively find software defects*” [34]. Kononenko *et al.* analyzed the same setting and find that review response time and acceptance are related to social factors, such as reviewer load and change author experience [26], which were not present in code inspections.

**Tool-based review.** To bring structure to the process of reviewing patches, several tools emerged in OSS and industrial settings. These tools support the *logistics* of the review process: (1) The author of a patch submits it to the code review tool, (2) the reviewers can see the diff of the proposed code change and (3) can start a threaded discussion on specific lines with the author and other reviewers, and then (4) the author can propose modifications to address reviewers’ comments. This feedback cycle continues until everybody is satisfied or the patch is discarded. Different projects adapted their tools to support their process. Microsoft uses **CodeFlow**, which tracks the state of each person (author or reviewer) and where they are in the process (signed off, waiting, reviewing); CodeFlow does not prevent authors from submitting changes without approval [33] and supports chats in comment threads [4]. Google’s Chromium project (along with several other OSS projects) relies on the externally-available **Gerrit** [17]; in Chromium, changes are only merged into the

Table 1: Convergent review practices [33].

id	Convergent Practice
CP <sub>1</sub>	Contemporary peer review follows a lightweight, flexible process
CP <sub>2</sub>	Reviews happen early (before a change is committed), quickly, and frequently
CP <sub>3</sub>	Change sizes are small
CP <sub>4</sub>	Two reviewers find an optimal number of defects
CP <sub>5</sub>	Review has changed from a defect finding activity to a group problem solving activity

master branch after explicit approval from reviewers and automated verification that the change does not break the build [12]. In Gerrit, unassigned reviewers can also make comments. VMware developed the open-source **ReviewBoard**, which integrates static analysis into the review process; this integration relies on change authors manually requesting analysis and has been shown to improve code review quality [5]. Facebook’s code review system, **Phabricator** [29], allows reviewers to “take over” a change and commit it themselves and provides hooks for automatic static analysis or continuous build/test integration.

In the context of tool-based reviews, researchers have investigated the relationship between code change acceptance or response time and features of the changed code and authors [41], as well as the agreement among reviewers [22]. Qualitative investigations have been also conducted to define what constitutes a good code review according to industrial [11] and OSS developers [26].

**Pull-based development model.** In the GitHub pull request process [18] a developer wanting to make a change forks an existing git repository and then makes changes in their fork. After a pull request has been sent out, it appears in the list of pull requests for the project in question, visible to anyone who can see the project. Gousios *et al.* qualitatively investigated work practices and challenges of pull-request integrators [21] and contributors [20], finding analogies to other tool-based code reviews.

### 2.2 Convergent Practices in Code Review

Rigby and Bird presented the first and most significant work that tries to identify convergent practices across several code review processes and contexts [33]. They considered OSS projects that use email based reviews, OSS projects that use Gerrit, an AMD project that uses a basic code review tool, and Microsoft with CodeFlow. They analyzed the process and the data available from these projects to describe several angles, such as iterative development, reviewer selection, and review discussions. They identified five practices of modern code review to which all the considered projects converged (Table 1). We will refer to these practices using their id, *e.g.*, CP<sub>1</sub>. Substantially, they found an agreement in terms of a quick, lightweight process (CP<sub>1</sub>,CP<sub>2</sub>,CP<sub>3</sub>) with few people involved (CP<sub>4</sub>) who conduct group problem solving (CP<sub>5</sub>).

### 3 METHODOLOGY

This section describes our research questions and settings; it also outlines our research method and its limitations.

#### 3.1 Research Questions

The overall goal of this research is to investigate modern code review at Google, which is a process that involves thousands of developers and that has been refined over more than a decade. To this aim, we conduct an exploratory investigation that we structure around three main research questions.

**RQ<sub>1</sub>: What are the motivations for code review at Google?**

Rigby and Bird found the motivations to be one of the converging traits of modern code review (CP<sub>5</sub>). Here we study what motivations and expectations drive code reviews at Google. In particular, we consider both the historical reasons for introducing modern code review (since Google is one of the first companies that used modern code review) and the current expectations.

**RQ<sub>2</sub>: What is the practice of code review at Google?**

The other four convergent practices found by Rigby and Bird regard how the process itself is executed, in terms of flow (CP<sub>1</sub>), speed and frequency (CP<sub>2</sub>), size of the analyzed changes (CP<sub>3</sub>), and number of reviewers (CP<sub>4</sub>). We analyze these aspects at Google to investigate whether the same findings hold for a company that has a longer history of code review, an explicit culture, and larger volume of reviews compared to those analyzed in previous studies [4, 33].

**RQ<sub>3</sub>: How do Google developers perceive code review?**

Finally, in our last research question, we are interested in understanding how Google developers perceive modern code review as implemented in their company. This exploration is needed to better understand practices (since perceptions drive behavior [39]) and to guide future research. We focus on two aspects: *breakdowns* of the review process developers experience during specific reviews and whether developers are *satisfied* with review despite these challenges.

#### 3.2 Research Setting

We briefly describe our research setting for context on our methodology. See Section 5.1 for a comprehensive description of Google's code review process and tooling.

Most software development at Google occurs in a monolithic source repository, accessed via an internal version control system [30]. Since code review is required at Google, every commit to Google's source control system goes first through code review using CRITIQUE: an internally developed, centralized, web-based code review tool. The development workflow in Google's monolithic repository, including the code review process, is very uniform. As with other tools described in Section 2, CRITIQUE allows reviewers to see a diff of the proposed code change and start a threaded discussion on specific lines with the author and other reviewers. CRITIQUE offers extensive logging functionalities; all developer interactions with the tool are captured (including opening the tool, viewing a diff, making comments, and approving changes).

#### 3.3 Research Method

To answer our research questions, we follow a mixed qualitative and quantitative approach [13], which combines data from several sources: semi-structured interviews with employees involved in software development at Google, logs from the code review tool, and a survey to other employees. We use the interviews as a tool to collect data on the diversity (as opposed to frequencies [23]) of the motivations for conducting code review (RQ<sub>1</sub>) and to elicit developers' perceptions of code review and its challenges (RQ<sub>3</sub>). We use CRITIQUE logs to quantify and describe the current review practices (RQ<sub>2</sub>). Finally, we use the survey to confirm the diverse motivations for code review that emerged from the interviews (RQ<sub>1</sub>) and elicit the developers' satisfaction with the process.

**Interviews.** We conducted a series of face-to-face semi-structured [13] interviews with selected Google employees, each taking approximately 1 hour. The initial pool of possible participants was selected using snowball sampling, starting with developers known to the paper authors. From this pool, participants were selected to ensure a spread of teams, technical areas, job roles, length of time within the company, and role in the code review process. The interview script (in appendix [1]) included questions about perceived motivations for code review, a recently reviewed/authored change, and best/worst review experiences. Before each interview we reviewed the participant's review history and located a change to discuss in the interview; we selected these changes based on the number of interactions, the number of people involved in the conversation and whether there were comments that seemed surprising. In an observation part of the interview, the participant was asked to think-aloud while reviewing the pending change and to provide some explicit information, such as the entry point for starting the review. The interviews continued until saturation [19] was achieved and interviews were bringing up broadly similar concepts. Overall, we conducted 12 interviews with staff who had been working at Google from 1 month to 10 years (median 5 years), in Software Engineering and Site Reliability Engineering. They included technical leads, managers and individual contributors. Each interview involved three to four people: The participant and 2-3 interviewees (two of which are authors of this paper). Interviews were live-transcribed by one of the interviewers while another one asked questions.

**Open Coding on Interview Data.** To identify the broad themes emerging from the interview data we performed an open coding pass [27]. Interview transcripts were discussed by two authors to establish common themes, then converted into a coding scheme. An additional author then performed a closed coding of the notes of the discussions to validate the themes. We iterated this process over one of the interviews until we had agreement on the scheme. We also tracked in what context (relationship between reviewer and author) these themes were mentioned. The combination of the design of the questions and the analysis process means that we can discuss stable themes in the results, but cannot meaningfully discuss relative frequencies of occurrence [23].

**Analysis of Review Data.** We analyzed quantitative data about the code review process by using the logs produced by CRITIQUE. We mainly focus on metrics related to convergent practices found by Rigby and Bird [33]. To allow for comparison, we do not consider changes without any reviewers, as we are interested in changes that have gone through an explicit code review process. We consider a “reviewer” to be any user who approves a code change, regardless of whether they were explicitly asked for a review by the change author. We use a name-based heuristic to filter out changes made by automated processes. We focus exclusively on changes that occur in the main codebase at Google. We also exclude changes not yet committed at the time of study and those for which our diff tool reports a delta of zero source lines changed, *e.g.*, a change that only modifies binary files. On an average workday at Google, about 20,000 changes are committed that meet the filter criteria described above. Our final dataset includes the approximately 9 million changes created by more than 25,000 authors and reviewers from January 2014 until July 2016 that meet these criteria, and about 13 million comments collected from all changes between September 2014 and July 2016.

**Survey.** We created an online questionnaire that we sent to 98 engineers who recently submitted a code change. The code change had already been reviewed, so we customized the questionnaire to ask respondents about how they perceived the code review for *their specific recent change*; this strategy allowed us to mitigate recall bias [35], yet collect comprehensive data. The survey consisted of three Likert scale questions on the value of the received review, one multiple choice on the effects of the review on their change (based on the expectations that emerged from the interviews) with an optional ‘other’ response, and one open-ended question eliciting the respondent’s opinion on the received review, the code review tool, and/or the process in general. We received 44 valid responses to our survey (45% response rate, which is considered high for software engineering research [31]).

### 3.4 Threats to Validity and Limitations

We describe threats to validity and limitations of the results of our work, as posed by our research method, and the actions that we took to mitigate them.

**Internal validity - Credibility.** Concerning the quantitative analysis of review data, we use heuristics to filter out robot-authored changes from our quantitative analysis, but these heuristics may allow some robot authored changes; we mitigated this as we only include robot-authored changes that have a human reviewer. Concerning the qualitative investigation, we used open coding to analyze the interviewees’ answers. This coding could have been influenced by the experience and motivations of the authors conducting it, although we tried to mitigate this bias by involving multiple coders. The employees that decided to participate in our interviews and the survey freely decided to do so, thus introducing the risk of *self-selection bias*. For this reason, results may have been different for developers who would not choose to participate;

to try to mitigate this issue, we combine information from both interviews and survey. Moreover, we used a snowball sampling method to identify engineers to interview, this is at the risk of *sampling bias*. Although we attempted to mitigate this risk by interviewing developers with a range of job roles and responsibilities, there may be other factors the developers we interviewed share that would not apply across the company. To mitigate *moderator acceptance bias*, the researchers involved in the qualitative data collection were not part of the CRITIQUE team. Social desirability bias may have influenced the answers to align more favorably to Google culture; however, at Google people are encouraged to criticize and improve broken workflows when discovered, thus reducing this bias. Finally, we did not interview research scientists or developers that interact with specialist reviewers (such as security reviews), thus our results are biased towards general developers.

**Generalizability - Transferability.** We designed our study with the stated aim of understanding modern code review within a specific company. For this reason, our results may not generalize to other contexts, rather we are interested in the diversity of the practices and breakdowns that are still occurring after years and millions of reviews of refinement. Given the similarity of the underlying code review mechanism across several companies and OSS projects, it is reasonable to think that should a review process reach the same level of maturity and use comparable tooling, developers would have similar experiences.

## 4 RESULTS: MOTIVATIONS

In our first research question, we seek to understand motivations and expectations of developers when conducting code review at Google, starting by investigating what led to the introduction of this process in the first place.

### 4.1 How it All Started

Code review at Google was introduced early on by one of the first employees; the first author of this paper interviewed this employee (referred to as *E* in the following) to better understand the initial motivation of code review and its evolution. *E* explained that the main impetus behind the introduction of code review was *to force developers to write code that other developers could understand*; this was deemed important since code must act as a teacher for future developers. *E* stated that the introduction of code review at Google signaled the transition from a research codebase, which is optimized towards quick prototyping, to a production codebase, where it is critical to think about future engineers reading source code. Code review was also perceived as capable of ensuring that more than one person would be familiar with each piece of code, thus increasing the chances of knowledge staying within the company.

*E* reiterated on the concept that, although it is great if reviewers find bugs, the foremost reason for introducing code review at Google was to improve code understandability and

maintainability. However, in addition to the initial educational motivation for code review, *E* explained that three additional benefits soon became clear to developers internally: checking the consistency of style and design; ensuring adequate tests; and improving security by making sure no single developer can commit arbitrary code without oversight.

## 4.2 Current expectations

By coding our interview data, we identified four key themes for what Google developers expect from code reviews: **education**, **maintaining norms**, **gatekeeping**, and **accident prevention**. Education regards either teaching or learning from a code review and is in line with the initial reasons for introducing code review; norms refer to an organization preference for a discretionary choice (*e.g.*, formatting or API usage patterns); gatekeeping concerns the establishment and maintenance of boundaries around source code, design choices or another artifact; and accidents refer to the introduction of bugs, defects or other quality related issues.

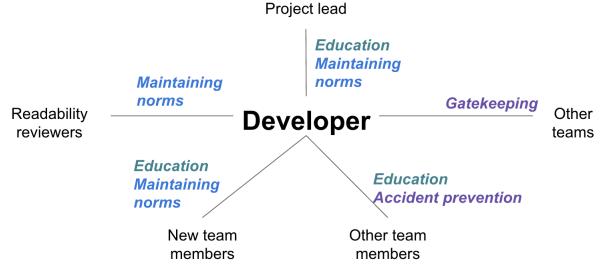
These are the main themes *during* the review process, but code review is also used *retrospectively* for **tracking history**. Developers value the review process after it has taken place; code review enables the ability to browse the history of a code change, including what comments occurred and how the change evolved. We also noticed developers using code review history to understand how bugs were introduced. Essentially, code review enables future auditing of changes.

In our survey, we further validated this coding scheme. We asked what authors found valuable about code review comments; they could select one or more of the four themes and/or write in their own. Each of the four themes identified earlier was selected by between 8 and 11 respondents in the context of a particular code review, thus providing additional confidence that the coding scheme described above aligns with developers' perception of the value of the code review.

Although these expectations can map over those found previously at Microsoft [4], the main focus at Google, as explained by our participants, is on education as well as code readability and understandability, in line with the historical impetus for review. For this reason, the focus does not align with that found by Rigby and Bird (*i.e.*, a group problem solving activity) [33].

**Finding 1.** *Expectations for code review at Google do not center around problem solving. Reviewing was introduced at Google to ensure code readability and maintainability. Today's developers also perceive this educational aspect, in addition to maintaining norms, tracking history, gatekeeping, and accident prevention. Defect finding is welcomed but not the only focus.*

As stated earlier, when coding interview transcripts we also tracked the review context in which a theme was mentioned. We found that the relative importance of these different themes depends on the relationship between the author and



**Figure 1:** Relationship diagram describing which themes of review expectations appeared primarily within a particular author/reviewer context.

reviewer (Figure 1). For example, maintaining norms came up between an engineer and those with a different seniority (project leads, expert readability reviewers or "new" team members) but less so with their peers or other teams, where instead gatekeeping and accident prevention are primary. The educational benefits are broadly valued and encompass several different relationships.

**Finding 2.** *Expectations about a specific code review at Google depend on the work relationship between the author and reviewers.*

## 5 RESULTS: PRACTICES

In our second research question, we describe the code review process and compare its quantitative aspects with the convergent practices found in previous work [33].

### 5.1 Describing The Review Process

The code review at Google is linked to two concepts: *ownership* and *readability*. We first introduce them, then we describe the flow of the review process, and we conclude with the distinct characteristics of the internal review tool, CRITIQUE, that serve in contrast with other review tools.

**Ownership.** The Google codebase is arranged in a tree structure, where each directory is explicitly *owned* by a set of people. Although any developer can propose a change to any part of the codebase, an owner of the directory in question (or a parent directory) must review and approve the change before it is committed; even directory owners are expected to have their code reviewed before committing.

**Readability.** Google defines a concept called *readability*, which was introduced very early on to ensure consistent code style and norms within the codebase. Developers can gain readability certification in a particular language. To apply for readability, a developer sends changes to a set of readability reviewers; once those reviewers are confident the developer understands the code style and best practices for a language, the developer is granted readability for that language. Every change must be either authored or reviewed by someone with a readability certification for the language(s) used.

**Code Review Flow.** The flow of a review is tightly coupled with CRITIQUE and works as follows:

1. *Creating:* Authors start modifying, adding, or deleting some code; once ready, they create a change.
2. *Previewing:* Authors then use CRITIQUE to view the diff of the change and view the results of automatic code analyzers (e.g. from Tricorder [36]). When they are ready, authors mail the change out to one or more reviewers.
3. *Commenting:* Reviewers view the diff in the web UI, drafting comments as they go. Program analysis results, if present, are also visible to reviewers. *Unresolved* comments represent action items for the change author to definitely address. *Resolved* comments include optional or informational comments that may not require any action by a change author.
4. *Addressing Feedback:* The author now addresses comments, either by updating the change or by replying to comments. When a change has been updated, the author uploads a new snapshot. The author and reviewers can look at diffs between any pairs of snapshots to see what changed.
5. *Approving:* Once all comments have been addressed, the reviewers approve the change and mark it ‘LGTM’ (Looks Good To Me). To eventually commit a change, a developer typically must have approval from at least one reviewer. Usually, only one reviewer is required to satisfy the aforementioned requirements of ownership and readability.

We attempt to quantify how “lightweight” reviews are (CP<sub>1</sub>). We measure how much back-and-forth there is in a review, by examining how many times a change author mails out a set of comments that resolve previously unresolved comments. We make the assumption that one iteration corresponds to one instance of the author resolving some comments; zero iterations means that the author can commit immediately. We find that over 80% of all changes involve at most one iteration of resolving comments.

**Suggesting Reviewers.** To identify the best person to review a change, CRITIQUE relies on a tool that analyzes the change and suggests possible reviewers. This tool identifies the smallest set of reviewers needed to fulfill the review requirements for all files in a change. Note that often only one reviewer is required since changes are often authored by someone with ownership and/or readability for the files in question. This tool prioritizes reviewers that have recently edited and/or reviewed the included files. New team members are explicitly added as reviewers since they have not yet built up reviewing/editing history. Unassigned reviewers can also make comments on (and can potentially approve) changes. Tool support for finding reviewers is typically only necessary for changes to files beyond those for a particular team. Within a team, developers know who to send changes to. For changes that could be sent to anyone on the team, many teams use a system that assigns reviews sent to the team email address to configured team members in a round-robin manner, taking into account review load and vacations.

**Code Analysis Results.** CRITIQUE shows code analysis results as comments along with human ones (although in different colors). Analyzers (or reviewers) can provide suggested

edits, which can be both proposed and applied to the change via CRITIQUE. To vet changes before they are committed, development at Google also includes pre-commit hooks: checks where a failure requires an explicit override by the developer to enable a commit. Pre-commit checks include things like basic automated style checking and running automated test suites related to a change. The results of all pre-commit checks are visible in the code review tool. Typically, pre-commit checks are automatically triggered. These checks are configurable such that teams can enforce project-specific invariants and automatically add email lists to changes to raise awareness and transparency. In addition to pre-commit results, CRITIQUE displays the result of a variety of automated code analyses through Tricorder [36] that may not block committing a change. Analysis results encompass simple style checks, more complicated compiler-based analysis passes and also project-specific checks. Currently, Tricorder includes 110 analyzers, 5 of which are plugin systems for hundreds of additional checks, in total analyzing more than 30 languages.

**Finding 3.** *The Google code review process is aligned with the convergent practice of being lightweight and flexible. In contrast to other studied systems, however, ownership and readability are explicit and play a key role. The review tool includes reviewer recommendation and code analysis results.*

## 5.2 Quantifying The Review Process

We replicate quantitative analysis that Rigby and Bird conducted that led to the discovery CP<sub>2-4</sub> so as to compare these practices to the traits that Google has converged to.

**Review frequency and speed.** Rigby and Bird found that fast-paced, iterative development also applies to modern code review: In their projects, developers work in remarkably consistent short intervals. To find this, they analyzed review frequency and speed.

At Google, in terms of frequency, we find that the median developer authors about 3 changes a week, and 80 percent of authors make fewer than 7 changes a week. Similarly, the median for changes reviewed by developers per week is 4, and 80 percent of reviewers review fewer than 10 changes a week. In terms of speed, we find that developers have to wait for initial feedback on their change a median time of under an hour for small changes and about 5 hours for very large changes. The overall (all code sizes) median latency for the entire review process is under 4 hours. This is significantly lower than the median time to approval reported by Rigby and Bird [33], which is 17.5 hours for AMD, 15.7 hours for Chrome OS and 14.7, 19.8, and 18.9 hours for the three Microsoft projects. Another study found the median time to approval at Microsoft to be 24 hours [14].

**Review size.** Rigby and Bird argued that quick review time could only be enabled by smaller changes to review and subsequently analyzed review sizes. At Google, over 35% of the changes under consideration modify only a single file and

about 90% modify fewer than 10 files. Over 10% of changes modify only a single line of code, and the median number of lines modified is 24. The median change size is significantly lower than reported by Rigby and Bird for companies such as AMD (44 lines), Lucent (263 lines), and Bing, Office and SQL Server at Microsoft (somewhere between those boundaries), but in line for change sizes in open source projects [33].

**Number of reviewers and comments.** The optimal number of reviewers has been controversial among researchers, even in the deeply investigated code inspections [37]. Rigby and Bird investigated whether the considered projects converged to a similar number of involved reviewers. They found this number to be two, remarkably regardless of whether the reviewers were explicitly invited (such as in Microsoft projects, who invited a median of up to 4 reviewers) or whether the change was openly broadcasted for review [33].

At Google, by contrast, fewer than 25% of changes have more than one reviewer,<sup>1</sup> and over 99% have at most five reviewers with a median reviewer count of 1. Larger changes tend to have more reviewers on average. However, even very large changes on average require fewer than two reviewers. Rigby and Bird also found a “minimal increase in the number of comments about the change when more [than 2] reviewers were active” [33], and concluded that two reviewers find an optimal number of defects. At Google, the situation is different: A greater number of reviewers results in a greater average number of comments on a change. Moreover, the average number of comments per change grows with the number of lines changed, reaching a peak of 12.5 comments per change for changes of about 1250 lines. Changes larger than this often contain auto-generated code or large deletions, resulting in a lower average number of comments.

**Finding 4.** *Code review at Google has converged to a process with markedly quicker reviews and smaller changes, compared to the other projects previously investigated. Moreover, one reviewer is often deemed as sufficient, compared to two in the other projects.*

## 6 RESULTS: DEVELOPERS' PERCEPTIONS

Our last research question investigates challenges and satisfaction of developers with code review at Google.

### 6.1 Code Review Breakdowns at Google

Previous studies investigated challenges in the review process overall [4, 26] and provided compelling evidence, also corroborated by our experience as engineers, that understanding code to review is a major hurdle. To broaden our empirical body of knowledge, we focus here on challenges encountered in specific reviews (“breakdowns”), such as delays or disagreements.

<sup>1</sup>30% of changes have comments by more than one commenter, meaning that about 5% of changes have additional comments from someone that did not act as an approver for the change.

Five main themes emerged from the analysis of our interview data. The first four themes regard breakdowns concerning aspects of the *process*:

**Distance:** Interviewees perceive distance in code review from two perspectives: geographical (*i.e.*, the physical distance between the author and reviewers) and organizational (*e.g.*, between different teams or different roles). Both these types of distance are perceived as the cause of delays in the review process or as leading to misunderstandings.

**Social interactions:** Interviewees perceive the communication within code review as possibly leading to problems from two aspects: *tone* and *power*. Tone refers to the fact that sometimes authors are sensitive to review comments; sentiment analysis on comments has provided evidence that comments with negative tone are less likely to be useful [11]. Power refers to using the code review process to induce another person to change their behavior; for example, dragging out reviews or withholding approvals. Issues with tone or power in reviews can make developers uncomfortable or frustrated with the review process.

**Review subject:** The interviews referenced disagreements as to whether code review was the most suitable context for reviewing certain aspects, particularly *design reviews*. This resulted in mismatched expectations (*e.g.*, some teams wanted most of the design to be complete before the first review, others wanted to discuss design in the review), which could cause friction among participants and within the process.

**Context:** Interviewees allowed us to see that misunderstandings can arise based on not knowing what gave rise to the change; for example, if the rationale for a change was an urgent fix to a production problem or a “nice to have” improvement. The resulting mismatch in expectations can result in delays or frustration.

The last theme regarded the *tool* itself:

**Customization:** Some teams have different requirements for code review, *e.g.*, concerning how many reviewers are required. This is a technical breakdown, since arbitrary customizations are not always supported in CRITIQUE, and may cause misunderstandings around these policies. Based on feedback, CRITIQUE recently released a new feature that allows change authors to require that all reviewers sign off.

### 6.2 Satisfaction and Time Invested

To understand the significance of the identified concerns, we used part of the survey to investigate whether code review is overall perceived as valuable.

We find (Table 2) that code review is broadly seen as being valuable and efficient within Google – all respondents agreed with the statement that code review is valuable. This sentiment is echoed by internal satisfaction surveys we conducted on CRITIQUE: 97% of developers are satisfied with it.

Within the context of a specific change, the sentiment is more varied. The least satisfied replies were associated with changes that were very small (1 word or 2 lines) or with changes which were necessary to achieve some other goal, *e.g.*, triggering a process from a change in the source code.

	Strongly disagree	4	14	11	13	Strongly agree
For this change, the review process was a good use of my time	2	4	14	11	13	
Overall, I find code review at Google valuable	0	0	0	14	30	
For this change, the amount of feedback was	Too little	2	2	34	5	0
	Too much					

**Table 2:** User satisfaction survey results

However, the majority of respondents felt that the amount of feedback for their change was appropriate. 8 respondents described the comments as not being helpful, of these 3 provided more detail, stating that the changes under review were small configuration changes for which code review had less impact. Only 2 respondents said the comments had found a bug.

To contextualize the answers in terms of satisfaction, we also investigate the time developers spend reviewing code. To accurately quantify the reviewer time spent, we tracked developer interactions with CRITIQUE (*e.g.*, opened a tab, viewed a diff, commented, approved a change) as well as other tools to estimate how long developers spend reviewing code per week. We group sequences of developer interactions into blocks of time, considering a “review session” to be a sequence of interactions related to the same uncommitted change, by a developer other than the change author, with no more than 10 minutes between each successive interaction. We sum up the total number of hours spent across all review sessions in the five weeks that started in October 2016 and then calculate the average per user per week, filtering out users for whom we do not have data for all 5 weeks. We find that developers spend an average of 3.2 (median 2.6 hours a week) reviewing changes. This is low compared to the 6.4 hours/week of self-reported time for OSS projects [10].

**Finding 5.** *Despite years of refinement, code review at Google still faces breakdowns. These are mostly linked to the complexity of the interactions that occur around the reviews. Yet, code review is strongly considered a valuable process by developers, who spend around 3 hours a week reviewing.*

## 7 DISCUSSION

We discuss the themes that have emerged from this investigation, which can inspire practitioners in the setting up of their code review process and researchers in future investigations.

### 7.1 A Truly Lightweight Process

Modern code review was born as a lighter weight alternative to the cumbersome code inspections [4]; indeed Rigby and Bird confirmed this trait (CP<sub>1</sub>) in their investigation across systems [33]. At Google, code review has converged to an even more lightweight process, which developers find both valuable and a good use of their time.

Median review times at Google are much shorter than

in other projects [14, 33, 34]. We postulate that these differences are due to the culture of Google on code review (strict reviewing standards and expectations around quick turnaround times for review). Moreover, there is a significant difference with reviewer counts (one at Google vs. two in most other projects); we posit that having one reviewer helps make reviews fast and lightweight.

Both low review times and reviewer counts may result from code review being a required part of the developer workflow; they can also result from small changes. The median change size in OSS projects varies from 11 to 32 lines changed [34], depending on the project. At companies, this change size is typically larger [33], sometimes as high as 263 lines. We find that change size at Google more closely matches OSS: most changes are small. The size distribution of changes is an important factor in the quality of the code review process. Previous studies have found that the number of useful comments decreases [11, 14] and the review latency increases [8, 24] as the size of the change increases. Size also influences developers’ perception of the code review process; a survey of Mozilla contributors found that developers feel that size-related factors have the greatest effect on review latency [26]. A correlation between change size and review quality is acknowledged by Google and developers are strongly encouraged to make small, incremental changes (with the exception of large deletions and automated refactoring). These findings and our study support the value of reviewing small changes and the need for research and tools to help developers create such small, self-contained code changes for review [6, 15].

### 7.2 Software Engineering Research in Practice

Some of the practices that are part of code review at Google are aligned with practices proposed in software engineering research. For example, a study of code ownership at Microsoft found that changes made by minor contributors should be received with more scrutiny to improve code quality [9]; we found that this concept is enforced at Google through the requirement of an approval from an owner. Also, previous research has shown that typically one reviewer for a change will take on the task of checking whether code matches conventions [4]; readability makes this process more explicit. In the following, we focus on the features of CRITIQUE that make it a ‘next generation code review tool’ [26].

**Reviewer recommendation.** Researchers found that reviewers with prior knowledge of the code under review give more useful comments [4, 11], thus tools can add support for reviewer selection [11, 26]. We have seen that reviewer recommendation is tool-supported, prioritizing those who recently

edited/reviewed the files under review. This corroborates recent research that frequent reviewers make large contributions to the evolution of modules and should be included along with frequent editors [40]. Detecting the right reviewer does not seem problematic in practice at Google, in fact the model of recommendation implemented is straightforward since it can programmatically identify owners. This is in contrast with other proposed tools that identify reviewers who have reviewed files with similar names [43] or take into account such features as the number of comments included in reviews [42]. At Google, there is also a focus on dealing with reviewers' workload and temporary absences (in line with a study at Microsoft [4]).

**Static Analysis integration.** A qualitative study of 88 Mozilla developers [26] found that static analysis integration was the most commonly-requested feature for code review. Automated analyses allow reviewers to focus on the understandability and maintainability of changes, instead of getting distracted by trivial comments (*e.g.*, about formatting). Our investigation at Google showed us the practical implications of having static analysis integration in a code review tool. CRITIQUE integrates feedback channels for analysis writers [36]: Reviewers have the option to click “Please fix” on an analysis-generated comment as a signal that the author should fix the issue, and either authors or reviewers can click “Not useful” in order to flag an analysis result that is not helpful in the review process. Analyzers with high “Not useful” click rates are fixed or disabled. We found that this feedback loop is critical for maintaining developer trust in the analysis results.

**A Review Tool Beyond Collaborative Review.** Finally, we found strong evidence that CRITIQUE’s uses extend beyond reviewing code. Change authors use CRITIQUE to examine diffs and browse analysis tool results. In some cases, code review is part of the development process of a change: a reviewer may send out an unfinished change in order to decide how to finish the implementation. Moreover, developers also use CRITIQUE to examine the history of submitted changes long after those changes have been approved; this is aligned with what Sutherland and Venolia envisioned as a beneficial use of code review data for development [38]. Future work can investigate these unexpected and potentially impactful non-review uses of code review tools.

### 7.3 Knowledge spreading

Knowledge transfer is a theme that emerged in the work by Rigby and Bird [33]. In an attempt to measure knowledge transfer due to code review, they built off of prior work that measured expertise in terms of the number of files changed [28], by measuring the number of distinct files changed, reviewed, and the union of those two sets. They find that developers know about more files due to code review.

At Google, knowledge transfer is part of the educational motivation for code review. We attempted to quantify this effect by looking at comments and files edited/reviewed. As developers build experience working at Google, the average number of comments on their changes decreases (Figure 2).

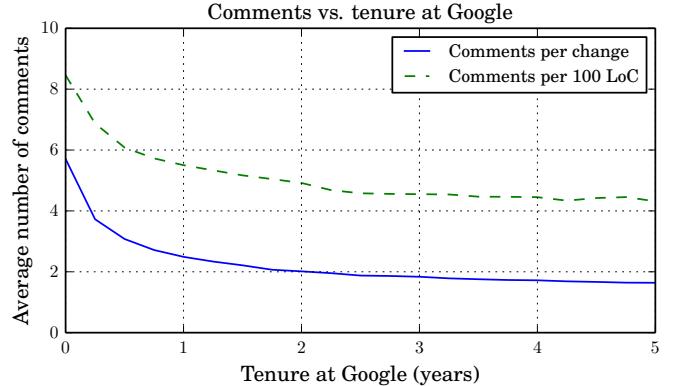


Figure 2: Reviewer comments vs. author’s tenure at Google

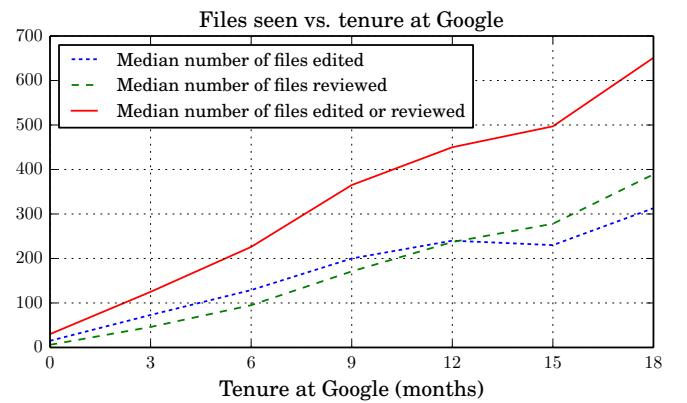


Figure 3: The number of distinct files seen (edited or reviewed, or both) by a full-time employee over time.

Developers at Google who have started within the past year typically have more than twice as many comments per change. Prior work found that authors considered comments with questions from the reviewer as not useful, and the number of not useful comments decreases with experience [11]. We postulate that this decrease in commenting is a result of reviewers needing to ask fewer questions as they build familiarity with the codebase and corroborates the hypothesis that the educational aspect of code review may pay off over time. Also, we can see that the number of distinct files edited and reviewed by engineers at Google, and the union of those two sets, increase with seniority (Figure 3) and the total number of files seen is clearly larger than the number of files edited. For this graph, we bucket our developers by how long they have been at the company (in 3-month increments) and then compute the number of files they have edited and reviewed. It would be interesting in future work to better understand how reviewing files impacts developer fluency [44].

## 8 CONCLUSION

Our study found code review is an important aspect of the development workflow at Google. Developers in all roles see it as providing multiple benefits and a context where developers can teach each other about the codebase, maintain

the integrity of their teams' codebases, and build, establish, and evolve norms that ensure readability and consistency of the codebase. Developers reported they were happy with the requirement to review code. The majority of changes are small, have one reviewer and no comments other than the authorization to commit. During the week, 70% of changes are committed less than 24 hours after they are mailed out for an initial review. These characteristics make code review at Google lighter weight than the other projects adopting a similar process. Moreover, we found that Google includes several research ideas in its practice, making the practical implications of current research trends visible.

## ACKNOWLEDGMENTS

The authors would like to thank Elizabeth Kemp for her assistance with the qualitative study, Craig Silverstein for context, and past and present members of the CRITIQUE team for feedback, including in particular Ben Rohlfs, Ilham Kurnia, and Kate Fitzpatrick. A. Bacchelli gratefully acknowledges the support of the Swiss National Science Foundation through the SNF Project No. PP00P2\_170529.

## REFERENCES

- [1] 2017. Appendix to this paper. <https://goo.gl/dP4ekg>. (2017).
- [2] A.F. Ackerman, L.S. Buchwald, and F.H. Lewski. 1989. Software inspections: An effective verification process. *IEEE Software* 6, 3 (1989), 31–36.
- [3] A.F. Ackerman, P.J. Fowler, and R.G. Ebenau. 1984. Software inspections and the industrial production of software. In *Symposium on Software validation: inspection-testing-verification-alternatives*.
- [4] Alberto Bacchelli and Christian Bird. 2013. Expectations, outcomes, and challenges of modern code review. In *ICSE*.
- [5] Vipin Balachandran. 2013. Reducing Human Effort and Improving Quality in Peer Code Reviews Using Automatic Static Analysis and Reviewer Recommendation. In *ICSE*.
- [6] Mike Barnett, Christian Bird, João Brunet, and Shuvendu K Lahiri. 2015. Helping developers help themselves: Automatic decomposition of code review changesets. In *ICSE*.
- [7] V.R. Basili, F. Shull, and F. Lanubile. 1999. Building knowledge through families of experiments. *IEEE Trans. on Software Eng.* 25, 4 (1999), 456–473.
- [8] Olga Baysal, Oleksii Kononenko, Reid Holmes, and Michael W Godfrey. 2013. The influence of non-technical factors on code review. In *WCER*.
- [9] Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. 2011. Don't touch my code!: examining the effects of ownership on software quality. In *FSE*.
- [10] Amiangshu Bosu and Jeffrey C Carver. 2013. Impact of peer code review on peer impression formation: A survey. In *ESEM*.
- [11] Amiangshu Bosu, Michaela Greiler, and Christian Bird. 2015. Characteristics of useful code reviews: an empirical study at Microsoft. In *MSR*.
- [12] chromium 2016. Chromium developer guidelines. <https://www.chromium.org/developers/contributing-code>. (August 2016).
- [13] J.W. Creswell. 2009. *Research design: Qualitative, quantitative, and mixed methods approaches* (3rd ed.). Sage Publications.
- [14] Jacek Czerwonka, Michaela Greiler, and Jack Tilford. 2015. Code Reviews Do Not Find Bugs: How the Current Code Review Best Practice Slows Us Down. In *ICSE (SEIP)*.
- [15] Martín Dias, Alberto Bacchelli, Georgios Gousios, Damien Cassou, and Stéphane Ducasse. 2015. Untangling fine-grained code changes. In *SANER*.
- [16] M.E. Fagan. 1976. Design and code inspections to reduce errors in program development. *IBM Systems Journal* 15, 3 (1976), 182–211.
- [17] gerrit 2016. <https://www.gerritcodereview.com/>. (August 2016).
- [18] githubpull 2016. GitHub pull request process. <https://help.github.com/articles/using-pull-requests/>. (2016).
- [19] Barney G Glaser and Anselm L Strauss. 2009. *The discovery of grounded theory: Strategies for qualitative research*. Transaction Books.
- [20] Georgios Gousios, Margaret-Anne Storey, and Alberto Bacchelli. 2016. Work Practices and Challenges in Pull-Based Development: The Contributor's Perspective. In *ICSE*.
- [21] Georgios Gousios, Andy Zaidman, Margaret-Anne Storey, and Arie van Deursen. 2015. Work Practices and Challenges in Pull-Based Development: The Integrator's Perspective. In *ICSE*.
- [22] Toshiki Hirao, Akinori Ihara, Yuki Ueda, Passakorn Phanachitta, and Ken-ichi Matsumoto. 2016. The Impact of a Low Level of Agreement Among Reviewers in a Code Review Process. In *IFIP International Conference on Open Source Systems*.
- [23] Harrie Jansen. 2010. The logic of qualitative survey research and its position in the field of social research methods. In *Forum Qualitative Sozialforschung*, Vol. 11.
- [24] Yujuan Jiang, Bram Adams, and Daniel M German. 2013. Will my patch make it? and how fast?: Case study on the linux kernel. In *MSR*.
- [25] Sami Kollanus and Jussi Koskinen. 2009. Survey of software inspection research. *The Open Software Engineering Journal* 3, 1 (2009), 15–34.
- [26] Oleksii Kononenko, Olga Baysal, and Michael W Godfrey. 2016. Code review quality: How developers see it. In *ICSE*.
- [27] Matthew B Miles and A Michael Huberman. 1994. *Qualitative data analysis: An expanded sourcebook*. Sage.
- [28] Audris Mockus and James D Herbsleb. 2002. Expertise browser: a quantitative approach to identifying expertise. In *ICSE*.
- [29] phabricator 2016. Meet Phabricator, The Witty Code Review Tool Built Inside Facebook. <https://techcrunch.com/2011/08/07/oh-what-noble-scribe-hath-penned-these-words/>. (August 2016).
- [30] Rachel Potvin and Josh Levenburg. 2016. Why Google Stores Billions of Lines of Code in a Single Repository. *Commun. ACM* (2016).
- [31] Teade Punter, Marcus Ciolkowski, Bernd Freimut, and Isabel John. 2003. Conducting on-line surveys in software engineering. In *Empirical Software Engineering*.
- [32] P. Rigby, B. Cleary, F. Painchaud, M.A. Storey, and D. German. 2012. Open Source Peer Review—Lessons and Recommendations for Closed Source. *IEEE Software* (2012).
- [33] Peter C Rigby and Christian Bird. 2013. Convergent software peer review practices. In *FSE*.
- [34] Peter C. Rigby, Daniel M. German, Laura Cowen, and Margaret-Anne Storey. 2014. Peer Review on Open-Source Software Projects: Parameters, Statistical Models, and Theory. *IEEE Transactions on Software Engineering* 23, 4, Article 35 (2014).
- [35] David L Sackett. 1979. Bias in analytic research. *Journal of chronic diseases* 32, 1-2 (1979), 51–63.
- [36] Caitlin Sadowski, Jeffrey van Gogh, Ciera Jaspan, Emma Söderberg, and Collin Winter. 2015. Tricorder: building a program analysis ecosystem. In *ICSE*.
- [37] F. Shull and C. Seaman. 2008. Inspecting the History of Inspections: An Example of Evidence-Based Technology Diffusion. *IEEE Software* 25, 1 (2008), 88–90.
- [38] Andrew Sutherland and Gina Venolia. 2009. Can peer code reviews be exploited for later information needs?. In *ICSE*.
- [39] William Isaac Thomas and Dorothy Swaine Thomas. 1928. The methodology of behavior study. *The child in America: Behavior problems and programs* (1928), 553–576.
- [40] Patanamon Thongtanunam, Shane McIntosh, Ahmed E Hassan, and Hajimu Iida. 2016. Revisiting code ownership and its relationship with software quality in the scope of modern code review. In *ICSE*.
- [41] Patanamon Thongtanunam, Shane McIntosh, Ahmed E Hassan, and Hajimu Iida. 2017. Review participation in modern code review. *Empirical Software Engineering* 22, 2 (2017), 768–817.
- [42] Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Raula Gaikovina Kula, Norihiro Yoshida, Hajimu Iida, and Ken-ichi Matsumoto. 2015. Who should review my code? A file location-based code-reviewer recommendation approach for modern code review. In *SANER*.
- [43] Motahareh Bahrami Zanjani, Huzefa Kagdi, and Christian Bird. 2016. Automatically recommending peer reviewers in modern code review. *IEEE Transactions on Software Engineering* 42, 6 (2016), 530–543.
- [44] Minghui Zhou and Audris Mockus. 2010. Developer Fluency: Achieving True Mastery in Software Projects. In *FSE*.

# Expectations, Outcomes, and Challenges Of Modern Code Review

Alberto Bacchelli

REVEAL @ Faculty of Informatics  
University of Lugano, Switzerland  
alberto.bacchelli@usi.ch

Christian Bird

Microsoft Research  
Redmond, Washington, USA  
cbird@microsoft.com

**Abstract**—Code review is a common software engineering practice employed both in open source and industrial contexts. Review today is less formal and more "lightweight" than the code inspections performed and studied in the 70s and 80s. We empirically explore the motivations, challenges, and outcomes of tool-based code reviews. We observed, interviewed, and surveyed developers and managers and manually classified hundreds of review comments across diverse teams at Microsoft. Our study reveals that while finding defects remains the main motivation for review, reviews are less about defects than expected and instead provide additional benefits such as knowledge transfer, increased team awareness, and creation of alternative solutions to problems. Moreover, we find that code and change understanding is the key aspect of code reviewing and that developers employ a wide range of mechanisms to meet their understanding needs, most of which are not met by current tools. We provide recommendations for practitioners and researchers.

## I. INTRODUCTION

Peer code review, a manual inspection of source code by developers other than the author, is recognized as a valuable tool for reducing software defects and improving the quality of software projects [1] [2]. In 1976, Fagan formalized a highly structured process for code reviewing [3], based on line-by-line group reviews, done in extended meetings—*code inspections*. Over the years, researchers provided evidence on code inspection's benefits, especially in terms of defect finding, but the cumbersome, time-consuming, and synchronous nature of this approach hinders its universal adoption in practice [4].

Nowadays, many organizations are adopting more lightweight code review practices to limit the inefficiencies of inspections. In particular, there is a clear trend toward the usage of tools specifically developed to support code review [5]. In the context of this paper, we define *Modern Code Review*, as review that is (1) informal (in contrast to Fagan-style), (2) tool-based, and that (3) occurs regularly in practice nowadays, for example at companies such as Microsoft, Google [6], Facebook [7], and in other companies and OSS projects [8].

This trend raises questions, such as: Can we apply the lessons learned from previous research on code inspections to modern code reviews? What are the expectations for code review nowadays? What are the actual outcomes of code review? What challenges do people face in code review?

Answers to these questions can provide insight for both practitioners and researchers. Developers and other software project stakeholders can use empirical evidence about expectations and outcomes to make informed decisions about

when to use code review and how it should fit into their development process. Researchers can focus their attention on practitioners' challenges to make code review more effective.

We present an in-depth study of practices in teams that use modern code review, revealing what practitioners think, do, and achieve when it comes to modern code review.

Since Microsoft is made up of many different teams working on very diverse products, it gives the opportunity to study teams performing code review *in situ* and understand their expectations, the benefits they derive from code review, the needs they have, and the problems they face.

We set up our study as an explorative investigation. We started without *a priori* hypotheses regarding how and why code review should be performed, with the aim of discovering what developers and managers expect from code review, how reviews are conducted in practice, and what the actual outcomes and challenges are. To that end, we (1) observed 17 industrial developers performing code review with various degrees of experience and seniority across 16 separate product teams with distinct reviewing cultures and policies; (2) interviewed these developers using a semi-structured interviews; (3) manually inspected and classified the content of 570 comments in discussions contained within code reviews; and (4) surveyed 165 managers and 873 programmers.

Our results show that, although the top motivation driving code reviews is still finding defects, the practice and the actual outcomes are less about finding errors than expected: Defect related comments comprise a small proportion and mainly cover small logical low-level issues. At the same time, code review additionally provides a wide spectrum of benefits to software teams, such as knowledge transfer, team awareness, and improved solutions to problems. Moreover, we found that context and change understanding is the key of any review. According to the outcomes they want to achieve, developers employ many mechanisms to fulfill their understanding needs, most of which are not currently met by any code review tool.

This paper makes the following contributions:

- Characterizing the motivations of developers and managers for code review and compare with actual outcomes.
- Relating the outcomes to understanding needs and discuss how developers achieve such needs.

Based on our findings, we provide recommendations for practitioners and implications for researchers as well as outline future avenues for research.

## II. RELATED WORK

Previous studies exist that have examined the practices of code inspection and code review. Stein et al. conducted a study focusing specifically on distributed, asynchronous code inspections [17]. The study included evaluation of a tool that allowed for identification and sharing of code faults or defects. Participants at separated locations can then discuss faults via the tool. Laitenburger conducted a survey of code inspection methods, and presented a taxonomy of code inspection techniques [9]. Johnson conducted an investigation into code review practices in open source development and the effect they have on choices made by software project managers [10].

Porter et al. [11] reported on a review of studies on code inspection in 1995 that examined the effects of factors such as team size, type of review, and number of sessions on code inspections. They also assessed costs and benefits across a number of studies. These studies differ from ours in that they were not tool-based and were the majority involved planned meetings to discuss the code.

However, prior research also sheds light on why review today is more often tool-based, informal, and often asynchronous. The current state of code review might be due to the time required for more formal inspections. Votta found that 20% of the interval in a “traditional inspection” is wasted due to scheduling [12]. The ICICLE tool [13], or “Intelligent Code Inspection in a C Language Environment,” was developed after researchers at Bellcore observed how much time and work was expended before and during formal code inspections. Many of today’s review tools are based on ideas that originated in ICICLE. Other similar tools have been developed in an effort to reduce time for inspection and allow asynchronous work on reviews. Examples include CAIS [14] and Scrutiny [15].

More recently, Rigby has done extensive work examining code review practices in open source software development [5]. For example in a study of practices in the Apache project [16] they data-mined the email archives and found that reviews were typically small and frequent, and that the contributions to a review were often brief and independent from one another.

Sutherland and Venolia conducted a study at Microsoft regarding using code review data for later information needs [17]. They hypothesized that the knowledge exchanged during code reviews could be of great value to engineers later trying to understand or modify the discussed code. They found that “the meat of the code review dialog, no matter what medium, is the articulation of design rationale” and, thus, “code reviews are an enticing opportunity for capturing design rationale.”

When studying developer work habits, Latoza *et al.* found that many problems encountered by developers were related to understanding the rationale behind code changes and gathering knowledge from other members of their team [18].

## III. METHODOLOGY

In this section we define the research questions, describe the research settings, and outline our research method.

### A. Research Questions

Our investigation of code review revolves around the following research questions, which we iteratively refined during our initial in-field observations and interviews:

1. What are the motivations and expectations for modern code review? Do they change from managers to developers and testers?
2. What are the actual outcomes of modern code review? Do they match the expectations?
3. What are the main challenges experienced when performing modern code reviews relative to the expectations and outcomes?

### B. Research Setting

Our study took place with professional developers, testers, and managers. Microsoft develops software in diverse domains, from high end server enterprise data management solutions such as SQL Server to mobile phone applications and smart phone apps to search engines. Each team has its own development culture and code review policies. Over the past two years, a common tool for code review at Microsoft has achieved wide-spread adoption. As it represents a common and growing solution for code review (over 40,000 developers used it so far), we focused on developers using this tool for code review—*CodeFlow*.

CodeFlow is a collaborative code review tool that allows users to directly annotate source code in its viewer and interact with review participants in a live chat model. The functionality of CodeFlow is similar to other review tools such Google’s Mondrian [6], Facebook’s Phabricator [7] or open-source Gerrit [8]. Developers who want their code to be reviewed create a package with the changed (new, deleted, and modified) files, select the reviewers, write a message to describe the code review, and submit everything to the CodeFlow service. CodeFlow then notifies the reviewers about the incoming task via email.

Once reviewers open a CodeFlow review, they interact with it via a single desktop window (Figure 1). On the top left (1), they see the list of files changed in the current submission, plus a “description.txt” file, which contains a textual explanation of the change, written by the author. On bottom left, CodeFlow shows the list of reviewers and their status (2). We see that Christian is the review author and Alberto, Tom, and Nachi are the reviewers. Alberto has reviewed and is waiting for the author to act, as the clock icon suggests, while Nachi already signed off on the changes. CodeFlow’s main view (3) shows the diff-highlighted content of the file currently under review. Both the reviewers and the author can highlight portions of the code and add comments inline (4). These comments can start threads of discussion and are the interaction points for the people involved in the review. Each user viewing the same review in CodeFlow sees events as they happen. Thus, if an author and reviewer are working on the review at the same time, the communication is synchronous and comment threads act similar to instant messaging. The comments are persisted so that if they work at different times, the communication



Figure 1. Screenshot of CodeFlow, the dominant code review tool used by developers at Microsoft.

becomes asynchronous. The bottom right pane (5) shows the summary of all the comments in the review.

CodeFlow centralizes and records all the information on code reviews on a central server. This provides an additional data source that we used to analyze real code review comments without incurring the Hawthorne effect [19].

### C. Research Method

Our research method followed a mixed approach [20], depicted in Figure 2, collecting data from different sources for triangulation: (1) analysis of previous study, (2) observations and interviews with developers, (3) card sort on interview data, (4) card sort on code review comments, (5) the creation of an affinity diagram, and (6) survey to managers and programmers.

**1. Analysis of previous study:** Our research started with the analysis of a study commissioned by Microsoft, between April and May 2012 carried out by an external vendor. The study investigated how different product teams were using CodeFlow. It consisted of structured interviews (lasting 30-50 minutes) to 23 people with different roles.

Most of the interview questions revolved around topics that are very specific to tool usage, and were only tangentially related to this work. We found one relevant as a starting point for our study: “*What do you hope to accomplish when you submit a code review?*” We analyzed the transcript of this answer, for each interview, through the process of *coding* [21] (also used in *grounded theory* [22]): breaking up the answers into smaller coherent units (sentences or paragraphs) and adding *codes* to them. We organized codes into *concepts*, which in turn were grouped into more abstract *categories*.

From this analysis, four motivations emerged for code review: finding defects, maintaining team awareness, improving code quality, and assessing the high-level design. We used them to draw an initial guideline for our interviews.

**2. Observations and interviews with developers:** Next, we conducted a series of one-to-one meetings with developers who use CodeFlow, each taking 40-60 minutes.

We contacted 100 random candidates who signed-off between 50 and 250 code reviews since the CodeFlow release and sampled across different product teams to address our research questions from a *multi-point* perspective. We wrote developers who used CodeFlow in the past and asked them to contact us, giving us 30 minute notice when they received their next review task so that we could observe. The respondents that we interviewed comprised five developers, four senior developers, six testers, one senior tester, and one software architect. Their time in the company ranged from 18 months to almost 10 years, with a median of five years.

Each meeting was comprised of two parts: In the first part, we observed them performing the code review that they had been assigned. To minimize invasiveness we used only one observer and to encourage the participant to narrate their work, we asked the participants to think of us as a newcomer to the team. In this way, most developers thought aloud without need of prompting. With consent, we recorded the audio, assuring the participants of anonymity. Since we, as observers, have backgrounds in software development and practices at Microsoft, we were able to understand most of the work and where and how information was obtained without inquiry.

The second part of the meeting was a *semi-structured interview* [23]. Semi-structured interviews make use of an *interview guide* that contains general groupings of topics and questions rather than a pre-determined exact set and order of questions. They are often used in an exploratory context to “find out what is happening [and] to seek new insights” [24]. The guideline was iteratively refined after each interview, in particular when developers started providing answers very similar to the earlier ones, thus reaching a *saturation* effect.

Observations also reached a saturation point, thus providing insights very similar to the earlier ones. For this, after the first 5-6 observations, we adjusted the meetings to have shorter observations, which we used as a starting point for our meetings and as a “hook” to talk about topics in our guideline.

The audio of each interview was then transcribed and broken up into smaller coherent units for subsequent analysis.

**3. Card sort (meetings):** To group codes that emerged from interviews and observations into categories, we conducted a *card sort*. Card sorting is a sorting technique that is widely used in information architecture to create mental models and derive taxonomies from input data [25]. In our case it helped to organize the codes into hierarchies to deduce a higher level of abstraction and identify common themes. A card sort involves three phases: In the (1) *preparation phase*, participants of the card sort are selected and the cards are created; in the (2) *execution phase*, cards are sorted into meaningful groups with a descriptive title; and in the (3) *analysis phase*, abstract hierarchies are formed to deduce general categories.

We applied an *open card sort*: There were no predefined groups. Instead, the groups emerged and evolved during the sorting process. In contrast, a closed card sort has predefined groups and is typically applied when themes are known in advance, which was not the case for our study.

The first author of this paper created all of the cards, from the 1,047 coherent units in the interviews. Throughout our

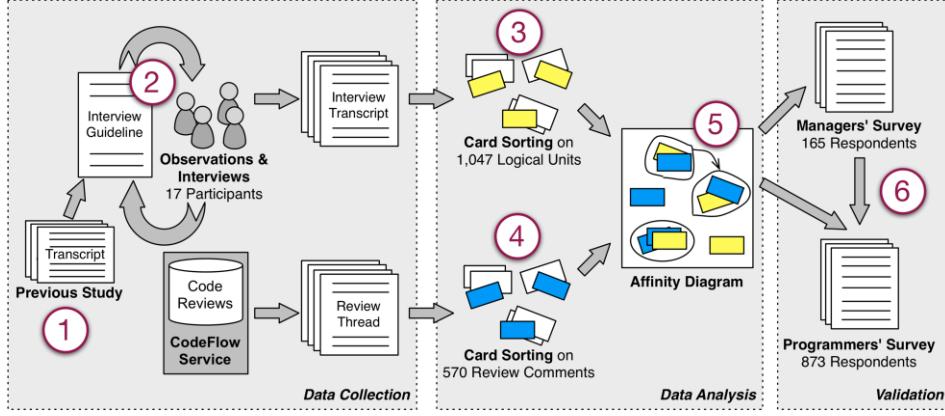


Figure 2. The mixed approach research method applied.

further analysis other researchers (the second author and external people) were involved in developing categories and assigning cards to categories, so as to strengthen the validity of the result. The first author played a special role of ensuring that the context of each question was appropriately considered in the categorization, and creating the initial categories. To ensure the integrity of our categories, the cards were sorted by the first author several times to identify initial themes. Next, all researchers reviewed and agreed on the final set of categories.

**4. Card sort (code review comments):** The same method was applied to group code review comments into categories: We randomly sampled 200 threads with at least two comments (e.g., Point 4 of Figure 2), from the entire dataset of CodeFlow reviews, which embeds data from dozens of independent software products at Microsoft. We printed one card for each comment (along with the entire discussion thread to give the context), totaling 570 cards, and conducted a card sort, as performed for the interviews, to identify common themes.

**5. Affinity Diagram:** We used an *affinity diagram* to organize the categories that emerged from the card sort. This tool allows large numbers of ideas to be sorted into groups for review and analysis [26]. We used it to generate an overview of the topics that emerged from the card sort, in order to connect the related concepts and derive the main themes. For generating the affinity diagram, we followed the five canonical steps: we (1) recorded the categories on post-it-notes, (2) spread them onto a wall, (3) sorted the categories based on discussions, until all are sorted and all participants agreed, (4) named each group with a description, and (5) captured and discussed the themes.

**6. Surveys:** The final step of our study was aimed at validating the concepts that emerged from the previous phases. Towards this goal, we created two surveys to reach a significant number of participants and to challenge our conclusions (The full surveys are available as a technical report [27]). For the design of the surveys, we followed Kitchenham and Pfleeger's guidelines for personal opinion surveys [28]. Both surveys were anonymous to increase response rates [29].

We sent the first survey to a cross section of managers. We considered managers for which at least half of their team performed code reviews regularly (on average, one per week or more) and sampled along two dimensions. The first dimension

was whether or not the manager had participated in a code review himself since the beginning of the year and the second dimension was whether the manager managed a single team or multiple teams (a manager of managers). Thus, we had one sample of first level managers who participated in review, another sample of second level managers who participated in reviews, etc. The first survey was a short survey comprising 6 questions (all optional), which we sent to 600 managers that had at least ten direct or indirect reporting developers who used CodeFlow in the past. The central focus was the open question asking to enumerate the main motivations for doing code reviews in their team. We received 165 answers (28% response rate), which we analyzed before devising the second survey.

The second survey comprised 18 questions, mostly closed with multiple choice answers, and was sent to 2,000 randomly chosen developers who signed off on average at least one code review per week since the beginning of the year. We used the time frame of January to June of 2012 to minimize the amount of organizational churn during the time period and identify employees' activity in their current role and team. We received 873 answers (44% response rate). Both response rates were high, as other online surveys in software engineering have reported response rates ranging from 14% to 20% [30].

#### IV. WHY DO PROGRAMMERS DO CODE REVIEWS?

Our first research question seeks to understand what motivations and expectations drive code reviews, and whether managers and developers share the same opinions.

Based on the responses that we coded from observations of developers performing code review as well as interviews, there are various motivations for code review. Overall, the interviews revealed that finding defects, even though prominent, is just one of the many motivations driving developers to perform code reviews. Especially when reinforced by a strong team culture around reviews, developers see code reviews as an activity that has multiple beneficial influences not only on the code, but also for the team and the entire development process. In this vein, one senior developer's comment summarized many of the responses: “[code review] also has several beneficial influences: (1) makes people less protective about their code, (2) gives another person insight into the code, so

*there is (3) better sharing of information across the team, (4) helps support coding conventions on the team, and [...] (5) helps improving the overall process and quality of code.”*

Through the card sort on both meetings and code review comments, we found several references to motivations for code review and identified six main topics. To complete this list, in the survey for managers, we included an open question on why they perform code reviews in their team. We analyzed the responses to create a comprehensive list of high-level motivations. We included this list in the developers' survey and asked them to rank the top three main reasons that described why they do code reviews.

In the rest of this section, we discuss the motivations that emerged as the most prominent. We order them according to the importance they were given by the 873 developers and testers who responded to the final survey.

#### A. Finding Defects

One interviewed senior tester explains that he performs code reviews because they “*are a great source of bugs;*” he goes even further stating: “*sometimes code reviews are a cheaper form of bug finding than testing.*” Moreover, the tool seems not to have an impact on this main motivation: “*using CodeFlow or using any other tool makes a little difference to us; it's more about being able to identify flaws in the logic.*”

Almost all the managers included “finding defects” as one of the reasons for doing code reviews; for 44% of the managers, it is the top reason. Managers considered defects to be both low level issues (e.g., “*correct logic is in place*”) and high level concerns (e.g., “*catch errors in design*”). Concerning surveyed developers/testers, “finding defects” is the first motivation for code review for 383 of the programmers (44%), second motivation for 204 (23%), and third for 96 (11%).

This is in-line with the reason why code inspections were devised in the first place: reducing software defects [1].

Nevertheless, even though “finding defects” emerged from our data as a strong motivation (the first for almost half of the programmers and managers), interviews and survey results indicate that this only tells part of the story of why practitioners do code reviews and the outcomes they expect.

#### B. Code Improvement

Code improvements are comments or changes about code in terms of readability, commenting, consistency, dead code removal, etc., but do not involve correctness or defects.

Programmers ranked “code improvement” as an important motivation for code review, close to “finding defects.” This is the primary motivation for 337 programmers (39%), the second for 208 (24%), and the third for 135 (15%). Managers reported code improvements as their primary motivation in 51 cases (31%). One manager wrote how code review in her view is a “*discipline of explaining your code to your peers [that] drives a higher standard of coding. I think the process is even more important than the result.*”

Most interviewed programmers mentioned that at least one of the reviewers involved in each code review takes care of checking whether the code follows the team conventions, for example in terms of code formatting and in terms of function

and variable naming. Some programmers use the “code improvement” check as a first step when doing code review: “*the first basic pass on the code is to check whether it is standard across the team.*”

The interviews also gave us a glimpse of the connection between the quality of code reviews and “code improvement” comments. Such comments seem easier to write and sometimes interviewees mentioned them as the way reviewers use to avoid spending time to conduct good code reviews. An observation by a senior developer, in the company for more than nine years, summarizes the opinions we received from many interviewees: “*I've seen quite a few code reviews where someone commented on formatting while missing the fact that there were security issues or data model issues.*”

#### C. Alternative Solutions

“Alternative solutions” regard changes and comments on improving the submitted code by adopting an idea that leads to a better implementation. This is one of the few motivations in which developers and managers do not agree. While 147 (17%) developers put this as the first motivation, 202 (23%) as the second, and 152 (17%) as the third, only 4 managers (2%) even mentioned it (e.g., “*Generate better ideas, alternative approaches*” and “*Collective wisdom: Someone else on the project may have a better idea to solve a problem*”). The outcome of the interviews was similar to the position of managers: Interviewees vaguely mentioned this motivation, and mostly in terms of generic “*better ways to do things.*”

#### D. Knowledge Transfer

All the interviewees but one motivated their code reviews also from a learning, or “knowledge transfer,” perspective. With the words of a senior developer: “*one of the things that should be happening with code reviews over time is a distribution of knowledge. If you do a code review and did not learn anything about the area and you still do not know anything about the area, then that was not as good code review as it could have been.*” Although we did not include questions related to “knowledge transfer” in our interview guideline, this topic kept emerging spontaneously from each meeting, thus underscoring its value for practitioners.

Sometimes programmers told us that they follow code reviews explicitly for learning purposes. For example, a tester explained: “[I read code reviews because] *from a code review you can learn about the different parts you have to touch to implement a certain feature.*”

According to interviewees, code review is a learning opportunity for both the author of the change and the reviewers: There is a bidirectional knowledge transfer about APIs usage, system design, best practices, team conventions, “*additional code tricks,*” etc. Moreover code reviews are recognized for educating new developers about code writing.

Managers included “knowledge transfer” as one of the reasons for code review, although never as the top motivation. They mostly wrote about code review as an education means by mentioning among the motivations for code review: “*developer education,*” “*education for junior developers who*

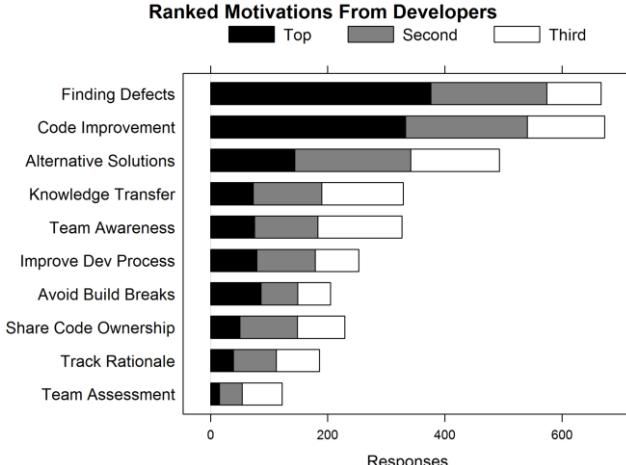


Figure 3. Developers' motivations for code review.

are learning the codebase,” and “learning tool to teach more junior team members.”

Programmers answering the survey declared “knowledge transfer” to be their first motivation for code review in 73 cases (8%), their second in 119 (14%), and their third in 141 (16%).

#### E. Team Awareness and Transparency

During one of our observations, one developer was preparing a code review submission as an author: He wanted other developers to “double check” his changes before committing them to the repository. After preparing the code, he specified the developers he wanted to review his code; he required not only two specific people, but he also put a generic email distribution group as an “optional” reviewer. When we inquired about this choice, he explained us: “*I am adding [this alias], so that everybody [in the team] is notified about the change I want to do before I check it in.*” In the subsequent interviews, this concept of using an email list as optional reviewer, or including specific optional reviewers exclusively for awareness emerged again frequently, e.g., “*Code reviews are good FYIs [for your information].*”

Managers often mentioned the concept of team awareness as a motivation for code review, frequently justifying it with the notion of “transparency.” Not only must the team be kept aware of the directions taken by the code, but also nobody should be allowed to “secretly” make changes that might break the code or alter functionalities.

The 873 programmers answering the survey ranked “team awareness and transparency” very close to “knowledge transfer.” In fact, the two concepts appeared logically related also in the interviews; for example one tester, while reviewing some code said: “*oh, this guy just implemented this feature, and now let me back and use it somewhere else.*” Showing that he both learned about the new feature and he was now aware of the possibility to use it in his own code. 75 (9%) developers considered team awareness their first motivation for code review, 108 (12%) their second, and 149 (17%) their third.

Although team awareness and transparency emerged from our data as clearly promoted by the code review process, academic research seems to have given little attention to it.

#### F. Share Code Ownership

The concept of “shared code ownership” is closely related to “team awareness and transparency,” but it has a stronger connotation toward active collaboration and overlapping coding activities. Programmers and managers believe that code review is not only an occasion to notify other team members about incoming changes, but also a means to have more than one knowledgeable person about specific parts of the codebase. A manager put the following as her second motivation for code review: “*Broaden knowledge & understanding of how specific features/areas are designed and implemented (e.g., grooming “backup developers” for areas where knowledge is too concentrated on one or two expert developers.*”

Moreover, both developers and managers have the opinion that practicing code review also improves the personal perception of team members about shared code ownership. On this note, a senior developer, with more than 30 years in the software industry, explained: “*In the past people did not use to do code reviews and were very reluctant to put themselves in positions where they were having other people critiquing their code. The fact that code reviews are considered as a normal thing helps immensely with making people less protective about their code.*” Similarly a manager wrote us explaining that she deems code reviews important because they “*Dilute any “rigid sense of ownership” that might develop over chunks of code.*”

In the programmers’ survey, 51 respondents (6%) marked “share code ownership” as their first motivation, 100 (11%) as their second, and (10%) as their third.

#### G. Summary

In this section, we analyzed the motivations that developers and managers have for doing code review. We abstracted them into a list, which we finally included in the programmers’ survey. Figure 3 reports the answers given to this question: The black bar is the number of developers that put that row as their top motivation, the gray bar is the number that put it as the second motivation, etc. We have ordered the factors by giving 3 points for a first motivation response, 2 points for a second motivation, etc. and then sorting by the sum.

We discussed the five most prominent motivations, which show that “finding defects” is the top motivation, although participants believe that code review brings other benefits. The first two motivations were already popular in research and their effectiveness have been evaluated in the context of code inspections; on the contrary, the other motivations are still unexplored, especially those regarding more “social” benefits on the team, such as shared code ownership.

Although motivations are well defined, we still have to verify whether they actually translate into real outcomes of a modern code review process.

## V. THE OUTCOMES OF CODE REVIEWS

Our second research question seeks to understand what the actual outcomes of code reviews are, and whether they match the motivations and expectations outlined in the previous section. To that end, we conducted *indirect field research* [31] by analyzing the content of 200 threads (corresponding to 570

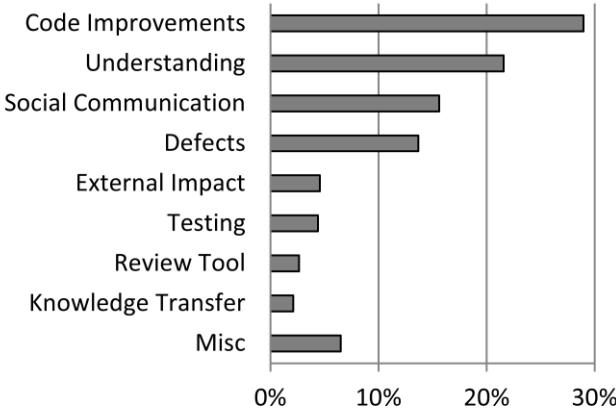


Figure 4. Frequency of comments by card sort category.

comments) recorded by CodeFlow. Figure 4 shows the categories of comments found through the card sort.

**Code Improvements:** The most frequent category, with 165 (29%) comments, is “code improvements.” In detail, among “code improvements” comments we find 58 on using better code practices, 55 on removing not necessary or unused code, and 52 on improving code readability.

**Defect Finding:** Although “defect finding” is the top motivation and expected outcome of code review for many practitioners, the category “defect” is the only the fourth most frequent, out of nine items, with 78 (14%) comments. Among “defect” comments, 65 are on logical issues (*e.g.*, a wrong expression in an *if* clause), 6 on high-level issues, 5 on security, and 3 on wrong exception handling.

**Knowledge Transfer:** Concerning the other expected outcomes of code reviews, we did not expect to find evidence about them, because of their more “social”—thus harder to quantify—nature. Nevertheless, we found some (12) comments specifically about “knowledge transfer,” where the reviewers were directing the code change author to external resources (*e.g.*, internal documentation or websites) for learning how to tackle some issues. This provides additional evidence on the importance of this aspect of reviews.

#### A. Finding defects: When expectations do not meet reality

Why do we see this significant gap in frequency between “code improvements” and “defects” comments? Possible reasons may be that our sample of 570 comments is too small to represent the population, that the submitted changes might require less need fixing of “real” defects than of small code improvements, or that programmers could consider “code improvements” as actual defects. However, by triangulating these numbers with the interview discussions, the survey answers, and the other categories of comments, another reason seems to justify this situation. First, we start by noting that most of the comments on “defects” regard *uncomplicated* logical errors, *e.g.*, corner cases, common configuration values, or operator precedence. Then, from interview data, we see that: (1) most interviewees explained how, with tool-based code reviews, most of the found defects regard “*logic issues*—where

*the author might not have considered a particular or corner case*; (2) some interviewees complained that the quality of code reviews is low, because reviewers only look for easy errors: “[*Some reviewers*] focus on formatting mistakes because they are easy [...], but it doesn’t really help. [...] In some ways it’s kind of embarrassing if someone asks you to do a code review and all you can find are formatting mistakes when there are real mistakes to be found”; and (3) other interviewees admitted that if the code is not among their codebase, they look at “*obvious bugs (such as, exception handling)*.” Finally, managers mentioned “*catching early obvious bugs*” or “*finding obvious inefficiencies or errors*” as reasons for doing code review.

These points illustrate that the reason for the gap between the number of comments on “code improvements” and on “defects” is not to be found in problems in the sample or in classification misconceptions, but it is rather just additional corroborating evidence that the outcome of code review does not match the main expectation of both programmers and managers—*finding defects*. Review comments about defects are few, comprising one-eighth of the total in our sample, and mostly address “micro” level and superficial concerns; while programmers and managers would expect more insightful remarks on conceptual and design level issues. Why does this happen? The high frequency of understanding comments hints at the answer to our question, addressed in the next section.

## VI. WHAT ARE THE CHALLENGES OF CODE REVIEW?

Our third research question seeks to understand the main challenges faced by the reviewers when performing modern code reviews, also with respect to the expected outcomes. We also seek to uncover the reason behind the mismatch between expectations and actual outcomes on finding defects in code reviews.

### A. Code Review is Understanding

Even though we did not ask any specific question concerning understanding, the theme emerged clearly from our interviews. Many interviewees eventually acknowledged that understanding is their main challenge when doing code reviews. For example, a senior developer autonomously explained to us: “*the most difficult thing when doing a code review is understanding the reason of the change;*” a tester, in the same vein: “*the biggest information need in code review: what instigated the change;*” and another senior developer: “*in a successful code review submission the author is sure that his peers understand and approve the change.*” Although the textual description should help reviewers understanding, some developers do not find it useful: “*people can say they are doing one thing, while they are doing many more of them;*” or “*the description is not enough;*” in general, developers seem to confirm that “*not knowing files (or [dealing with] new ones) is a major reason for not understanding a change.*”

From interviews, no other code review challenge emerged as clearly as understanding the submitted change. Even though scheduling and time issues also appeared challenging, we could always trace them back to the first challenge—through the words of a tester: “*understanding the code takes most of the*

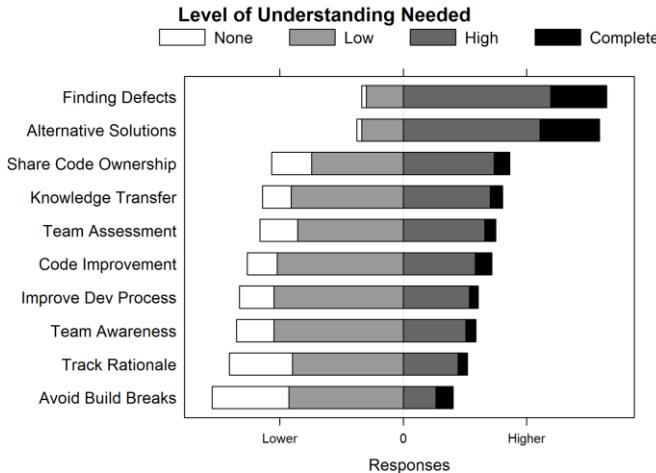


Figure 5. Developers' responses in surveys of the amount of code understanding for code review outcomes.

reviewing time.” On the same note, in the code review comments we analyzed, the second most frequent category concerns *understanding*. This category includes clarification questions and doubts raised by the reviewers who want to grasp the rationale of the changes done on the code, and the corresponding clarification answers. This is also in line with the evidence delivered by Sutherland & Venolia on the relevance of rationale articulation in reviews [17].

Do *understanding needs* change with the expected outcome of code review? We included a question in the programmers’ survey to know how much understanding they needed to achieve each of the motivations listed in Figure 3. The outcome of the question is summarized in Figure 5. The respondents could answer with a four values Likert’s scale, by selecting the understanding of the change they felt was required to achieve the specific outcome. The most difficult task from the understanding perspective is “finding defects,” immediately followed by “alternative solutions.” Both clearly stand out from the other items. The gap in understanding needs between “finding defects” and “code improvement” seems to corroborate our hypothesis that the difference in the number of comments about these two items in review comments is mostly due to understanding issues. Thus, if managers and developers want code review to match their need for “finding defects,” context and change understanding must be improved.

#### B. A Priori Understanding

By observing developers performing code reviews, we noticed that some started code reviews by thoroughly reading the accompanying textual description, while others went directly to a specific changed file. In the first group, the time required for putting the first review comments and understanding the change rationale was noticeably longer, and some of the comments were asking to clarify the reasons for a change. To better comprehend this situation, we included in our interview guideline a question about how the interviewees start code reviews. Participants explained that when they own or are very familiar with the files being changed, they have a

better context and it is easier for them to understand the change submitted: “*when doing code review I start with things I am familiar with, so it is easier to see what is going on.*” When they are file owners, they often do not even need to read the description, but they “*go directly to the files they own.*” On the contrary, when they do not own files, or have to review new files, they need more information and try to get it from the description, which is deemed good when it states “*what was changed and why.*”

To better understand this aspect we included two questions in the programmers’ survey to know (1) whether it takes longer to review files they are not familiar with, and why; and (2) whether reviewers familiar with the changed files give different feedback, and how.

Most of the respondents (798, 91%) answered positively to the first question, motivating it with the fact that it takes time to familiarize with the code and “*learn enough about the files being modified to understand their purpose, invariants, APIs, etc.,*” because “*big-picture impact analysis requires contextual understanding. When reviewing a small, unfamiliar change, it is often necessary to read through much more code than that being reviewed.*” The comment of a developer anticipates the answer to the second question: “*It takes a lot longer to understand unknown code, but even then understanding isn’t very deep. With code I am familiar with I have more to say. I know what to say faster. What I have to say is deeper. And I can be more insistent on it.*” In fact, the answer to the second question is positive in 716 (82%) cases. The main difference with file owner comments is that they are substantially deeper, more detailed and insightful. A respondent explained: “*Comments reflect their deeper understanding -- more likely to find subtle defects, feedback is more conceptual (better ideas, approaches) instead of superficial (naming, mechanical style, etc.)*” another tried to boldly summarize the concept: “*Difference between algorithmic analysis and comments on coding style. The difference is big.*”

In fact, when the context is clear and understanding is very high, as in the case when the reviewer is the owner of changed files, code review authors receive comments that explore “*deeper details,*” are “*more directed*” and “*more actionable and pertinent,*” and find “*more subtle issues.*”

#### C. Dealing with Understanding Needs

From the interviews, we found that, in the current situation, reviewers try different paths to understand the context and the changes: They read the change description, try to run the changed code, send emails for understanding high level details about the review, and often (from 20% to 40% of the times) even go to talk in person to have a “*higher communication bandwidth*” for asking clarifications to the author. All code review tools that we see in practice today deliver only basic support for the understanding needs of reviewers – providing features such as diffing capabilities, inline commenting, or syntax highlighting, which are limited when dealing with complex code understanding.

## VII. RECOMMENDATIONS AND IMPLICATIONS

### A. Recommendations for Practitioners

From our work we derive recommendations to developers:

**Quality Assurance:** There is a mismatch between the expectations and the actual outcomes of code reviews. From our study, review does not result in identifying defects as often as project members would like and even more rarely detects deep, subtle, or “macro” level issues. Relying on code review in this way for quality assurance may be fraught.

**Understanding:** When reviewers have *a priori* knowledge of the context and the code, they complete reviews more quickly and provide more valuable feedback to the author. Teams should aim to increase the breadth of understanding of developers (if the author of a change is the only expert, she has no potential reviewers) and change authors should include code owners and others with understanding as much as possible when using review to identify defects. Developers indicated that when the author provided context and direction to them in a review, they could respond better and faster.

**Beyond Defects:** Modern code reviews provide benefits beyond finding defects. Code review can be used to improve code style, find alternative solutions, increase learning, share code ownership, *etc.* This should guide code review policies.

**Communication:** Despite the growth of tools for supporting code reviews, developers still have need of richer communication than comments annotating the changed code when reviewing. Teams should provide mechanisms for in-person or, at least, synchronous communication.

### B. Implications for Researchers

Our work uncovered aspects of code review—beyond our research questions—that deserve further study:

**Automate Code Review Tasks:** We observed that many code review comments were related to “code improvement” concerns and low-level “micro” defects. Identifying both of these are problems that research has begun to solve. Tools for enforcing team code conventions, checking for typos, and identifying dead code already exist. Even more advanced tasks such as checking boundary conditions or catching common mistakes have been shown to work in practice on real code. For example Google experimented with adding FindBugs to their review process, though little is reported about the results [32]. Automating these tasks frees reviewers to look for deeper, more subtle defects. Code review is fertile ground to have an impact with code analysis tools.

**Program Comprehension in Practice:** We identified context and change understanding as challenges that developers face when reviewing, with a direct relationship to the quality of review comments. Interestingly, modern IDEs ship with many tools to aid context and understanding, and there is an entire conference (ICPC) devoted to code comprehension, yet all current code review tools we know of show a highlighted diff of the changed files to a reviewer with no additional tool support. The most common motivation that we have seen for code comprehension research is a developer that is working on new code, but we argue that reviewers reviewing code they have not seen before may be more common than a developer

working on new code. This is a ripe opportunity for code understanding researchers to have impact on real world scenarios.

**Socio-technical Effects:** Awareness and learning were cited as motivations for code review, but these outcomes are difficult to observe from traces in reviews. We did not investigate these further, but studies can be designed and carried out to determine if and how awareness and learning increase as a result of being involved in code review.

## VIII. LIMITATIONS

As a qualitative study, gauging the validity of our findings is a difficult undertaking [33]. While we have endeavored to uncover and report the expectations, outcomes, and challenges of code review, limitations may exist. We describe them with the steps that we took to increase confidence and validity.

To achieve a comprehensive view of code review, we triangulated by collecting and comparing results from multiple sources. For example, we found strong agreement among the results of expectations collected from interviews, surveys of manager, and surveys of developers. By starting with exploratory interviews of a smaller set of subjects (17) followed by open coding to extract themes, we identified core questions that we addressed to a larger audience via survey.

One common notion is that empirical research within one company or one project provides little value for the academic community, and does not contribute to scientific development. Historical evidence shows otherwise. Flyvbjerg provides several examples of individual cases that contributed to discovery in physics, economics, and social science [34]. Beveridge observed for social sciences: “*More discoveries have arisen from intense observation than from statistics applied to large groups*” (as quoted in Kuper and Kuper [35], page 95). This should not be interpreted as a criticism of research that focuses on large samples. For the development of an empirical body of knowledge as championed by Basili [36], both types of research are essential. To understand code review across many contexts, we observed, interviewed, surveyed, and examined code reviews from developers across a diverse group of software teams that work with codebases in various domains, of varying sizes, and with varying processes.

Concerning the representativeness of our results in other contexts, other companies and OSS use tools similar to CodeFlow [8] [7] [6]. However, team dynamics may differ. The need for code understanding may already be met in contexts where projects are smaller or there is shared code ownership and a broad system understanding across the team. We found that higher levels of understanding lead to more informative comments, which identify defects or aid the author in other ways so review in these contexts may uncover more defects. In OSS contexts, project-specific expertise often must be demonstrated prior to being accepted as a “core committer” [37], so learning may not be as important or frequent an outcome for review.

In this work, we have used discussions within CodeFlow to identify and quantify outcomes of code review. However, some motivations that managers and developers described are

not easily observable because they leave little trace. For example, determining how often code review improves team awareness or transfers knowledge is difficult to assess from the discussions in reviews. For these outcomes, we have responses indicating that they occur, but not "hard evidence."

Based on review comments, survey responses, and interviews, we know that in-person discussions occurred frequently. While we are unable to compare frequency of these events to other outcomes as we can with events that are recorded in CodeFlow, we know that they most often occurred to address understanding needs.

## IX. CONCLUSION

In this work, we have investigated modern, tool-based code review, uncovered both a wide range of motivations for review, and determined that the outcomes do not always match those motivations. We identified understanding as a key component and provided recommendations to both practitioners and researchers. It is our hope that the insights we have discovered lead to more effective review in practice and improved tools, based on research, to aid developers perform code reviews.

## REFERENCES

- [1] A. Frank Ackerman, Priscilla J. Fowler, and Robert G. Ebenau, "Software inspections and the industrial production of software," in *Proc. of a symposium on Software validation: inspection-testing-verification-alternatives*, 1984, pp. 13--40.
- [2] A.F. Ackerman, L.S. Buchwald, and F.H. Lewski, "Software inspections: an Effective Verification Process," *IEEE Software*, 1989.
- [3] M.E. Fagan, "Design and code inspections to reduce errors in program development," *IBM Systems Journal*, 1976.
- [4] Shull, "Inspecting the History of Inspections," 2008.
- [5] P. Rigby, B. Cleary, F. Painchaud, M. Storey, and D. German, "Open Source Peer Review – Lessons and Recommendations for," *IEEE Software*, 2012.
- [6] Niall Kennedy. (2006, Dec) How Google does web-based code reviews with Mondrian. [Online]. <http://www.niallkennedy.com/blog/2006/11/google-mondrian.html>
- [7] Alexia Tsotsis. (2011, August) Meet Phabricator, The Witty Code Review Tool Built Inside Facebook. [Online]. <http://techcrunch.com/2011/08/07/oh-what-noble-scribe-hath-penned-these-words/>
- [8] (2012, June) Gerrit (software). Wikipedia Article. [Online]. [http://en.wikipedia.org/wiki/Gerrit\\_\(software\)](http://en.wikipedia.org/wiki/Gerrit_(software))
- [9] O. Laitenberger, "A Survey of Software Inspection Technologies," in *Handbook on Software Engineering and Knowledge Engineering*., 2002, pp. 517-555.
- [10] J.P. Johnson, "Collaboration, Peer Review, and Open Source Software," *Information Economics and Policy*, vol. 18, pp. 477-497, 2006.
- [11] A. Porter, H. Siy, and L. Votta, "A review of software inspections," *Advances in Computers*, vol. 42, pp. 39-76, 1996.
- [12] L.G. Votta, "Does every inspection need a meeting?," *ACM SIGSOFT Software Engineering Notes*, vol. 18, pp. 107--114, 1993.
- [13] L. Brothers, V. Sembugamoorthy, and M. Muller, "ICICLE: groupware for code inspection," in *Conference on Computer-Supported Cooperative Work*, 1990, pp. 169--181.
- [14] Vahid Mashayekhi, Chris Feulner, and John Riedl, "CAIS: collaborative asynchronous inspection of software," *SIGSOFT Softw. Eng. Notes*, vol. 19, pp. 21--34, dec 1994.
- [15] John Gintell et al., "Scrutiny: A Collaborative Inspection and Review System," in *Proceedings of the 4th European Software Engineering Conference*, 1993, pp. 344--360.
- [16] P. Rigby, D. German, and M. Storey, "Open source software peer review practices: a case study of the apache server," in *Proceedings of ICSE*, 2008.
- [17] A. Sutherland and G. Venolia, "Can peer code reviews be exploited for later information needs?," in *Proceedings of ICSE*, may 2009.
- [18] T. LaToza, G. Venolia, and R. DeLine, "Maintaining mental models: a study of developer work habits," in *Proceedings of ICSE*, 2006.
- [19] John G. Adair, "The Hawthorne effect: A reconsideration of the methodological artifact," *Journal of Applied Psychology*, vol. 69, no. 2, pp. 334-345, 1984.
- [20] J. Creswell, *Research Design: Qualitative, Quantitative, and Mixed Methods Approaches*, 3rd ed.: Sage Publications, Inc., 2009.
- [21] B. L. Berg, *Qualitative Research Methods for Social Sciences*. Boston: Pearson, 2004.
- [22] S., W. Hall, and P. Kruchten Adolph, "Using Grounded Theory to Study the Experience of Software Development," *Empirical Software Engineering*, vol. 16, no. 4, pp. 487-513, 2011.
- [23] T. Lindlof and B. Taylor, *Qualitative Communication Research Methods*.: Sage, 2002.
- [24] R. S. Weiss, *Learning From Strangers: The Art and Method of Qualitative Interview Studies*.: The Free Press, 1995.
- [25] I. Barker. (2005, May) What is information architecture? KM Column. [Online]. <http://www.steptwo.com.au/>
- [26] Janice E. Shade Stuart J. Janis, *Improving Performance Through Statistical Thinking*.: McGraw-Hill, 2000.
- [27] A. Bacchelli and C. Bird, "Appendix to Expectations, Outcomes, and Challenges of Modern Code Review," Microsoft Research, Technical Report MSR-TR-2012-83 2012. [Online]. <http://research.microsoft.com/apps/pubs/?id=171426>
- [28] B. A. Kitchenham and S. L. Pfleeger, "Personal Opinion Surveys," in *Guide to Advanced Empirical Software Engineering*, Forrest Shull, Janice Singer, and Dag I. K. Sjøberg, Eds.: Springer, 2007.
- [29] P. K. Tyagi, "The Effects of Appeals, Anonymity, and Feedback on Mail Survey Response Patterns from Salespeople," *Journal of The Academy of Marketing Science*, 1989.
- [30] T. Punter, M. Ciolkowski, B. G. Freimut, and I. John, "Conducting on-line surveys in software engineering," in *Proc. of International Symposium on Empirical Software Engineering*, 2003.
- [31] T. Lethbridge, S. Sim, and J. Singer, "Studying software engineers: Data collection techniques for software field studies," *Empirical Software Engineering*, vol. 10, pp. 311--341, 2005.
- [32] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou, "Using FindBugs on production software," in *OOPSLA '07*, 2007.
- [33] N. Golafshani, "Understanding reliability and validity in qualitative research," *The qualitative report*, vol. 8, pp. 597--607, 2003.
- [34] B. Flyvbjerg, "Five misunderstandings about case-study research," *Qualitative inquiry*, vol. 12, no. 2, pp. 219-245, 2006.
- [35] A. Kuper and J. Kuper, *The Social Science Encyclopedia*.: Routledge, 1985.
- [36] V. Basili, F. Shull, and F. Lanubile, "Building knowledge through families of experiments..," *IEEE Trans. Software Eng.*, vol. 25, no. 4, pp. 456-173, 1999.
- [37] C. Bird, A. Gourley, P. Devanbu, A. Swaminathan, and G. Hsu, "Open Borders? Immigration in Open Source Projects," in *International Workshop on Mining Software Repositories*, 2007.



# 4

---

# Requirements engineering

## Objectives

The objective of this chapter is to introduce software requirements and to explain the processes involved in discovering and documenting these requirements. When you have read the chapter, you will:

- understand the concepts of user and system requirements and why these requirements should be written in different ways;
- understand the differences between functional and non-functional software requirements;
- understand the main requirements engineering activities of elicitation, analysis, and validation, and the relationships between these activities;
- understand why requirements management is necessary and how it supports other requirements engineering activities.

## Contents

- 4.1 Functional and non-functional requirements**
- 4.2 Requirements engineering processes**
- 4.3 Requirements elicitation**
- 4.4 Requirements specification**
- 4.5 Requirements validation**
- 4.6 Requirements change**

The requirements for a system are the descriptions of the services that a system should provide and the constraints on its operation. These requirements reflect the needs of customers for a system that serves a certain purpose such as controlling a device, placing an order, or finding information. The process of finding out, analyzing, documenting and checking these services and constraints is called requirements engineering (RE).

The term *requirement* is not used consistently in the software industry. In some cases, a requirement is simply a high-level, abstract statement of a service that a system should provide or a constraint on a system. At the other extreme, it is a detailed, formal definition of a system function. Davis (Davis 1993) explains why these differences exist:

*If a company wishes to let a contract for a large software development project, it must define its needs in a sufficiently abstract way that a solution is not pre-defined. The requirements must be written so that several contractors can bid for the contract, offering, perhaps, different ways of meeting the client organization's needs. Once a contract has been awarded, the contractor must write a system definition for the client in more detail so that the client understands and can validate what the software will do. Both of these documents may be called the requirements document for the system<sup>†</sup>.*

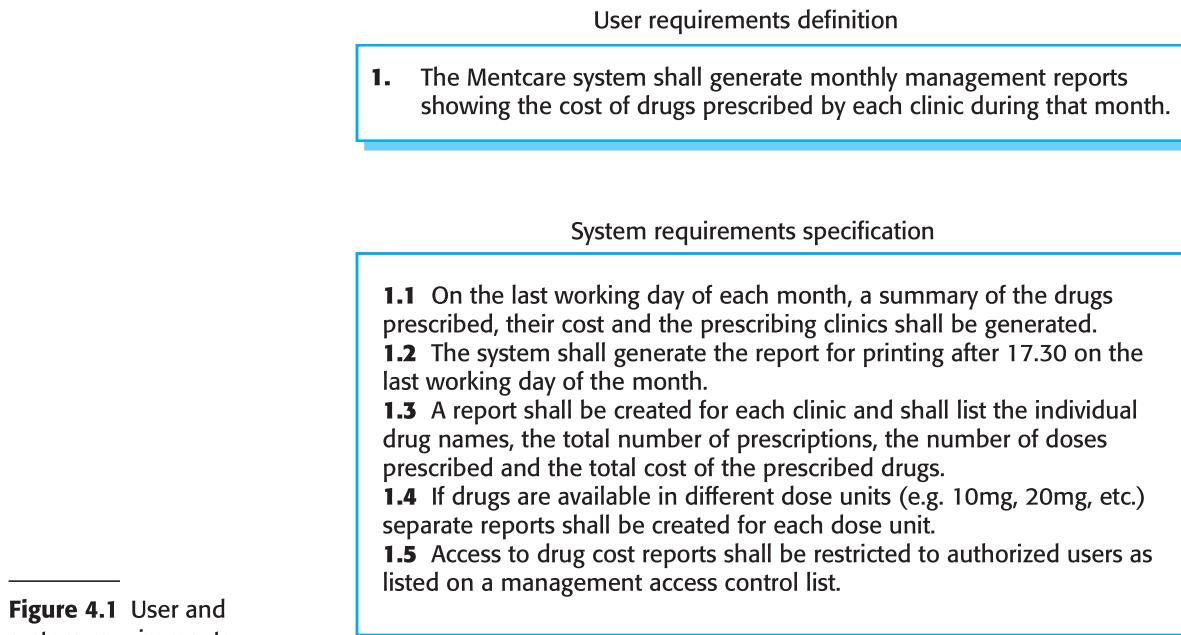
Some of the problems that arise during the requirements engineering process are a result of failing to make a clear separation between these different levels of description. I distinguish between them by using the term *user requirements* to mean the high-level abstract requirements and *system requirements* to mean the detailed description of what the system should do. User requirements and system requirements may be defined as follows:

1. User requirements are statements, in a natural language plus diagrams, of what services the system is expected to provide to system users and the constraints under which it must operate. The user requirements may vary from broad statements of the system features required to detailed, precise descriptions of the system functionality.
2. System requirements are more detailed descriptions of the software system's functions, services, and operational constraints. The system requirements document (sometimes called a functional specification) should define exactly what is to be implemented. It may be part of the contract between the system buyer and the software developers.

Different kinds of requirement are needed to communicate information about a system to different types of reader. Figure 4.1 illustrates the distinction between user and system requirements. This example from the mental health care patient information system (Mentcare) shows how a user requirement may be expanded into several system requirements. You can see from Figure 4.1 that the user requirement is quite

---

<sup>†</sup>Davis, A. M. 1993. Software Requirements: Objects, Functions and States. Englewood Cliffs, NJ: Prentice-Hall.

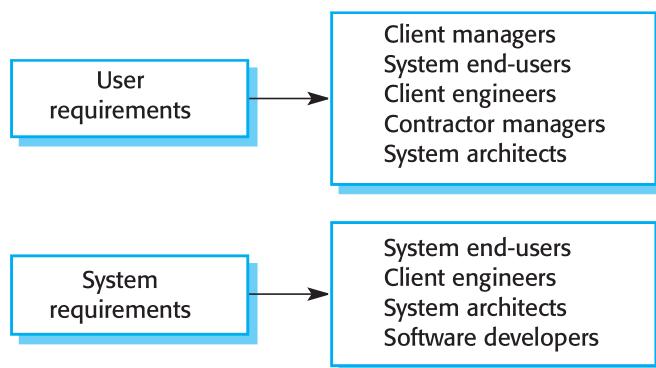


**Figure 4.1** User and system requirements

general. The system requirements provide more specific information about the services and functions of the system that is to be implemented.

You need to write requirements at different levels of detail because different types of readers use them in different ways. Figure 4.2 shows the types of readers of the user and system requirements. The readers of the user requirements are not usually concerned with how the system will be implemented and may be managers who are not interested in the detailed facilities of the system. The readers of the system requirements need to know more precisely what the system will do because they are concerned with how it will support the business processes or because they are involved in the system implementation.

The different types of document readers shown in Figure 4.2 are examples of system stakeholders. As well as users, many other people have some kind of interest in the system. System stakeholders include anyone who is affected by the system in some way and so anyone who has a legitimate interest in it. Stakeholders range from end-users of a system through managers to external stakeholders such as regulators,



**Figure 4.2** Readers of different types of requirements specification



### Feasibility studies

A feasibility study is a short, focused study that should take place early in the RE process. It should answer three key questions: (1) Does the system contribute to the overall objectives of the organization? (2) Can the system be implemented within schedule and budget using current technology? and (3) Can the system be integrated with other systems that are used?

If the answer to any of these questions is no, you should probably not go ahead with the project.

<http://software-engineering-book.com/web/feasibility-study/>

who certify the acceptability of the system. For example, system stakeholders for the Mentcare system include:

1. Patients whose information is recorded in the system and relatives of these patients.
2. Doctors who are responsible for assessing and treating patients.
3. Nurses who coordinate the consultations with doctors and administer some treatments.
4. Medical receptionists who manage patients' appointments.
5. IT staff who are responsible for installing and maintaining the system.
6. A medical ethics manager who must ensure that the system meets current ethical guidelines for patient care.
7. Health care managers who obtain management information from the system.
8. Medical records staff who are responsible for ensuring that system information can be maintained and preserved, and that record keeping procedures have been properly implemented.

Requirements engineering is usually presented as the first stage of the software engineering process. However, some understanding of the system requirements may have to be developed before a decision is made to go ahead with the procurement or development of a system. This early-stage RE establishes a high-level view of what the system might do and the benefits that it might provide. These may then be considered in a feasibility study, which tries to assess whether or not the system is technically and financially feasible. The results of that study help management decide whether or not to go ahead with the procurement or development of the system.

In this chapter, I present a “traditional” view of requirements rather than requirements in agile processes, which I discussed in Chapter 3. For the majority of large systems, it is still the case that there is a clearly identifiable requirements engineering phase before implementation of the system begins. The outcome is a requirements document, which may be part of the system development contract. Of course, subsequent changes are made to the requirements, and user requirements may be expanded into

more detailed system requirements. Sometimes an agile approach of concurrently eliciting the requirements as the system is developed may be used to add detail and to refine the user requirements.

## 4.1 Functional and non-functional requirements

Software system requirements are often classified as functional or non-functional requirements:

1. *Functional requirements* These are statements of services the system should provide, how the system should react to particular inputs, and how the system should behave in particular situations. In some cases, the functional requirements may also explicitly state what the system should not do.
2. *Non-functional requirements* These are constraints on the services or functions offered by the system. They include timing constraints, constraints on the development process, and constraints imposed by standards. Non-functional requirements often apply to the system as a whole rather than individual system features or services.

In reality, the distinction between different types of requirements is not as clear-cut as these simple definitions suggest. A user requirement concerned with security, such as a statement limiting access to authorized users, may appear to be a non-functional requirement. However, when developed in more detail, this requirement may generate other requirements that are clearly functional, such as the need to include user authentication facilities in the system.

This shows that requirements are not independent and that one requirement often generates or constrains other requirements. The system requirements therefore do not just specify the services or the features of the system that are required; they also specify the necessary functionality to ensure that these services/features are delivered effectively.

### 4.1.1 Functional requirements

The functional requirements for a system describe what the system should do. These requirements depend on the type of software being developed, the expected users of the software, and the general approach taken by the organization when writing requirements. When expressed as user requirements, functional requirements should be written in natural language so that system users and managers can understand them. Functional system requirements expand the user requirements and are written for system developers. They should describe the system functions, their inputs and outputs, and exceptions in detail.

Functional system requirements vary from general requirements covering what the system should do to very specific requirements reflecting local ways of working or an organization's existing systems. For example, here are examples of functional



### Domain requirements

Domain requirements are derived from the application domain of the system rather than from the specific needs of system users. They may be new functional requirements in their own right, constrain existing functional requirements, or set out how particular computations must be carried out.

The problem with domain requirements is that software engineers may not understand the characteristics of the domain in which the system operates. This means that these engineers may not know whether or not a domain requirement has been missed out or conflicts with other requirements.

<http://software-engineering-book.com/web/domain-requirements/>

requirements for the Mentcare system, used to maintain information about patients receiving treatment for mental health problems:

1. A user shall be able to search the appointments lists for all clinics.
2. The system shall generate each day, for each clinic, a list of patients who are expected to attend appointments that day.
3. Each staff member using the system shall be uniquely identified by his or her eight-digit employee number.

These user requirements define specific functionality that should be included in the system. The requirements show that functional requirements may be written at different levels of detail (contrast requirements 1 and 3).

Functional requirements, as the name suggests, have traditionally focused on what the system should do. However, if an organization decides that an existing off-the-shelf system software product can meet its needs, then there is very little point in developing a detailed functional specification. In such cases, the focus should be on the development of information requirements that specify the information needed for people to do their work. Information requirements specify the information needed and how it is to be delivered and organized. Therefore, an information requirement for the Mentcare system might specify what information is to be included in the list of patients expected for appointments that day.

Imprecision in the requirements specification can lead to disputes between customers and software developers. It is natural for a system developer to interpret an ambiguous requirement in a way that simplifies its implementation. Often, however, this is not what the customer wants. New requirements have to be established and changes made to the system. Of course, this delays system delivery and increases costs.

For example, the first Mentcare system requirement in the above list states that a user shall be able to search the appointments lists for all clinics. The rationale for this requirement is that patients with mental health problems are sometimes confused. They may have an appointment at one clinic but actually go to a different clinic. If they have an appointment, they will be recorded as having attended, regardless of the clinic.

A medical staff member specifying a search requirement may expect “search” to mean that, given a patient name, the system looks for that name in all appointments at all clinics. However, this is not explicit in the requirement. System developers may interpret the requirement so that it is easier to implement. Their search function may require the user to choose a clinic and then carry out the search of the patients who attended that clinic. This involves more user input and so takes longer to complete the search.

Ideally, the functional requirements specification of a system should be both complete and consistent. Completeness means that all services and information required by the user should be defined. Consistency means that requirements should not be contradictory.

In practice, it is only possible to achieve requirements consistency and completeness for very small software systems. One reason is that it is easy to make mistakes and omissions when writing specifications for large, complex systems. Another reason is that large systems have many stakeholders, with different backgrounds and expectations. Stakeholders are likely to have different—and often inconsistent—needs. These inconsistencies may not be obvious when the requirements are originally specified, and the inconsistent requirements may only be discovered after deeper analysis or during system development.

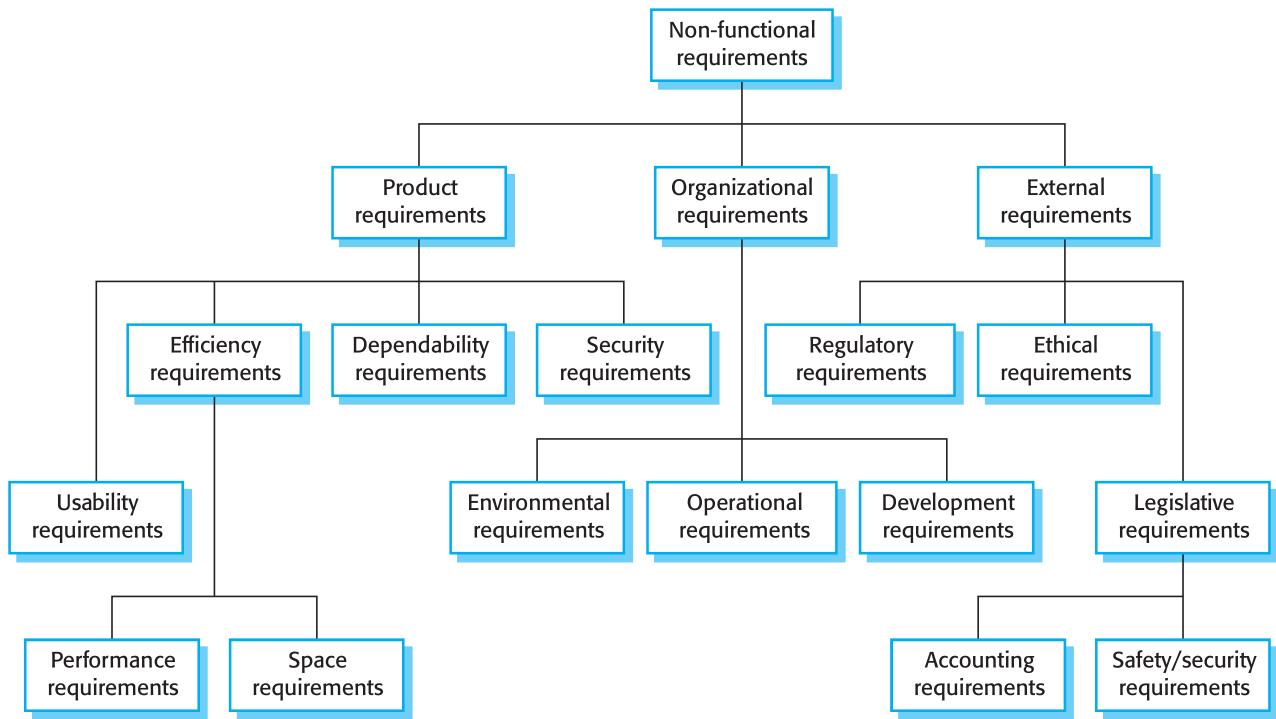
#### 4.1.2 Non-functional requirements

Non-functional requirements, as the name suggests, are requirements that are not directly concerned with the specific services delivered by the system to its users. These non-functional requirements usually specify or constrain characteristics of the system as a whole. They may relate to emergent system properties such as reliability, response time, and memory use. Alternatively, they may define constraints on the system implementation, such as the capabilities of I/O devices or the data representations used in interfaces with other systems.

Non-functional requirements are often more critical than individual functional requirements. System users can usually find ways to work around a system function that doesn't really meet their needs. However, failing to meet a non-functional requirement can mean that the whole system is unusable. For example, if an aircraft system does not meet its reliability requirements, it will not be certified as safe for operation; if an embedded control system fails to meet its performance requirements, the control functions will not operate correctly.

While it is often possible to identify which system components implement specific functional requirements (e.g., there may be formatting components that implement reporting requirements), this is often more difficult with non-functional requirements. The implementation of these requirements may be spread throughout the system, for two reasons:

1. Non-functional requirements may affect the overall architecture of a system rather than the individual components. For example, to ensure that performance requirements are met in an embedded system, you may have to organize the system to minimize communications between components.



**Figure 4.3** Types of non-functional requirements

2. An individual non-functional requirement, such as a security requirement, may generate several, related functional requirements that define new system services that are required if the non-functional requirement is to be implemented. In addition, it may also generate requirements that constrain existing requirements; for example, it may limit access to information in the system.

Nonfunctional requirements arise through user needs because of budget constraints, organizational policies, the need for interoperability with other software or hardware systems, or external factors such as safety regulations or privacy legislation. Figure 4.3 is a classification of non-functional requirements. You can see from this diagram that the non-functional requirements may come from required characteristics of the software (product requirements), the organization developing the software (organizational requirements), or external sources:

1. *Product requirements* These requirements specify or constrain the runtime behavior of the software. Examples include performance requirements for how fast the system must execute and how much memory it requires; reliability requirements that set out the acceptable failure rate; security requirements; and usability requirements.
2. *Organizational requirements* These requirements are broad system requirements derived from policies and procedures in the customer's and developer's organizations. Examples include operational process requirements that define how the system will be used; development process requirements that specify the

**PRODUCT REQUIREMENT**

The Mentcare system shall be available to all clinics during normal working hours (Mon–Fri, 08:30–17:30). Downtime within normal working hours shall not exceed 5 seconds in any one day.

**ORGANIZATIONAL REQUIREMENT**

Users of the Mentcare system shall identify themselves using their health authority identity card.

**EXTERNAL REQUIREMENT**

The system shall implement patient privacy provisions as set out in HStan-03-2006-priv.

**Figure 4.4** Examples of possible non-functional requirements for the Mentcare system

programming language; the development environment or process standards to be used; and environmental requirements that specify the operating environment of the system.

3. *External requirements* This broad heading covers all requirements that are derived from factors external to the system and its development process. These may include regulatory requirements that set out what must be done for the system to be approved for use by a regulator, such as a nuclear safety authority; legislative requirements that must be followed to ensure that the system operates within the law; and ethical requirements that ensure that the system will be acceptable to its users and the general public.

Figure 4.4 shows examples of product, organizational, and external requirements that could be included in the Mentcare system specification. The product requirement is an availability requirement that defines when the system has to be available and the allowed downtime each day. It says nothing about the functionality of the Mentcare system and clearly identifies a constraint that has to be considered by the system designers.

The organizational requirement specifies how users authenticate themselves to the system. The health authority that operates the system is moving to a standard authentication procedure for all software where, instead of users having a login name, they swipe their identity card through a reader to identify themselves. The external requirement is derived from the need for the system to conform to privacy legislation. Privacy is obviously a very important issue in health care systems, and the requirement specifies that the system should be developed in accordance with a national privacy standard.

A common problem with non-functional requirements is that stakeholders propose requirements as general goals, such as ease of use, the ability of the system to recover from failure, or rapid user response. Goals set out good intentions but cause problems for system developers as they leave scope for interpretation and subsequent dispute once the system is delivered. For example, the following system goal is typical of how a manager might express usability requirements:

*The system should be easy to use by medical staff and should be organized in such a way that user errors are minimized.*

Property	Measure
Speed	Processed transactions/second User/event response time Screen refresh time
Size	Megabytes/Number of ROM chips
Ease of use	Training time Number of help frames
Reliability	Mean time to failure Probability of unavailability Rate of failure occurrence Availability
Robustness	Time to restart after failure Percentage of events causing failure Probability of data corruption on failure
Portability	Percentage of target dependent statements Number of target systems

**Figure 4.5** Metrics for

specifying non-

functional requirements

I have rewritten this to show how the goal could be expressed as a “testable” non-functional requirement. It is impossible to objectively verify the system goal, but in the following description you can at least include software instrumentation to count the errors made by users when they are testing the system.

*Medical staff shall be able to use all the system functions after two hours of training. After this training, the average number of errors made by experienced users shall not exceed two per hour of system use.*

Whenever possible, you should write non-functional requirements quantitatively so that they can be objectively tested. Figure 4.5 shows metrics that you can use to specify non-functional system properties. You can measure these characteristics when the system is being tested to check whether or not the system has met its non-functional requirements.

In practice, customers for a system often find it difficult to translate their goals into measurable requirements. For some goals, such as maintainability, there are no simple metrics that can be used. In other cases, even when quantitative specification is possible, customers may not be able to relate their needs to these specifications. They don’t understand what some number defining the reliability (for example) means in terms of their everyday experience with computer systems. Furthermore, the cost of objectively verifying measurable, non-functional requirements can be very high, and the customers paying for the system may not think these costs are justified.

Non-functional requirements often conflict and interact with other functional or non-functional requirements. For example, the identification requirement in Figure 4.4 requires a card reader to be installed with each computer that connects to the system. However, there may be another requirement that requests mobile access to the system from doctors’ or nurses’ tablets or smartphones. These are not normally

equipped with card readers so, in these circumstances, some alternative identification method may have to be supported.

It is difficult to separate functional and non-functional requirements in the requirements document. If the non-functional requirements are stated separately from the functional requirements, the relationships between them may be hard to understand. However, you should, ideally, highlight requirements that are clearly related to emergent system properties, such as performance or reliability. You can do this by putting them in a separate section of the requirements document or by distinguishing them, in some way, from other system requirements.

Non-functional requirements such as reliability, safety, and confidentiality requirements are particularly important for critical systems. I cover these dependability requirements in Part 2, which describes ways of specifying reliability, safety, and security requirements.

## 4.2 Requirements engineering processes

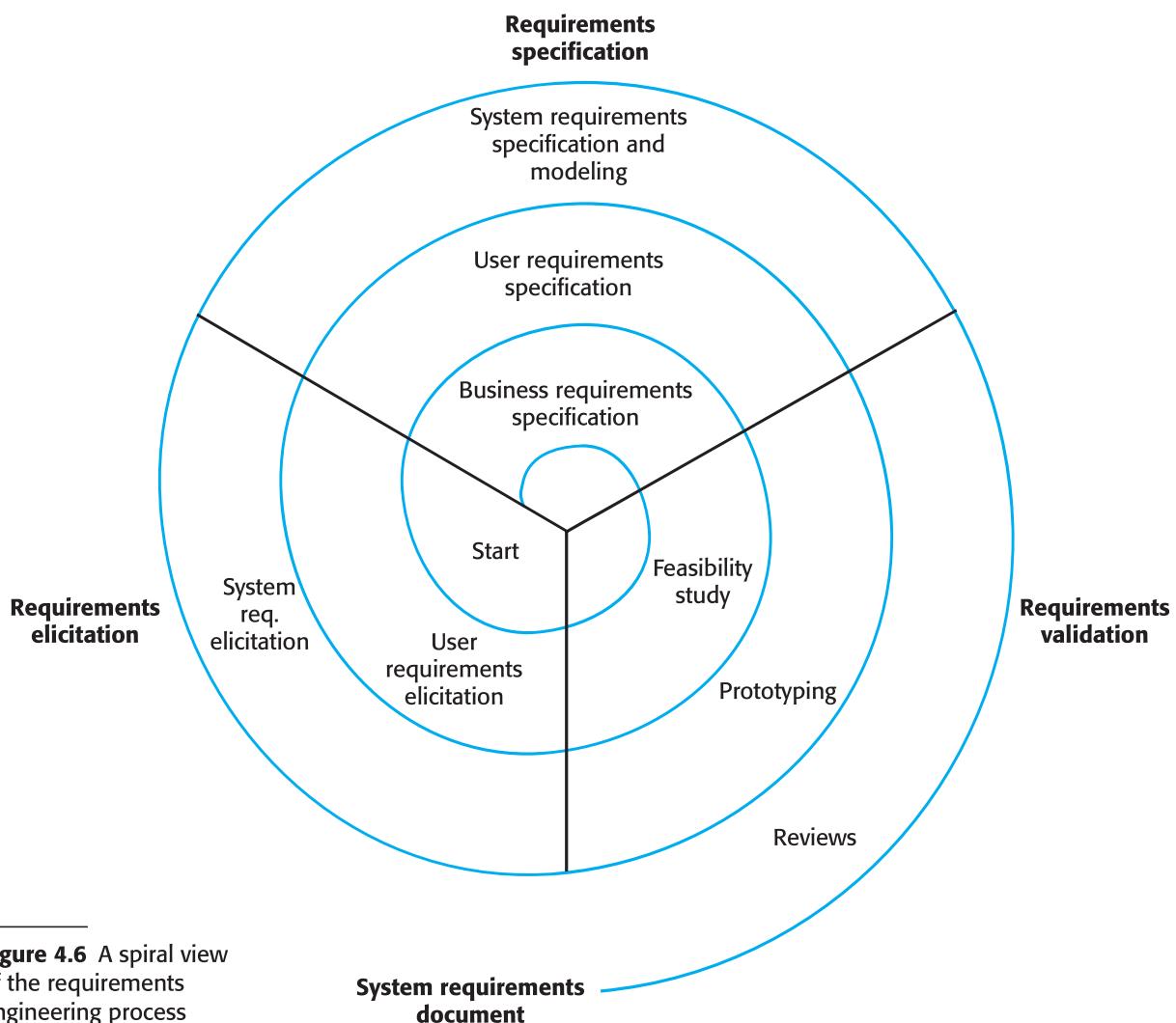
As I discussed in Chapter 2, requirements engineering involves three key activities. These are discovering requirements by interacting with stakeholders (elicitation and analysis); converting these requirements into a standard form (specification); and checking that the requirements actually define the system that the customer wants (validation). I have shown these as sequential processes in Figure 2.4. However, in practice, requirements engineering is an iterative process in which the activities are interleaved.

Figure 4.6 shows this interleaving. The activities are organized as an iterative process around a spiral. The output of the RE process is a system requirements document. The amount of time and effort devoted to each activity in an iteration depends on the stage of the overall process, the type of system being developed, and the budget that is available.

Early in the process, most effort will be spent on understanding high-level business and non-functional requirements, and the user requirements for the system. Later in the process, in the outer rings of the spiral, more effort will be devoted to eliciting and understanding the non-functional requirements and more detailed system requirements.

This spiral model accommodates approaches to development where the requirements are developed to different levels of detail. The number of iterations around the spiral can vary so that the spiral can be exited after some or all of the user requirements have been elicited. Agile development can be used instead of prototyping so that the requirements and the system implementation are developed together.

In virtually all systems, requirements change. The people involved develop a better understanding of what they want the software to do; the organization buying the system changes; and modifications are made to the system's hardware, software, and organizational environment. Changes have to be managed to understand the impact on other requirements and the cost and system implications of making the change. I discuss this process of requirements management in Section 4.6.



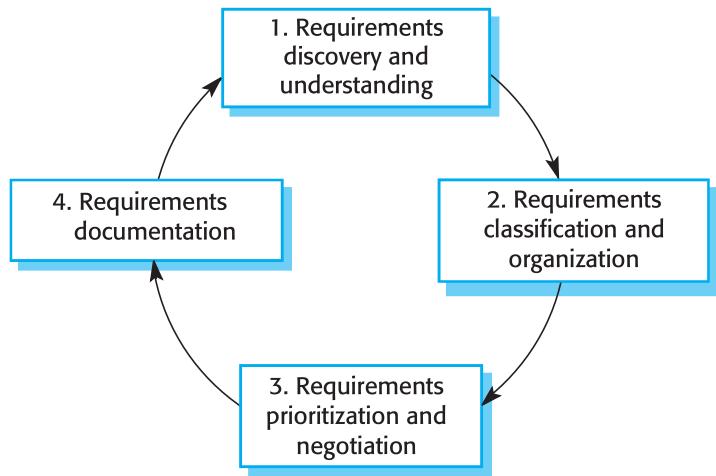
**Figure 4.6** A spiral view of the requirements engineering process

### 4.3 Requirements elicitation

The aims of the requirements elicitation process are to understand the work that stakeholders do and how they might use a new system to help support that work. During requirements elicitation, software engineers work with stakeholders to find out about the application domain, work activities, the services and system features that stakeholders want, the required performance of the system, hardware constraints, and so on.

Eliciting and understanding requirements from system stakeholders is a difficult process for several reasons:

1. Stakeholders often don't know what they want from a computer system except in the most general terms; they may find it difficult to articulate what they want the system to do; they may make unrealistic demands because they don't know what is and isn't feasible.



**Figure 4.7** The requirements elicitation and analysis process

2. Stakeholders in a system naturally express requirements in their own terms and with implicit knowledge of their own work. Requirements engineers, without experience in the customer's domain, may not understand these requirements.
3. Different stakeholders, with diverse requirements, may express their requirements in different ways. Requirements engineers have to discover all potential sources of requirements and discover commonalities and conflict.
4. Political factors may influence the requirements of a system. Managers may demand specific system requirements because these will allow them to increase their influence in the organization.
5. The economic and business environment in which the analysis takes place is dynamic. It inevitably changes during the analysis process. The importance of particular requirements may change. New requirements may emerge from new stakeholders who were not originally consulted.

A process model of the elicitation and analysis process is shown in Figure 4.7. Each organization will have its own version or instantiation of this general model, depending on local factors such as the expertise of the staff, the type of system being developed, and the standards used.

The process activities are:

1. *Requirements discovery and understanding* This is the process of interacting with stakeholders of the system to discover their requirements. Domain requirements from stakeholders and documentation are also discovered during this activity.
2. *Requirements classification and organization* This activity takes the unstructured collection of requirements, groups related requirements and organizes them into coherent clusters.
3. *Requirements prioritization and negotiation* Inevitably, when multiple stakeholders are involved, requirements will conflict. This activity is concerned with prioritizing requirements and finding and resolving requirements conflicts



### Viewpoints

A viewpoint is a way of collecting and organizing a set of requirements from a group of stakeholders who have something in common. Each viewpoint therefore includes a set of system requirements. Viewpoints might come from end-users, managers, or others. They help identify the people who can provide information about their requirements and structure the requirements for analysis.

<http://www.software-engineering-book.com/web/viewpoints/>

through negotiation. Usually, stakeholders have to meet to resolve differences and agree on compromise requirements.

4. *Requirements documentation* The requirements are documented and input into the next round of the spiral. An early draft of the software requirements documents may be produced at this stage, or the requirements may simply be maintained informally on whiteboards, wikis, or other shared spaces.

Figure 4.7 shows that requirements elicitation and analysis is an iterative process with continual feedback from each activity to other activities. The process cycle starts with requirements discovery and ends with the requirements documentation. The analyst's understanding of the requirements improves with each round of the cycle. The cycle ends when the requirements document has been produced.

To simplify the analysis of requirements, it is helpful to organize and group the stakeholder information. One way of doing so is to consider each stakeholder group to be a viewpoint and to collect all requirements from that group into the viewpoint. You may also include viewpoints to represent domain requirements and constraints from other systems. Alternatively, you can use a model of the system architecture to identify subsystems and to associate requirements with each subsystem.

Inevitably, different stakeholders have different views on the importance and priority of requirements, and sometimes these views are conflicting. If some stakeholders feel that their views have not been properly considered, then they may deliberately attempt to undermine the RE process. Therefore, it is important that you organize regular stakeholder meetings. Stakeholders should have the opportunity to express their concerns and agree on requirements compromises.

At the requirements documentation stage, it is important that you use simple language and diagrams to describe the requirements. This makes it possible for stakeholders to understand and comment on these requirements. To make information sharing easier, it is best to use a shared document (e.g., on Google Docs or Office 365) or a wiki that is accessible to all interested stakeholders.

#### 4.3.1 Requirements elicitation techniques

Requirements elicitation involves meeting with stakeholders of different kinds to discover information about the proposed system. You may supplement this information

with knowledge of existing systems and their usage and information from documents of various kinds. You need to spend time understanding how people work, what they produce, how they use other systems, and how they may need to change to accommodate a new system.

There are two fundamental approaches to requirements elicitation:

1. Interviewing, where you talk to people about what they do.
2. Observation or ethnography, where you watch people doing their job to see what artifacts they use, how they use them, and so on.

You should use a mix of interviewing and observation to collect information and, from that, you derive the requirements, which are then the basis for further discussions.

#### 4.3.1.1 Interviewing

Formal or informal interviews with system stakeholders are part of most requirements engineering processes. In these interviews, the requirements engineering team puts questions to stakeholders about the system that they currently use and the system to be developed. Requirements are derived from the answers to these questions. Interviews may be of two types:

1. Closed interviews, where the stakeholder answers a predefined set of questions.
2. Open interviews, in which there is no predefined agenda. The requirements engineering team explores a range of issues with system stakeholders and hence develops a better understanding of their needs.

In practice, interviews with stakeholders are normally a mixture of both of these. You may have to obtain the answer to certain questions, but these usually lead to other issues that are discussed in a less structured way. Completely open-ended discussions rarely work well. You usually have to ask some questions to get started and to keep the interview focused on the system to be developed.

Interviews are good for getting an overall understanding of what stakeholders do, how they might interact with the new system, and the difficulties that they face with current systems. People like talking about their work, and so they are usually happy to get involved in interviews. However, unless you have a system prototype to demonstrate, you should not expect stakeholders to suggest specific and detailed requirements. Everyone finds it difficult to visualize what a system might be like. You need to analyze the information collected and to generate the requirements from this.

Eliciting domain knowledge through interviews can be difficult, for two reasons:

1. All application specialists use jargon specific to their area of work. It is impossible for them to discuss domain requirements without using this terminology. They normally use words in a precise and subtle way that requirements engineers may misunderstand.

2. Some domain knowledge is so familiar to stakeholders that they either find it difficult to explain or they think it is so fundamental that it isn't worth mentioning. For example, for a librarian, it goes without saying that all acquisitions are catalogued before they are added to the library. However, this may not be obvious to the interviewer, and so it isn't taken into account in the requirements.

Interviews are not an effective technique for eliciting knowledge about organizational requirements and constraints because there are subtle power relationships between the different people in the organization. Published organizational structures rarely match the reality of decision making in an organization, but interviewees may not wish to reveal the actual rather than the theoretical structure to a stranger. In general, most people are generally reluctant to discuss political and organizational issues that may affect the requirements.

To be an effective interviewer, you should bear two things in mind:

1. You should be open-minded, avoid preconceived ideas about the requirements, and willing to listen to stakeholders. If the stakeholder comes up with surprising requirements, then you should be willing to change your mind about the system.
2. You should prompt the interviewee to get discussions going by using a spring-board question or a requirements proposal, or by working together on a prototype system. Saying to people "tell me what you want" is unlikely to result in useful information. They find it much easier to talk in a defined context rather than in general terms.

Information from interviews is used along with other information about the system from documentation describing business processes or existing systems, user observations, and developer experience. Sometimes, apart from the information in the system documents, the interview information may be the only source of information about the system requirements. However, interviewing on its own is liable to miss essential information, and so it should be used in conjunction with other requirements elicitation techniques.

#### **4.3.1.2 Ethnography**

Software systems do not exist in isolation. They are used in a social and organizational environment, and software system requirements may be generated or constrained by that environment. One reason why many software systems are delivered but never used is that their requirements do not take proper account of how social and organizational factors affect the practical operation of the system. It is therefore very important that, during the requirements engineering process, you try to understand the social and organizational issues that affect the use of the system.

Ethnography is an observational technique that can be used to understand operational processes and help derive requirements for software to support these processes. An analyst immerses himself or herself in the working environment where

the system will be used. The day-to-day work is observed, and notes are made of the actual tasks in which participants are involved. The value of ethnography is that it helps discover implicit system requirements that reflect the actual ways that people work, rather than the formal processes defined by the organization.

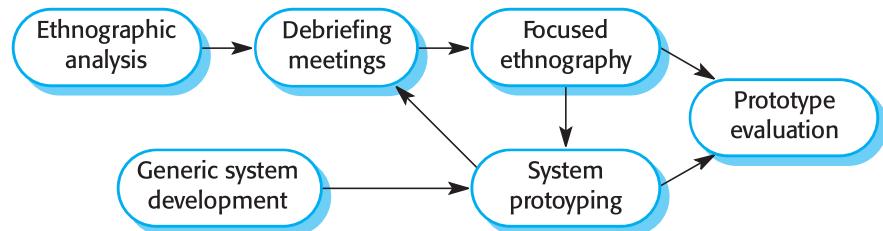
People often find it very difficult to articulate details of their work because it is second nature to them. They understand their own work but may not understand its relationship to other work in the organization. Social and organizational factors that affect the work, but that are not obvious to individuals, may only become clear when noticed by an unbiased observer. For example, a workgroup may self-organize so that members know of each other's work and can cover for each other if someone is absent. This may not be mentioned during an interview as the group might not see it as an integral part of their work.

Suchman (Suchman 1983) pioneered the use of ethnography to study office work. She found that actual work practices were far richer, more complex, and more dynamic than the simple models assumed by office automation systems. The difference between the assumed and the actual work was the most important reason why these office systems had no significant effect on productivity. Crabtree (Crabtree 2003) discusses a wide range of studies since then and describes, in general, the use of ethnography in systems design. In my own research, I have investigated methods of integrating ethnography into the software engineering process by linking it with requirements engineering methods (Viller and Sommerville 2000) and documenting patterns of interaction in cooperative systems (Martin and Sommerville 2004).

Ethnography is particularly effective for discovering two types of requirements:

1. Requirements derived from the way in which people actually work, rather than the way in which business process definitions say they ought to work. In practice, people never follow formal processes. For example, air traffic controllers may switch off a conflict alert system that detects aircraft with intersecting flight paths, even though normal control procedures specify that it should be used. The conflict alert system is sensitive and issues audible warnings even when planes are far apart. Controllers may find these distracting and prefer to use other strategies to ensure that planes are not on conflicting flight paths.
2. Requirements derived from cooperation and awareness of other people's activities. For example, air traffic controllers (ATCs) may use an awareness of other controllers' work to predict the number of aircraft that will be entering their control sector. They then modify their control strategies depending on that predicted workload. Therefore, an automated ATC system should allow controllers in a sector to have some visibility of the work in adjacent sectors.

Ethnography can be combined with the development of a system prototype (Figure 4.8). The ethnography informs the development of the prototype so that fewer prototype refinement cycles are required. Furthermore, the prototyping focuses the ethnography by identifying problems and questions that can then be discussed with the ethnographer. He or she should then look for the answers to these questions during the next phase of the system study (Sommerville et al. 1993).



**Figure 4.8** Ethnography and prototyping for requirements analysis

Ethnography is helpful to understand existing systems, but this understanding does not always help with innovation. Innovation is particularly relevant for new product development. Commentators have suggested that Nokia used ethnography to discover how people used their phones and developed new phone models on that basis; Apple, on the other hand, ignored current use and revolutionized the mobile phone industry with the introduction of the iPhone.

Ethnographic studies can reveal critical process details that are often missed by other requirements elicitation techniques. However, because of its focus on the end-user, this approach is not effective for discovering broader organizational or domain requirements or for suggestion innovations. You therefore have to use ethnography as one of a number of techniques for requirements elicitation.

### 4.3.2 Stories and scenarios

People find it easier to relate to real-life examples than abstract descriptions. They are not good at telling you the system requirements. However, they may be able to describe how they handle particular situations or imagine things that they might do in a new way of working. Stories and scenarios are ways of capturing this kind of information. You can then use these when interviewing groups of stakeholders to discuss the system with other stakeholders and to develop more specific system requirements.

Stories and scenarios are essentially the same thing. They are a description of how the system can be used for some particular task. They describe what people do, what information they use and produce, and what systems they may use in this process. The difference is in the ways that descriptions are structured and in the level of detail presented. Stories are written as narrative text and present a high-level description of system use; scenarios are usually structured with specific information collected such as inputs and outputs. I find stories to be effective in setting out the “big picture.” Parts of stories can then be developed in more detail and represented as scenarios.

Figure 4.9 is an example of a story that I developed to understand the requirements for the iLearn digital learning environment that I introduced in Chapter 1. This story describes a situation in a primary (elementary) school where the teacher is using the environment to support student projects on the fishing industry. You can see this is a very high-level description. Its purpose is to facilitate discussion of how the iLearn system might be used and to act as a starting point for eliciting the requirements for that system.

### Photo sharing in the classroom

Jack is a primary school teacher in Ullapool (a village in northern Scotland). He has decided that a class project should be focused on the fishing industry in the area, looking at the history, development, and economic impact of fishing. As part of this project, pupils are asked to gather and share reminiscences from relatives, use newspaper archives, and collect old photographs related to fishing and fishing communities in the area. Pupils use an iLearn wiki to gather together fishing stories and SCARAN (a history resources site) to access newspaper archives and photographs. However, Jack also needs a photo-sharing site because he wants pupils to take and comment on each other's photos and to upload scans of old photographs that they may have in their families.

Jack sends an email to a primary school teachers' group, which he is a member of, to see if anyone can recommend an appropriate system. Two teachers reply, and both suggest that he use KidsTakePics, a photo-sharing site that allows teachers to check and moderate content. As KidsTakePics is not integrated with the iLearn authentication service, he sets up a teacher and a class account. He uses the iLearn setup service to add KidsTakePics to the services seen by the pupils in his class so that when they log in, they can immediately use the system to upload photos from their mobile devices and class computers.

**Figure 4.9** A user story for the iLearn system

The advantage of stories is that everyone can easily relate to them. We found this approach to be particularly useful to get information from a wider community than we could realistically interview. We made the stories available on a wiki and invited teachers and students from across the country to comment on them.

These high-level stories do not go into detail about a system, but they can be developed into more specific scenarios. Scenarios are descriptions of example user interaction sessions. I think that it is best to present scenarios in a structured way rather than as narrative text. User stories used in agile methods such as Extreme Programming, are actually narrative scenarios rather than general stories to help elicit requirements.

A scenario starts with an outline of the interaction. During the elicitation process, details are added to create a complete description of that interaction. At its most general, a scenario may include:

1. A description of what the system and users expect when the scenario starts.
2. A description of the normal flow of events in the scenario.
3. A description of what can go wrong and how resulting problems can be handled.
4. Information about other activities that might be going on at the same time.
5. A description of the system state when the scenario ends.

As an example of a scenario, Figure 4.10 describes what happens when a student uploads photos to the KidsTakePics system, as explained in Figure 4.9. The key difference between this system and other systems is that a teacher moderates the uploaded photos to check that they are suitable for sharing.

You can see this is a much more detailed description than the story in Figure 4.9, and so it can be used to propose requirements for the iLearn system. Like stories, scenarios can be used to facilitate discussions with stakeholders who sometimes may have different ways of achieving the same result.

### Uploading photos to KidsTakePics

**Initial assumption:** A user or a group of users have one or more digital photographs to be uploaded to the picture-sharing site. These photos are saved on either a tablet or a laptop computer. They have successfully logged on to KidsTakePics.

**Normal:** The user chooses to upload photos and is prompted to select the photos to be uploaded on the computer and to select the project name under which the photos will be stored. Users should also be given the option of inputting keywords that should be associated with each uploaded photo. Uploaded photos are named by creating a conjunction of the user name with the filename of the photo on the local computer.

On completion of the upload, the system automatically sends an email to the project moderator, asking them to check new content, and generates an on-screen message to the user that this checking has been done.

**What can go wrong:** No moderator is associated with the selected project. An email is automatically generated to the school administrator asking them to nominate a project moderator. Users should be informed of a possible delay in making their photos visible.

Photos with the same name have already been uploaded by the same user. The user should be asked if he or she wishes to re-upload the photos with the same name, rename the photos, or cancel the upload. If users choose to re-upload the photos, the originals are overwritten. If they choose to rename the photos, a new name is automatically generated by adding a number to the existing filename.

**Other activities:** The moderator may be logged on to the system and may approve photos as they are uploaded.

**System state on completion:** User is logged on. The selected photos have been uploaded and assigned a status "awaiting moderation." Photos are visible to the moderator and to the user who uploaded them.

**Figure 4.10** Scenario  
for uploading photos  
in KidsTakePics

## 4.4 Requirements specification

Requirements specification is the process of writing down the user and system requirements in a requirements document. Ideally, the user and system requirements should be clear, unambiguous, easy to understand, complete, and consistent. In practice, this is almost impossible to achieve. Stakeholders interpret the requirements in different ways, and there are often inherent conflicts and inconsistencies in the requirements.

User requirements are almost always written in natural language supplemented by appropriate diagrams and tables in the requirements document. System requirements may also be written in natural language, but other notations based on forms, graphical, or mathematical system models can also be used. Figure 4.11 summarizes possible notations for writing system requirements.

The user requirements for a system should describe the functional and nonfunctional requirements so that they are understandable by system users who don't have detailed technical knowledge. Ideally, they should specify only the external behavior of the system. The requirements document should not include details of the system architecture or design. Consequently, if you are writing user requirements, you should not use software jargon, structured notations, or formal notations. You should write user requirements in natural language, with simple tables, forms, and intuitive diagrams.

Notation	Description
Natural language sentences	The requirements are written using numbered sentences in natural language. Each sentence should express one requirement.
Structured natural language	The requirements are written in natural language on a standard form or template. Each field provides information about an aspect of the requirement.
Graphical notations	Graphical models, supplemented by text annotations, are used to define the functional requirements for the system. UML (unified modeling language) use case and sequence diagrams are commonly used.
Mathematical specifications	These notations are based on mathematical concepts such as finite-state machines or sets. Although these unambiguous specifications can reduce the ambiguity in a requirements document, most customers don't understand a formal specification. They cannot check that it represents what they want, and they are reluctant to accept it as a system contract. (I discuss this approach, in Chapter 10, which covers system dependability.)

**Figure 4.11** Notations for writing system requirements

System requirements are expanded versions of the user requirements that software engineers use as the starting point for the system design. They add detail and explain how the system should provide the user requirements. They may be used as part of the contract for the implementation of the system and should therefore be a complete and detailed specification of the whole system.

Ideally, the system requirements should only describe the external behavior of the system and its operational constraints. They should not be concerned with how the system should be designed or implemented. However, at the level of detail required to completely specify a complex software system, it is neither possible nor desirable to exclude all design information. There are several reasons for this:

1. You may have to design an initial architecture of the system to help structure the requirements specification. The system requirements are organized according to the different subsystems that make up the system. We did this when we were defining the requirements for the iLearn system, where we proposed the architecture shown in Figure 1.8.
2. In most cases, systems must interoperate with existing systems, which constrain the design and impose requirements on the new system.
3. The use of a specific architecture to satisfy non-functional requirements, such as N-version programming to achieve reliability, discussed in Chapter 11, may be necessary. An external regulator who needs to certify that the system is safe may specify that an architectural design that has already been certified should be used.

#### 4.4.1 Natural language specification

Natural language has been used to write requirements for software since the 1950s. It is expressive, intuitive, and universal. It is also potentially vague and ambiguous, and its interpretation depends on the background of the reader. As a result, there

3.2 The system shall measure the blood sugar and deliver insulin, if required, every 10 minutes. (*Changes in blood sugar are relatively slow, so more frequent measurement is unnecessary; less frequent measurement could lead to unnecessarily high sugar levels.*)

3.6 The system shall run a self-test routine every minute with the conditions to be tested and the associated actions defined in Table 1. (*A self-test routine can discover hardware and software problems and alert the user to the fact the normal operation may be impossible.*)

**Figure 4.12** Example requirements for the insulin pump software system

have been many proposals for alternative ways to write requirements. However, none of these proposals has been widely adopted, and natural language will continue to be the most widely used way of specifying system and software requirements.

To minimize misunderstandings when writing natural language requirements, I recommend that you follow these simple guidelines:

1. Invent a standard format and ensure that all requirement definitions adhere to that format. Standardizing the format makes omissions less likely and requirements easier to check. I suggest that, wherever possible, you should write the requirement in one or two sentences of natural language.
2. Use language consistently to distinguish between mandatory and desirable requirements. Mandatory requirements are requirements that the system must support and are usually written using “shall.” Desirable requirements are not essential and are written using “should.”
3. Use text highlighting (bold, italic, or color) to pick out key parts of the requirement.
4. Do not assume that readers understand technical, software engineering language. It is easy for words such as “architecture” and “module” to be misunderstood. Wherever possible, you should avoid the use of jargon, abbreviations, and acronyms.
5. Whenever possible, you should try to associate a rationale with each user requirement. The rationale should explain why the requirement has been included and who proposed the requirement (the requirement source), so that you know whom to consult if the requirement has to be changed. Requirements rationale is particularly useful when requirements are changed, as it may help decide what changes would be undesirable.

Figure 4.12 illustrates how these guidelines may be used. It includes two requirements for the embedded software for the automated insulin pump, introduced in Chapter 1. Other requirements for this embedded system are defined in the insulin pump requirements document, which can be downloaded from the book’s web pages.

#### 4.4.2 Structured specifications

Structured natural language is a way of writing system requirements where requirements are written in a standard way rather than as free-form text. This approach maintains most of the expressiveness and understandability of natural language but



### Problems with using natural language for requirements specification

The flexibility of natural language, which is so useful for specification, often causes problems. There is scope for writing unclear requirements, and readers (the designers) may misinterpret requirements because they have a different background to the user. It is easy to amalgamate several requirements into a single sentence, and structuring natural language requirements can be difficult.

<http://software-engineering-book.com/web/natural-language/>

ensures that some uniformity is imposed on the specification. Structured language notations use templates to specify system requirements. The specification may use programming language constructs to show alternatives and iteration, and may highlight key elements using shading or different fonts.

The Robertsons (Robertson and Robertson 2013), in their book on the VOLERE requirements engineering method, recommend that user requirements be initially written on cards, one requirement per card. They suggest a number of fields on each card, such as the requirements rationale, the dependencies on other requirements, the source of the requirements, and supporting materials. This is similar to the approach used in the example of a structured specification shown in Figure 4.13.

To use a structured approach to specifying system requirements, you define one or more standard templates for requirements and represent these templates as structured forms. The specification may be structured around the objects manipulated by the system, the functions performed by the system, or the events processed by the system. An example of a form-based specification, in this case, one that defines how to calculate the dose of insulin to be delivered when the blood sugar is within a safe band, is shown in Figure 4.13.

When a standard format is used for specifying functional requirements, the following information should be included:

1. A description of the function or entity being specified.
2. A description of its inputs and the origin of these inputs.
3. A description of its outputs and the destination of these outputs.
4. Information about the information needed for the computation or other entities in the system that are required (the “requires” part).
5. A description of the action to be taken.
6. If a functional approach is used, a precondition setting out what must be true before the function is called, and a postcondition specifying what is true after the function is called.
7. A description of the side effects (if any) of the operation.

Using structured specifications removes some of the problems of natural language specification. Variability in the specification is reduced, and requirements are organized

<b>Insulin Pump/Control Software/SRS/3.3.2</b>	
<b>Function</b>	Compute insulin dose: Safe sugar level.
<b>Description</b>	Computes the dose of insulin to be delivered when the current measured sugar level is in the safe zone between 3 and 7 units.
<b>Inputs</b>	Current sugar reading ( $r_2$ ), the previous two readings ( $r_0$ and $r_1$ ).
<b>Source</b>	Current sugar reading from sensor. Other readings from memory.
<b>Outputs</b>	CompDose—the dose in insulin to be delivered.
<b>Destination</b>	Main control loop.
<b>Action:</b>	CompDose is zero if the sugar level is stable or falling or if the level is increasing but the rate of increase is decreasing. If the level is increasing and the rate of increase is increasing, then CompDose is computed by dividing the difference between the current sugar level and the previous level by 4 and rounding the result. If the result, is rounded to zero then CompDose is set to the minimum dose that can be delivered. (see Figure 4.14)
<b>Requires</b>	Two previous readings so that the rate of change of sugar level can be computed.
<b>Precondition</b>	The insulin reservoir contains at least the maximum allowed single dose of insulin.
<b>Postcondition</b>	$r_0$ is replaced by $r_1$ then $r_1$ is replaced by $r_2$ .
<b>Side effects</b>	None.

**Figure 4.13** The structured specification of a requirement for an insulin pump

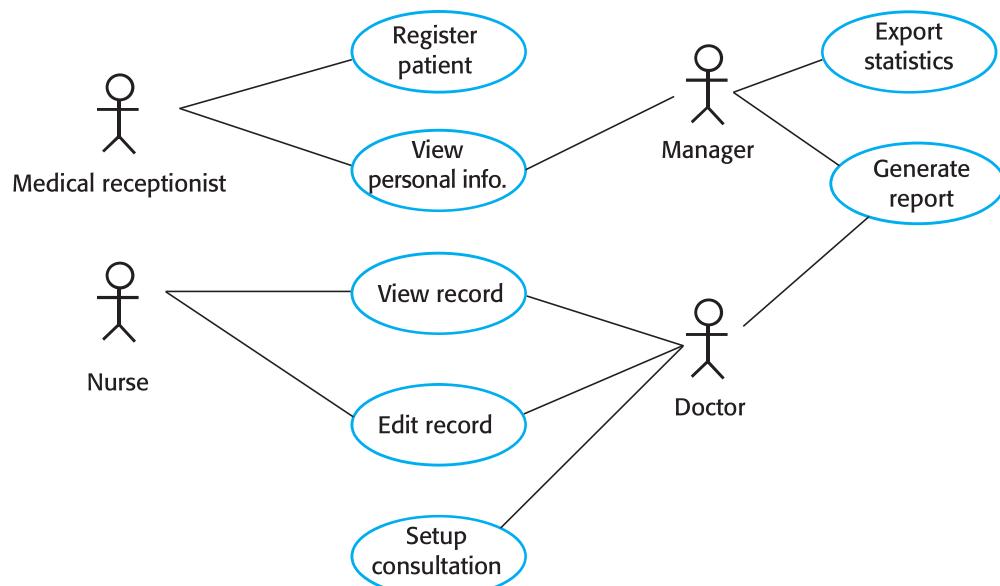
more effectively. However, it is still sometimes difficult to write requirements in a clear and unambiguous way, particularly when complex computations (e.g., how to calculate the insulin dose) are to be specified.

To address this problem, you can add extra information to natural language requirements, for example, by using tables or graphical models of the system. These can show how computations proceed, how the system state changes, how users interact with the system, and how sequences of actions are performed.

Tables are particularly useful when there are a number of possible alternative situations and you need to describe the actions to be taken for each of these. The insulin pump bases its computations of the insulin requirement on the rate of change of blood sugar levels. The rates of change are computed using the current and previous readings. Figure 4.14 is a tabular description of how the rate of change of blood sugar is used to calculate the amount of insulin to be delivered.

**Figure 4.14** The tabular specification of computation in an insulin pump

Condition	Action
Sugar level falling ( $r_2 < r_1$ )	CompDose = 0
Sugar level stable ( $r_2 = r_1$ )	CompDose = 0
Sugar level increasing and rate of increase decreasing ( $(r_2 - r_1) < (r_1 - r_0)$ )	CompDose = 0
Sugar level increasing and rate of increase stable or increasing $r_2 > r_1 \& ((r_2 - r_1) \geq (r_1 - r_0))$	CompDose = round $((r_2 - r_1)/4)$ If rounded result = 0 then CompDose = MinimumDose



**Figure 4.15** Use cases for the Mentcare system

### 4.4.3 Use cases

Use cases are a way of describing interactions between users and a system using a graphical model and structured text. They were first introduced in the Objectory method (Jacobsen et al. 1993) and have now become a fundamental feature of the Unified Modeling Language (UML). In their simplest form, a use case identifies the actors involved in an interaction and names the type of interaction. You then add additional information describing the interaction with the system. The additional information may be a textual description or one or more graphical models such as the UML sequence or state charts (see Chapter 5).

Use cases are documented using a high-level use case diagram. The set of use cases represents all of the possible interactions that will be described in the system requirements. Actors in the process, who may be human or other systems, are represented as stick figures. Each class of interaction is represented as a named ellipse. Lines link the actors with the interaction. Optionally, arrowheads may be added to lines to show how the interaction is initiated. This is illustrated in Figure 4.15, which shows some of the use cases for the Mentcare system.

Use cases identify the individual interactions between the system and its users or other systems. Each use case should be documented with a textual description. These can then be linked to other models in the UML that will develop the scenario in more detail. For example, a brief description of the Setup Consultation use case from Figure 4.15 might be:

*Setup consultation allows two or more doctors, working in different offices, to view the same patient record at the same time. One doctor initiates the consultation by choosing the people involved from a dropdown menu of doctors who are online. The patient record is then displayed on their screens, but only the initiating doctor can edit the record. In addition, a text chat window is created*

*to help coordinate actions. It is assumed that a phone call for voice communication can be separately arranged.*

The UML is a standard for object-oriented modeling, so use cases and use case-based elicitation are used in the requirements engineering process. However, my experience with use cases is that they are too fine-grained to be useful in discussing requirements. Stakeholders don't understand the term *use case*; they don't find the graphical model to be useful, and they are often not interested in a detailed description of each and every system interaction. Consequently, I find use cases to be more helpful in systems design than in requirements engineering. I discuss use cases further in Chapter 5, which shows how they are used alongside other system models to document a system design.

Some people think that each use case is a single, low-level interaction scenario. Others, such as Stevens and Pooley (Stevens and Pooley 2006), suggest that each use case includes a set of related, low-level scenarios. Each of these scenarios is a single thread through the use case. Therefore, there would be a scenario for the normal interaction plus scenarios for each possible exception. In practice, you can use them in either way.

#### 4.4.4 The software requirements document

---

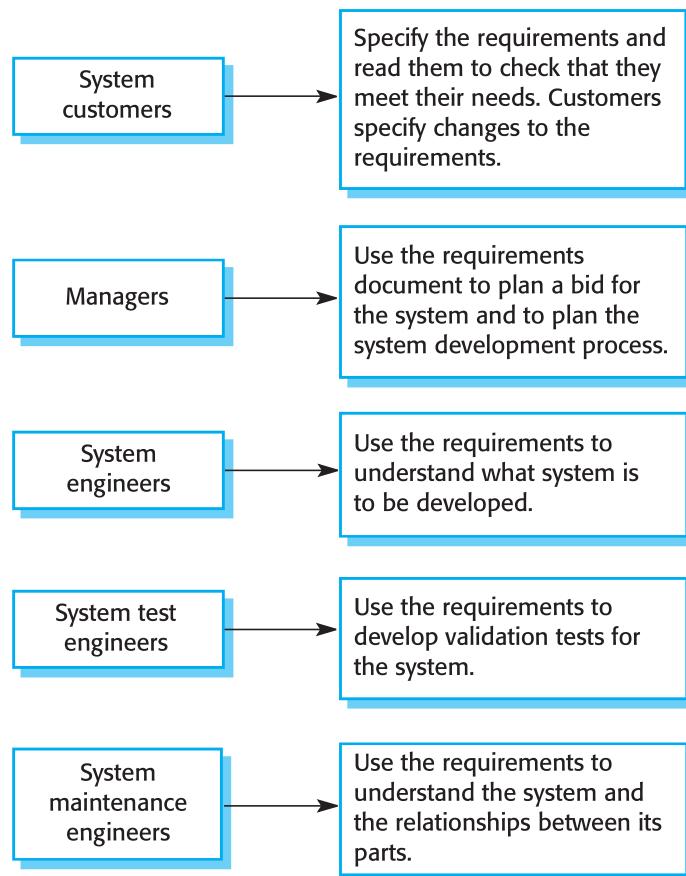
The software requirements document (sometimes called the software requirements specification or SRS) is an official statement of what the system developers should implement. It may include both the user requirements for a system and a detailed specification of the system requirements. Sometimes the user and system requirements are integrated into a single description. In other cases, the user requirements are described in an introductory chapter in the system requirements specification.

Requirements documents are essential when systems are outsourced for development, when different teams develop different parts of the system, and when a detailed analysis of the requirements is mandatory. In other circumstances, such as software product or business system development, a detailed requirements document may not be needed.

Agile methods argue that requirements change so rapidly that a requirements document is out of date as soon as it is written, so the effort is largely wasted. Rather than a formal document, agile approaches often collect user requirements incrementally and write these on cards or whiteboards as short user stories. The user then prioritizes these stories for implementation in the next increment of the system.

For business systems where requirements are unstable, I think that this approach is a good one. However, I think that it is still useful to write a short supporting document that defines the business and dependability requirements for the system; it is easy to forget the requirements that apply to the system as a whole when focusing on the functional requirements for the next system release.

The requirements document has a diverse set of users, ranging from the senior management of the organization that is paying for the system to the engineers responsible for developing the software. Figure 4.16 shows possible users of the document and how they use it.



**Figure 4.16** Users of a requirements document

The diversity of possible users means that the requirements document has to be a compromise. It has to describe the requirements for customers, define the requirements in precise detail for developers and testers, as well as include information about future system evolution. Information on anticipated changes helps system designers to avoid restrictive design decisions and maintenance engineers to adapt the system to new requirements.

The level of detail that you should include in a requirements document depends on the type of system that is being developed and the development process used. Critical systems need detailed requirements because safety and security have to be analyzed in detail to find possible requirements errors. When the system is to be developed by a separate company (e.g., through outsourcing), the system specifications need to be detailed and precise. If an in-house, iterative development process is used, the requirements document can be less detailed. Details can be added to the requirements and ambiguities resolved during development of the system.

Figure 4.17 shows one possible organization for a requirements document that is based on an IEEE standard for requirements documents (IEEE 1998). This standard is a generic one that can be adapted to specific uses. In this case, the standard has been extended to include information about predicted system evolution. This information helps the maintainers of the system and allows designers to include support for future system features.

Chapter	Description
Preface	This defines the expected readership of the document and describe its version history, including a rationale for the creation of a new version and a summary of the changes made in each version.
Introduction	This describes the need for the system. It should briefly describe the system's functions and explain how it will work with other systems. It should also describe how the system fits into the overall business or strategic objectives of the organization commissioning the software.
Glossary	This defines the technical terms used in the document. You should not make assumptions about the experience or expertise of the reader.
User requirements definition	Here, you describe the services provided for the user. The nonfunctional system requirements should also be described in this section. This description may use natural language, diagrams, or other notations that are understandable to customers. Product and process standards that must be followed should be specified.
System architecture	This chapter presents a high-level overview of the anticipated system architecture, showing the distribution of functions across system modules. Architectural components that are reused should be highlighted.
System requirements specification	This describes the functional and nonfunctional requirements in more detail. If necessary, further detail may also be added to the nonfunctional requirements. Interfaces to other systems may be defined.
System models	This chapter includes graphical system models showing the relationships between the system components and the system and its environment. Examples of possible models are object models, data-flow models, or semantic data models.
System evolution	This describes the fundamental assumptions on which the system is based, and any anticipated changes due to hardware evolution, changing user needs, and so on. This section is useful for system designers as it may help them avoid design decisions that would constrain likely future changes to the system.
Appendices	These provide detailed, specific information that is related to the application being developed—for example, hardware and database descriptions. Hardware requirements define the minimal and optimal configurations for the system. Database requirements define the logical organization of the data used by the system and the relationships between data.
Index	Several indexes to the document may be included. As well as a normal alphabetic index, there may be an index of diagrams, an index of functions, and so on.

**Figure 4.17** The structure of a requirements document

Naturally, the information included in a requirements document depends on the type of software being developed and the approach to development that is to be used. A requirements document with a structure like that shown in Figure 4.17 might be produced for a complex engineering system that includes hardware and software developed by different companies. The requirements document is likely to be long and detailed. It is therefore important that a comprehensive table of contents and document index be included so that readers can easily find the information they need.

By contrast, the requirements document for an in-house software product will leave out many of detailed chapters suggested above. The focus will be on defining the user requirements and high-level, nonfunctional system requirements. The system designers and programmers use their judgment to decide how to meet the outline user requirements for the system.



### Requirements document standards

A number of large organizations, such as the U.S. Department of Defense and the IEEE, have defined standards for requirements documents. These are usually very generic but are nevertheless useful as a basis for developing more detailed organizational standards. The U.S. Institute of Electrical and Electronic Engineers (IEEE) is one of the best-known standards providers, and they have developed a standard for the structure of requirements documents. This standard is most appropriate for systems such as military command and control systems that have a long lifetime and are usually developed by a group of organizations.

<http://software-engineering-book.com/web/requirements-standard/>

## 4.5 Requirements validation

Requirements validation is the process of checking that requirements define the system that the customer really wants. It overlaps with elicitation and analysis, as it is concerned with finding problems with the requirements. Requirements validation is critically important because errors in a requirements document can lead to extensive rework costs when these problems are discovered during development or after the system is in service.

The cost of fixing a requirements problem by making a system change is usually much greater than repairing design or coding errors. A change to the requirements usually means that the system design and implementation must also be changed. Furthermore, the system must then be retested.

During the requirements validation process, different types of checks should be carried out on the requirements in the requirements document. These checks include:

1. *Validity checks* These check that the requirements reflect the real needs of system users. Because of changing circumstances, the user requirements may have changed since they were originally elicited.
2. *Consistency checks* Requirements in the document should not conflict. That is, there should not be contradictory constraints or different descriptions of the same system function.
3. *Completeness checks* The requirements document should include requirements that define all functions and the constraints intended by the system user.
4. *Realism checks* By using knowledge of existing technologies, the requirements should be checked to ensure that they can be implemented within the proposed budget for the system. These checks should also take account of the budget and schedule for the system development.
5. *Verifiability* To reduce the potential for dispute between customer and contractor, system requirements should always be written so that they are verifiable. This means that you should be able to write a set of tests that can demonstrate that the delivered system meets each specified requirement.



### Requirements reviews

A requirements review is a process in which a group of people from the system customer and the system developer read the requirements document in detail and check for errors, anomalies, and inconsistencies. Once these have been detected and recorded, it is then up to the customer and the developer to negotiate how the identified problems should be solved.

<http://software-engineering-book.com/web/requirements-reviews/>

A number of requirements validation techniques can be used individually or in conjunction with one another:

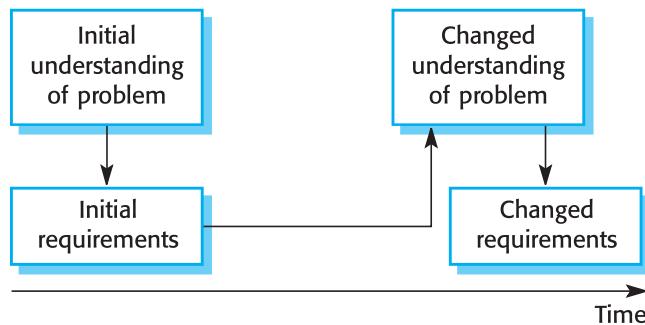
1. *Requirements reviews* The requirements are analyzed systematically by a team of reviewers who check for errors and inconsistencies.
2. *Prototyping* This involves developing an executable model of a system and using this with end-users and customers to see if it meets their needs and expectations. Stakeholders experiment with the system and feed back requirements changes to the development team.
3. *Test-case generation* Requirements should be testable. If the tests for the requirements are devised as part of the validation process, this often reveals requirements problems. If a test is difficult or impossible to design, this usually means that the requirements will be difficult to implement and should be reconsidered. Developing tests from the user requirements before any code is written is an integral part of test-driven development.

You should not underestimate the problems involved in requirements validation. Ultimately, it is difficult to show that a set of requirements does in fact meet a user's needs. Users need to picture the system in operation and imagine how that system would fit into their work. It is hard even for skilled computer professionals to perform this type of abstract analysis and harder still for system users.

As a result, you rarely find all requirements problems during the requirements validation process. Inevitably, further requirements changes will be needed to correct omissions and misunderstandings after agreement has been reached on the requirements document.

## 4.6 Requirements change

The requirements for large software systems are always changing. One reason for the frequent changes is that these systems are often developed to address “wicked” problems—problems that cannot be completely defined (Rittel and Webber 1973). Because the problem cannot be fully defined, the software requirements are bound to



**Figure 4.18**  
Requirements evolution

be incomplete. During the software development process, the stakeholders' understanding of the problem is constantly changing (Figure 4.18). The system requirements must then evolve to reflect this changed problem understanding.

Once a system has been installed and is regularly used, new requirements inevitably emerge. This is partly a consequence of errors and omissions in the original requirements that have to be corrected. However, most changes to system requirements arise because of changes to the business environment of the system:

1. The business and technical environment of the system always changes after installation. New hardware may be introduced and existing hardware updated. It may be necessary to interface the system with other systems. Business priorities may change (with consequent changes in the system support required), and new legislation and regulations may be introduced that require system compliance.
  2. The people who pay for a system and the users of that system are rarely the same people. System customers impose requirements because of organizational and budgetary constraints. These may conflict with end-user requirements, and, after delivery, new features may have to be added for user support if the system is to meet its goals.
  3. Large systems usually have a diverse stakeholder community, with stakeholders having different requirements. Their priorities may be conflicting or contradictory. The final system requirements are inevitably a compromise, and some stakeholders have to be given priority. With experience, it is often discovered that the balance of support given to different stakeholders has to be changed and the requirements re-prioritized.

As requirements are evolving, you need to keep track of individual requirements and maintain links between dependent requirements so that you can assess the impact of requirements changes. You therefore need a formal process for making change proposals and linking these to system requirements. This process of “requirements management” should start as soon as a draft version of the requirements document is available.

Agile development processes have been designed to cope with requirements that change during the development process. In these processes, when a user proposes a requirements change, this change does not go through a formal change management



### Enduring and volatile requirements

Some requirements are more susceptible to change than others. Enduring requirements are the requirements that are associated with the core, slow-to-change activities of an organization. Enduring requirements are associated with fundamental work activities. Volatile requirements are more likely to change. They are usually associated with supporting activities that reflect how the organization does its work rather than the work itself.

<http://software-engineering-book.com/web/changing-requirements/>

process. Rather, the user has to prioritize that change and, if it is high priority, decide what system features that were planned for the next iteration should be dropped for the change to be implemented.

The problem with this approach is that users are not necessarily the best people to decide on whether or not a requirements change is cost-effective. In systems with multiple stakeholders, changes will benefit some stakeholders and not others. It is often better for an independent authority, who can balance the needs of all stakeholders, to decide on the changes that should be accepted.

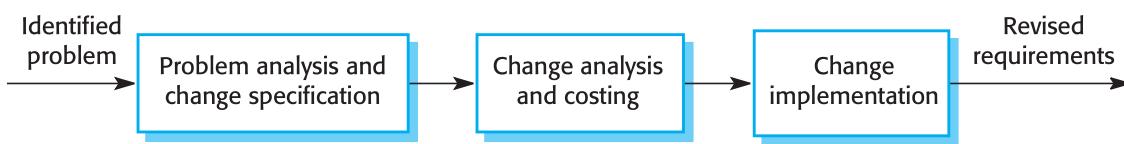
#### 4.6.1 Requirements management planning

Requirements management planning is concerned with establishing how a set of evolving requirements will be managed. During the planning stage, you have to decide on a number of issues:

1. *Requirements identification* Each requirement must be uniquely identified so that it can be cross-referenced with other requirements and used in traceability assessments.
2. *A change management process* This is the set of activities that assess the impact and cost of changes. I discuss this process in more detail in the following section.
3. *Traceability policies* These policies define the relationships between each requirement and between the requirements and the system design that should be recorded. The traceability policy should also define how these records should be maintained.
4. *Tool support* Requirements management involves the processing of large amounts of information about the requirements. Tools that may be used range from specialist requirements management systems to shared spreadsheets and simple database systems.

Requirements management needs automated support, and the software tools for this should be chosen during the planning phase. You need tool support for:

1. *Requirements storage* The requirements should be maintained in a secure, managed data store that is accessible to everyone involved in the requirements engineering process.



**Figure 4.19**  
Requirements change management

2. *Change management* The process of change management (Figure 4.19) is simplified if active tool support is available. Tools can keep track of suggested changes and responses to these suggestions.
3. *Traceability management* As discussed above, tool support for traceability allows related requirements to be discovered. Some tools are available which use natural language processing techniques to help discover possible relationships between requirements.

For small systems, you do not need to use specialized requirements management tools. Requirements management can be supported using shared web documents, spreadsheets, and databases. However, for larger systems, more specialized tool support, using systems such as DOORS (IBM 2013), makes it much easier to keep track of a large number of changing requirements.

#### 4.6.2 Requirements change management

Requirements change management (Figure 4.19) should be applied to all proposed changes to a system's requirements after the requirements document has been approved. Change management is essential because you need to decide if the benefits of implementing new requirements are justified by the costs of implementation. The advantage of using a formal process for change management is that all change proposals are treated consistently and changes to the requirements document are made in a controlled way.

There are three principal stages to a change management process:

1. *Problem analysis and change specification* The process starts with an identified requirements problem or, sometimes, with a specific change proposal. During this stage, the problem or the change proposal is analyzed to check that it is valid. This analysis is fed back to the change requestor who may respond with a more specific requirements change proposal, or decide to withdraw the request.
2. *Change analysis and costing* The effect of the proposed change is assessed using traceability information and general knowledge of the system requirements. The cost of making the change is estimated in terms of modifications to the requirements document and, if appropriate, to the system design and implementation. Once this analysis is completed, a decision is made as to whether or not to proceed with the requirements change.



### Requirements traceability

You need to keep track of the relationships between requirements, their sources, and the system design so that you can analyze the reasons for proposed changes and the impact that these changes are likely to have on other parts of the system. You need to be able to trace how a change ripples its way through the system. Why?

<http://software-engineering-book.com/web/traceability/>

3. *Change implementation* The requirements document and, where necessary, the system design and implementation, are modified. You should organize the requirements document so that you can make changes to it without extensive rewriting or reorganization. As with programs, changeability in documents is achieved by minimizing external references and making the document sections as modular as possible. Thus, individual sections can be changed and replaced without affecting other parts of the document.

If a new requirement has to be urgently implemented, there is always a temptation to change the system and then retrospectively modify the requirements document. This almost inevitably leads to the requirements specification and the system implementation getting out of step. Once system changes have been made, it is easy to forget to include these changes in the requirements document. In some circumstances, emergency changes to a system have to be made. In those cases, it is important that you update the requirements document as soon as possible in order to include the revised requirements.

## KEY POINTS

- Requirements for a software system set out what the system should do and define constraints on its operation and implementation.
- Functional requirements are statements of the services that the system must provide or are descriptions of how some computations must be carried out.
- Non-functional requirements often constrain the system being developed and the development process being used. These might be product requirements, organizational requirements, or external requirements. They often relate to the emergent properties of the system and therefore apply to the system as a whole.
- The requirements engineering process includes requirements elicitation, requirements specification, requirements validation, and requirements management.
- Requirements elicitation is an iterative process that can be represented as a spiral of activities—requirements discovery, requirements classification and organization, requirements negotiation, and requirements documentation.

- Requirements specification is the process of formally documenting the user and system requirements and creating a software requirements document.
- The software requirements document is an agreed statement of the system requirements. It should be organized so that both system customers and software developers can use it.
- Requirements validation is the process of checking the requirements for validity, consistency, completeness, realism, and verifiability.
- Business, organizational, and technical changes inevitably lead to changes to the requirements for a software system. Requirements management is the process of managing and controlling these changes.

## FURTHER READING

“Integrated Requirements Engineering: A Tutorial.” This is a tutorial paper that discusses requirements engineering activities and how these can be adapted to fit with modern software engineering practice. (I. Sommerville, *IEEE Software*, 22(1), January–February 2005) <http://dx.doi.org/10.1109/MS.2005.13>.

“Research Directions in Requirements Engineering.” This is a good survey of requirements engineering research that highlights future research challenges in the area to address issues such as scale and agility. (B. H. C. Cheng and J. M. Atlee, *Proc. Conf. on Future of Software Engineering*, IEEE Computer Society, 2007) <http://dx.doi.org/10.1109/FOSE.2007.17>.

*Mastering the Requirements Process, 3rd ed.* A well-written, easy-to-read book that is based on a particular method (VOLERE) but that also includes lots of good general advice about requirements engineering. (S. Robertson and J. Robertson, 2013, Addison-Wesley).

## WEBSITE

PowerPoint slides for this chapter:

[www.pearsonglobaleditions.com/Sommerville](http://www.pearsonglobaleditions.com/Sommerville)

Links to supporting videos:

<http://software-engineering-book.com/videos/requirements-and-design/>

Requirements document for the insulin pump:

<http://software-engineering-book.com/case-studies/insulin-pump/>

Mentcare system requirements information:

<http://software-engineering-book.com/case-studies/mentcare-system/>

## EXERCISES

- 4.1.** Identify and briefly describe four types of requirements that may be defined for a computer-based system.
- 4.2.** Discover ambiguities or omissions in the following statement of the requirements for part of a drone system intended for search and recovery:

*The drone, a quad chopper, will be very useful in search and recovery operations, especially in remote areas or in extreme weather conditions. It will click high-resolution images. It will fly according to a path preset by a ground operator, but will be able to avoid obstacles on its own, returning to its original path whenever possible. The drone will also be able to identify various objects and match them to the target it is looking for.*

- 4.3.** Rewrite the above description using the structured approach described in this chapter. Resolve the identified ambiguities in a sensible way.
- 4.4.** Write a set of non-functional requirements for the drone system, setting out its expected safety and response time.
- 4.5.** Using the technique suggested here, where natural language descriptions are presented in a standard format, write plausible user requirements for the following functions:

An unattended petrol (gas) pump system that includes a credit card reader. The customer swipes the card through the reader, then specifies the amount of fuel required. The fuel is delivered and the customer's account debited.

The cash-dispensing function in a bank ATM.

In an Internet banking system, a facility that allows customers to transfer funds from one account held with the bank to another account with the same bank.

- 4.6.** Suggest how an engineer responsible for drawing up a system requirements specification might keep track of the relationships between functional and non-functional requirements.
- 4.7.** Using your knowledge of how an ATM is used, develop a set of use cases that could serve as a basis for understanding the requirements for an ATM system.
- 4.8.** To minimize mistakes during a requirements review, an organization decides to allocate two scribes to document the review session. Explain how this can be done.
- 4.9.** When emergency changes have to be made to systems, the system software may have to be modified before changes to the requirements have been approved. Suggest a model of a process for making these modifications that will ensure that the requirements document and the system implementation do not become inconsistent.
- 4.10.** You have taken a job with a software user who has contracted your previous employer to develop a system for them. You discover that your company's interpretation of the requirements is different from the interpretation taken by your previous employer. Discuss what you

should do in such a situation. You know that the costs to your current employer will increase if the ambiguities are not resolved. However, you also have a responsibility of confidentiality to your previous employer.

## REFERENCES

- Crabtree, A. 2003. *Designing Collaborative Systems: A Practical Guide to Ethnography*. London: Springer-Verlag.
- Davis, A. M. 1993. *Software Requirements: Objects, Functions and States*. Englewood Cliffs, NJ: Prentice-Hall.
- IBM. 2013. “Rational Doors Next Generation: Requirements Engineering for Complex Systems.” <https://jazz.net/products/rational-doors-next-generation/>
- IEEE. 1998. “IEEE Recommended Practice for Software Requirements Specifications.” In *IEEE Software Engineering Standards Collection*. Los Alamitos, CA: IEEE Computer Society Press.
- Jacobsen, I., M. Christerson, P. Jonsson, and G. Overgaard. 1993. *Object-Oriented Software Engineering*. Wokingham, UK: Addison-Wesley.
- Martin, D., and I. Sommerville. 2004. “Patterns of Cooperative Interaction: Linking Ethnomethodology and Design.” *ACM Transactions on Computer-Human Interaction* 11 (1) (March 1): 59–89. doi:10.1145/972648.972651.
- Rittel, H., and M. Webber. 1973. “Dilemmas in a General Theory of Planning.” *Policy Sciences* 4: 155–169. doi:10.1007/BF01405730.
- Robertson, S., and J. Robertson. 2013. *Mastering the Requirements Process*, 3rd ed. Boston: Addison-Wesley.
- Sommerville, I., T. Rodden, P. Sawyer, R. Bentley, and M. Twidale. 1993. “Integrating Ethnography into the Requirements Engineering Process.” In *RE’93*, 165–173. San Diego, CA: IEEE Computer Society Press. doi:10.1109/ISRE.1993.324821.
- Stevens, P., and R. Pooley. 2006. *Using UML: Software Engineering with Objects and Components*, 2nd ed. Harlow, UK: Addison-Wesley.
- Suchman, L. 1983. “Office Procedures as Practical Action: Models of Work and System Design.” *ACM Transactions on Office Information Systems* 1 (3): 320–328. doi:10.1145/357442.357445.
- Viller, S., and I. Sommerville. 2000. “Ethnographically Informed Analysis for Software Engineers.” *Int. J. of Human-Computer Studies* 53 (1): 169–196. doi:10.1006/ijhc.2000.0370.



# 5

---

# System modeling

## Objectives

The aim of this chapter is to introduce system models that may be developed as part of requirements engineering and system design processes. When you have read the chapter, you will:

- understand how graphical models can be used to represent software systems and why several types of model are needed to fully represent a system;
- understand the fundamental system modeling perspectives of context, interaction, structure, and behavior;
- understand the principal diagram types in the Unified Modeling Language (UML) and how these diagrams may be used in system modeling;
- have been introduced to model-driven engineering, where an executable system is automatically generated from structural and behavioral models.

## Contents

- 5.1 Context models**
- 5.2 Interaction models**
- 5.3 Structural models**
- 5.4 Behavioral models**
- 5.5 Model-driven engineering**

**IEEE Std 830-1998**  
(Revision of  
IEEE Std 830-1993)

# **IEEE Recommended Practice for Software Requirements Specifications**

Sponsor

**Software Engineering Standards Committee  
of the  
IEEE Computer Society**

Approved 25 June 1998

**IEEE-SA Standards Board**

**Abstract:** The content and qualities of a good software requirements specification (SRS) are described and several sample SRS outlines are presented. This recommended practice is aimed at specifying requirements of software to be developed but also can be applied to assist in the selection of in-house and commercial software products. Guidelines for compliance with IEEE/EIA 12207.1-1997 are also provided.

**Keywords:** contract, customer, prototyping, software requirements specification, supplier, system requirements specifications

---

The Institute of Electrical and Electronics Engineers, Inc.  
345 East 47th Street, New York, NY 10017-2394, USA

Copyright © 1998 by the Institute of Electrical and Electronics Engineers, Inc.  
All rights reserved. Published 1998. Printed in the United States of America.

ISBN 0-7381-0332-2

No part of this publication may be reproduced in any form, in an electronic retrieval system or otherwise, without the prior written permission of the publisher.

**IEEE Standards** documents are developed within the IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Association (IEEE-SA) Standards Board. Members of the committees serve voluntarily and without compensation. They are not necessarily members of the Institute. The standards developed within IEEE represent a consensus of the broad expertise on the subject within the Institute as well as those activities outside of IEEE that have expressed an interest in participating in the development of the standard.

Use of an IEEE Standard is wholly voluntary. The existence of an IEEE Standard does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of the IEEE Standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change brought about through developments in the state of the art and comments received from users of the standard. Every IEEE Standard is subjected to review at least every five years for revision or reaffirmation. When a document is more than five years old and has not been reaffirmed, it is reasonable to conclude that its contents, although still of some value, do not wholly reflect the present state of the art. Users are cautioned to check to determine that they have the latest edition of any IEEE Standard.

Comments for revision of IEEE Standards are welcome from any interested party, regardless of membership affiliation with IEEE. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments.

**Interpretations:** Occasionally questions may arise regarding the meaning of portions of standards as they relate to specific applications. When the need for interpretations is brought to the attention of IEEE, the Institute will initiate action to prepare appropriate responses. Since IEEE Standards represent a consensus of all concerned interests, it is important to ensure that any interpretation has also received the concurrence of a balance of interests. For this reason, IEEE and the members of its societies and Standards Coordinating Committees are not able to provide an instant response to interpretation requests except in those cases where the matter has previously received formal consideration.

Comments on standards and requests for interpretations should be addressed to:

Secretary, IEEE-SA Standards Board  
445 Hoes Lane  
P.O. Box 1331  
Piscataway, NJ 08855-1331  
USA

Note: Attention is called to the possibility that implementation of this standard may require use of subject matter covered by patent rights. By publication of this standard, no position is taken with respect to the existence or validity of any patent rights in connection therewith. The IEEE shall not be responsible for identifying patents for which a license may be required by an IEEE standard or for conducting inquiries into the legal validity or scope of those patents that are brought to its attention.

Authorization to photocopy portions of any individual standard for internal or personal use is granted by the Institute of Electrical and Electronics Engineers, Inc., provided that the appropriate fee is paid to Copyright Clearance Center. To arrange for payment of licensing fee, please contact Copyright Clearance Center, Customer Service, 222 Rosewood Drive, Danvers, MA 01923 USA; (978) 750-8400. Permission to photocopy portions of any individual standard for educational classroom use can also be obtained through the Copyright Clearance Center.

# Introduction

(This introduction is not a part of IEEE Std 830-1998, IEEE Recommended Practice for Software Requirements Specifications.)

This recommended practice describes recommended approaches for the specification of software requirements. It is based on a model in which the result of the software requirements specification process is an unambiguous and complete specification document. It should help

- a) Software customers to accurately describe what they wish to obtain;
- b) Software suppliers to understand exactly what the customer wants;
- c) Individuals to accomplish the following goals:
  - 1) Develop a standard software requirements specification (SRS) outline for their own organizations;
  - 2) Define the format and content of their specific software requirements specifications;
  - 3) Develop additional local supporting items such as an SRS quality checklist, or an SRS writer's handbook.

To the customers, suppliers, and other individuals, a good SRS should provide several specific benefits, such as the following:

- *Establish the basis for agreement between the customers and the suppliers on what the software product is to do.* The complete description of the functions to be performed by the software specified in the SRS will assist the potential users to determine if the software specified meets their needs or how the software must be modified to meet their needs.
- *Reduce the development effort.* The preparation of the SRS forces the various concerned groups in the customer's organization to consider rigorously all of the requirements before design begins and reduces later redesign, recoding, and retesting. Careful review of the requirements in the SRS can reveal omissions, misunderstandings, and inconsistencies early in the development cycle when these problems are easier to correct.
- *Provide a basis for estimating costs and schedules.* The description of the product to be developed as given in the SRS is a realistic basis for estimating project costs and can be used to obtain approval for bids or price estimates.
- *Provide a baseline for validation and verification.* Organizations can develop their validation and verification plans much more productively from a good SRS. As a part of the development contract, the SRS provides a baseline against which compliance can be measured.
- *Facilitate transfer.* The SRS makes it easier to transfer the software product to new users or new machines. Customers thus find it easier to transfer the software to other parts of their organization, and suppliers find it easier to transfer it to new customers.
- *Serve as a basis for enhancement.* Because the SRS discusses the product but not the project that developed it, the SRS serves as a basis for later enhancement of the finished product. The SRS may need to be altered, but it does provide a foundation for continued production evaluation.

The readers of this document are referred to Annex B for guidelines for using this recommended practice to meet the requirements of IEEE/EIA 12207.1-1997, IEEE/EIA Guide—Industry Implementation of ISO/IEC 12207: 1995, Standard for Information Technology—Software life cycle processes—Life cycle data.

## Participants

This recommended practice was prepared by the Life Cycle Data Harmonization Working Group of the Software Engineering Standards Committee of the IEEE Computer Society. At the time this recommended practice was approved, the working group consisted of the following members:

### **Leonard L. Tripp, Chair**

Edward Byrne  
Paul R. Croll  
Perry DeWeese  
Robin Fralick  
Marilyn Ginsberg-Finner  
John Harauz  
Mark Henley

Dennis Lawrence  
David Maibor  
Ray Milovanovic  
James Moore  
Timothy Niesen  
Dennis Rilling

Terry Rout  
Richard Schmidt  
Norman F. Schneidewind  
David Schultz  
Basil Sherlund  
Peter Voldner  
Ronald Wade

The following persons were on the balloting committee:

Syed Ali	David A. Gustafson	Indradeb P. Pal
Theodore K. Atchinson	Jon D. Hagar	Alex Polack
Mikhail Auguston	John Harauz	Peter T. Poon
Robert E. Barry	Robert T. Harley	Lawrence S. Przybylski
Leo Beltracchi	Herbert Hecht	Kenneth R. Ptack
H. Ronald Berlack	William Hefley	Annette D. Reilly
Richard E. Biehl	Manfred Hein	Dennis Rilling
Michael A. Blackledge	Mark Heinrich	Andrew P. Sage
Sandro Bologna	Mark Henley	Helmut Sandmayer
Juris Borzovs	Debra Herrmann	Stephen R. Schach
Kathleen L. Briggs	John W. Horch	Hans Schaefer
M. Scott Buck	Jerry Huller	Norman Schneidewind
Michael Caldwell	Peter L. Hung	David J. Schultz
James E. Cardow	George Jackelen	Lisa A. Selmon
Enrico A. Carrara	Frank V. Jorgensen	Robert W. Shillato
Lawrence Catchpole	William S. Junk	David M. Siebert
Keith Chan	George X. Kambic	Carl A. Singer
Antonio M. Cicu	Richard Karcich	James M. Sivak
Theo Clarke	Ron S. Kenett	Richard S. Sky
Sylvain Clermont	Judith S. Kerner	Nancy M. Smith
Rosemary Coleman	Robert J. Kierzyk	Melford E. Smyre
Virgil Lee Cooper	Dwayne L. Knirk	Harry M. Sneed
W. W. Geoff Cozens	Shaye Koenig	Alfred R. Sorkowitz
Paul R. Croll	Thomas M. Kurihara	Donald W. Sova
Gregory T. Daich	John B. Lane	Luca Spotorno
Geoffrey Darnton	J. Dennis Lawrence	Julia Stesney
Taz Daughtrey	Fang Ching Lim	Fred J. Strauss
Bostjan K. Derganc	William M. Lively	Christine Brown Stryzik
Perry R. DeWeese	James J. Longbucco	Toru Takeshita
James Do	Dieter Look	Richard H. Thayer
Evelyn S. Dow	John Lord	Booker Thomas
Carl Einar Dragstedt	Stan Magee	Patricia Trellue
Sherman Eagles	David Maibor	Theodore J. Urbanowicz
Christof Ebert	Harold Mains	Glenn D. Venables
Leo Egan	Robert A. Martin	Udo Voges
Richard E. Fairley	Tomoo Matsubara	David D. Walden
John W. Fendrich	Mike McAndrew	Dolores Wallace
Jay Forster	Patrick D. McCray	William M. Walsh
Kirby Fortenberry	Christopher McMacken	John W. Walz
Eva Freund	Jerome W. Mersky	Camille SW White-Partain
Richard C. Fries	Bret Michael	Scott A. Whitmire
Roger U. Fujii	Alan Miller	P. A. Wolfgang
Adel N. Ghannam	Celia H. Modell	Paul R. Work
Marilyn Ginsberg-Finner	James W. Moore	Natalie C. Yopconka
John Garth Glynn	Pavol Navrat	Janusz Zalewski
Julio Gonzalez-Sanz	Myrna L. Olson	Geraldine Zimmerman
L. M. Gunther		Peter F. Zoll

When the IEEE-SA Standards Board approved this recommended practice on 25 June 1998, it had the following membership:

**Richard J. Holleman, Chair**

**Donald N. Heirman, Vice Chair**  
**Judith Gorman, Secretary**

Satish K. Aggarwal  
Clyde R. Camp  
James T. Carlo  
Gary R. Engmann  
Harold E. Epstein  
Jay Forster\*  
Thomas F. Garrity  
Ruben D. Garzon

James H. Gurney  
Jim D. Isaak  
Lowell G. Johnson  
Robert Kennelly  
E. G. "Al" Kiener  
Joseph L. Koepfinger\*  
Stephen R. Lambert  
Jim Logothetis  
Donald C. Loughry

L. Bruce McClung  
Louis-François Pau  
Ronald C. Petersen  
Gerald H. Peterson  
John B. Posey  
Gary S. Robinson  
Hans E. Weinrich  
Donald W. Zipse

\*Member Emeritus

Valerie E. Zelenty  
*IEEE Standards Project Editor*

# Contents

1.	Overview.....	1
1.1	Scope.....	1
2.	References.....	2
3.	Definitions.....	2
4.	Considerations for producing a good SRS.....	3
4.1	Nature of the SRS .....	3
4.2	Environment of the SRS .....	3
4.3	Characteristics of a good SRS.....	4
4.4	Joint preparation of the SRS .....	8
4.5	SRS evolution .....	8
4.6	Prototyping.....	9
4.7	Embedding design in the SRS.....	9
4.8	Embedding project requirements in the SRS .....	10
5.	The parts of an SRS .....	10
5.1	Introduction (Section 1 of the SRS).....	11
5.2	Overall description (Section 2 of the SRS).....	12
5.3	Specific requirements (Section 3 of the SRS).....	15
5.4	Supporting information.....	19
	Annex A (informative) SRS templates.....	21
	Annex B (informative) Guidelines for compliance with IEEE/EIA 12207.1-1997 .....	27

# **IEEE Recommended Practice for Software Requirements Specifications**

## **1. Overview**

This recommended practice describes recommended approaches for the specification of software requirements. It is divided into five clauses. Clause 1 explains the scope of this recommended practice. Clause 2 lists the references made to other standards. Clause 3 provides definitions of specific terms used. Clause 4 provides background information for writing a good SRS. Clause 5 discusses each of the essential parts of an SRS. This recommended practice also has two annexes, one which provides alternate format templates, and one which provides guidelines for compliance with IEEE/EIA 12207.1-1997.

### **1.1 Scope**

This is a recommended practice for writing software requirements specifications. It describes the content and qualities of a good software requirements specification (SRS) and presents several sample SRS outlines.

This recommended practice is aimed at specifying requirements of software to be developed but also can be applied to assist in the selection of in-house and commercial software products. However, application to already-developed software could be counterproductive.

When software is embedded in some larger system, such as medical equipment, then issues beyond those identified in this recommended practice may have to be addressed.

This recommended practice describes the process of creating a product and the content of the product. The product is an SRS. This recommended practice can be used to create such an SRS directly or can be used as a model for a more specific standard.

This recommended practice does not identify any specific method, nomenclature, or tool for preparing an SRS.

## 2. References

This recommended practice shall be used in conjunction with the following publications.

ASTM E1340-96, Standard Guide for Rapid Prototyping of Computerized Systems.<sup>1</sup>

IEEE Std 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology.<sup>2</sup>

IEEE Std 730-1998, IEEE Standard for Software Quality Assurance Plans.

IEEE Std 730.1-1995, IEEE Guide for Software Quality Assurance Planning.

IEEE Std 828-1998, IEEE Standard for Software Configuration Management Plans.<sup>3</sup>

IEEE Std 982.1-1988, IEEE Standard Dictionary of Measures to Produce Reliable Software.

IEEE Std 982.2-1988, IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software.

IEEE Std 1002-1987 (Reaff 1992), IEEE Standard Taxonomy for Software Engineering Standards.

IEEE Std 1012-1998, IEEE Standard for Software Verification and Validation.

IEEE Std 1012a-1998, IEEE Standard for Software Verification and Validation: Content Map to IEEE/EIA 12207.1-1997.<sup>4</sup>

IEEE Std 1016-1998, IEEE Recommended Practice for Software Design Descriptions.<sup>5</sup>

IEEE Std 1028-1997, IEEE Standard for Software Reviews.

IEEE Std 1042-1987 (Reaff 1993), IEEE Guide to Software Configuration Management.

IEEE P1058/D2.1, Draft Standard for Software Project Management Plans, dated 5 August 1998.<sup>6</sup>

IEEE Std 1058a-1998, IEEE Standard for Software Project Management Plans: Content Map to IEEE/EIA 12207.1-1997.<sup>7</sup>

IEEE Std 1074-1997, IEEE Standard for Developing Software Life Cycle Processes.

IEEE Std 1233, 1998 Edition, IEEE Guide for Developing System Requirements Specifications.<sup>8</sup>

---

<sup>1</sup>ASTM publications are available from the American Society for Testing and Materials, 100 Barr Harbor Drive, West Conshohocken, PA 19428-2959, USA.

<sup>2</sup>IEEE publications are available from the Institute of Electrical and Electronics Engineers, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331, USA.

<sup>3</sup>As this standard goes to press, IEEE Std 828-1998; IEEE Std 1012a-1998; IEEE Std 1016-1998; and IEEE Std 1233, 1998 Edition are approved but not yet published. The draft standards are, however, available from the IEEE. Anticipated publication date is Fall 1998. Contact the IEEE Standards Department at 1 (732) 562-3800 for status information.

<sup>4</sup>See Footnote 3.

<sup>5</sup>See Footnote 3.

<sup>6</sup>Upon approval of IEEE P1058 by the IEEE-SA Standards Board, this standard will be integrated with IEEE Std 1058a-1998 and published as IEEE Std 1058, 1998 Edition. Approval is expected 8 December 1998.

<sup>7</sup>As this standard goes to press, IEEE Std 1058a-1998 is approved but not yet published. The draft standard is, however, available from the IEEE. Anticipated publication date is December 1998. Contact the IEEE Standards Department at 1 (732) 562-3800 for status information. See Footnote 6.

<sup>8</sup>See Footnote 3.

### 3. Definitions

In general the definitions of terms used in this recommended practice conform to the definitions provided in IEEE Std 610.12-1990. The definitions below are key terms as they are used in this recommended practice.

**3.1 contract:** A legally binding document agreed upon by the customer and supplier. This includes the technical and organizational requirements, cost, and schedule for a product. A contract may also contain informal but useful information such as the commitments or expectations of the parties involved.

**3.2 customer:** The person, or persons, who pay for the product and usually (but not necessarily) decide the requirements. In the context of this recommended practice the customer and the supplier may be members of the same organization.

**3.3 supplier:** The person, or persons, who produce a product for a customer. In the context of this recommended practice, the customer and the supplier may be members of the same organization.

**3.4 user:** The person, or persons, who operate or interact directly with the product. The user(s) and the customer(s) are often not the same person(s).

### 4. Considerations for producing a good SRS

This clause provides background information that should be considered when writing an SRS. This includes the following:

- a) Nature of the SRS;
- b) Environment of the SRS;
- c) Characteristics of a good SRS;
- d) Joint preparation of the SRS;
- e) SRS evolution;
- f) Prototyping;
- g) Embedding design in the SRS;
- h) Embedding project requirements in the SRS.

#### 4.1 Nature of the SRS

The SRS is a specification for a particular software product, program, or set of programs that performs certain functions in a specific environment. The SRS may be written by one or more representatives of the supplier, one or more representatives of the customer, or by both. Subclause 4.4 recommends both.

The basic issues that the SRS writer(s) shall address are the following:

- a) *Functionality.* What is the software supposed to do?
- b) *External interfaces.* How does the software interact with people, the system's hardware, other hardware, and other software?
- c) *Performance.* What is the speed, availability, response time, recovery time of various software functions, etc.?
- d) *Attributes.* What are the portability, correctness, maintainability, security, etc. considerations?
- e) *Design constraints imposed on an implementation.* Are there any required standards in effect, implementation language, policies for database integrity, resource limits, operating environment(s) etc.?

The SRS writer(s) should avoid placing either design or project requirements in the SRS.

For recommended contents of an SRS see Clause 5.

## 4.2 Environment of the SRS

It is important to consider the part that the SRS plays in the total project plan, which is defined in IEEE Std 610.12-1990. The software may contain essentially all the functionality of the project or it may be part of a larger system. In the latter case typically there will be an SRS that will state the interfaces between the system and its software portion, and will place external performance and functionality requirements upon the software portion. Of course the SRS should then agree with and expand upon these system requirements.

IEEE Std 1074-1997 describes the steps in the software life cycle and the applicable inputs for each step. Other standards, such as those listed in Clause 2, relate to other parts of the software life cycle and so may complement software requirements.

Since the SRS has a specific role to play in the software development process, the SRS writer(s) should be careful not to go beyond the bounds of that role. This means the SRS

- a) Should correctly define all of the software requirements. A software requirement may exist because of the nature of the task to be solved or because of a special characteristic of the project.
- b) Should not describe any design or implementation details. These should be described in the design stage of the project.
- c) Should not impose additional constraints on the software. These are properly specified in other documents such as a software quality assurance plan.

Therefore, a properly written SRS limits the range of valid designs, but does not specify any particular design.

## 4.3 Characteristics of a good SRS

An SRS should be

- a) Correct;
- b) Unambiguous;
- c) Complete;
- d) Consistent;
- e) Ranked for importance and/or stability;
- f) Verifiable;
- g) Modifiable;
- h) Traceable.

### 4.3.1 Correct

An SRS is correct if, and only if, every requirement stated therein is one that the software shall meet.

There is no tool or procedure that ensures correctness. The SRS should be compared with any applicable superior specification, such as a system requirements specification, with other project documentation, and with other applicable standards, to ensure that it agrees. Alternatively the customer or user can determine if the SRS correctly reflects the actual needs. Traceability makes this procedure easier and less prone to error (see 4.3.8).

### 4.3.2 Unambiguous

An SRS is unambiguous if, and only if, every requirement stated therein has only one interpretation. As a minimum, this requires that each characteristic of the final product be described using a single unique term.

In cases where a term used in a particular context could have multiple meanings, the term should be included in a glossary where its meaning is made more specific.

An SRS is an important part of the requirements process of the software life cycle and is used in design, implementation, project monitoring, verification and validation, and in training as described in IEEE Std 1074-1997. The SRS should be unambiguous both to those who create it and to those who use it. However, these groups often do not have the same background and therefore do not tend to describe software requirements the same way. Representations that improve the requirements specification for the developer may be counterproductive in that they diminish understanding to the user and vice versa.

Subclauses 4.3.2.1 through 4.3.2.3 recommend how to avoid ambiguity.

#### **4.3.2.1 Natural language pitfalls**

Requirements are often written in natural language (e.g., English). Natural language is inherently ambiguous. A natural language SRS should be reviewed by an independent party to identify ambiguous use of language so that it can be corrected.

#### **4.3.2.2 Requirements specification languages**

One way to avoid the ambiguity inherent in natural language is to write the SRS in a particular requirements specification language. Its language processors automatically detect many lexical, syntactic, and semantic errors.

One disadvantage in the use of such languages is the length of time required to learn them. Also, many non-technical users find them unintelligible. Moreover, these languages tend to be better at expressing certain types of requirements and addressing certain types of systems. Thus, they may influence the requirements in subtle ways.

#### **4.3.2.3 Representation tools**

In general, requirements methods and languages and the tools that support them fall into three general categories—object, process, and behavioral. Object-oriented approaches organize the requirements in terms of real-world objects, their attributes, and the services performed by those objects. Process-based approaches organize the requirements into hierarchies of functions that communicate via data flows. Behavioral approaches describe external behavior of the system in terms of some abstract notion (such as predicate calculus), mathematical functions, or state machines.

The degree to which such tools and methods may be useful in preparing an SRS depends upon the size and complexity of the program. No attempt is made here to describe or endorse any particular tool.

When using any of these approaches it is best to retain the natural language descriptions. That way, customers unfamiliar with the notations can still understand the SRS.

#### **4.3.3 Complete**

An SRS is complete if, and only if, it includes the following elements:

- a) All significant requirements, whether relating to functionality, performance, design constraints, attributes, or external interfaces. In particular any external requirements imposed by a system specification should be acknowledged and treated.

- b) Definition of the responses of the software to all realizable classes of input data in all realizable classes of situations. Note that it is important to specify the responses to both valid and invalid input values.
- c) Full labels and references to all figures, tables, and diagrams in the SRS and definition of all terms and units of measure.

#### **4.3.3.1 Use of TBDs**

Any SRS that uses the phrase “to be determined” (TBD) is not a complete SRS. The TBD is, however, occasionally necessary and should be accompanied by

- a) A description of the conditions causing the TBD (e.g., why an answer is not known) so that the situation can be resolved;
- b) A description of what must be done to eliminate the TBD, who is responsible for its elimination, and by when it must be eliminated.

#### **4.3.4 Consistent**

Consistency refers to internal consistency. If an SRS does not agree with some higher-level document, such as a system requirements specification, then it is not correct (see 4.3.1).

##### **4.3.4.1 Internal consistency**

An SRS is internally consistent if, and only if, no subset of individual requirements described in it conflict. The three types of likely conflicts in an SRS are as follows:

- a) The specified characteristics of real-world objects may conflict. For example,
  - 1) The format of an output report may be described in one requirement as tabular but in another as textual.
  - 2) One requirement may state that all lights shall be green while another may state that all lights shall be blue.
- b) There may be logical or temporal conflict between two specified actions. For example,
  - 1) One requirement may specify that the program will add two inputs and another may specify that the program will multiply them.
  - 2) One requirement may state that “A” must always follow “B,” while another may require that “A and B” occur simultaneously.
- c) Two or more requirements may describe the same real-world object but use different terms for that object. For example, a program’s request for a user input may be called a “prompt” in one requirement and a “cue” in another. The use of standard terminology and definitions promotes consistency.

#### **4.3.5 Ranked for importance and/or stability**

An SRS is ranked for importance and/or stability if each requirement in it has an identifier to indicate either the importance or stability of that particular requirement.

Typically, all of the requirements that relate to a software product are not equally important. Some requirements may be essential, especially for life-critical applications, while others may be desirable.

Each requirement in the SRS should be identified to make these differences clear and explicit. Identifying the requirements in the following manner helps:

- a) Have customers give more careful consideration to each requirement, which often clarifies any hidden assumptions they may have.
- b) Have developers make correct design decisions and devote appropriate levels of effort to the different parts of the software product.

#### **4.3.5.1 Degree of stability**

One method of identifying requirements uses the dimension of stability. Stability can be expressed in terms of the number of expected changes to any requirement based on experience or knowledge of forthcoming events that affect the organization, functions, and people supported by the software system.

#### **4.3.5.2 Degree of necessity**

Another way to rank requirements is to distinguish classes of requirements as essential, conditional, and optional.

- a) *Essential.* Implies that the software will not be acceptable unless these requirements are provided in an agreed manner.
- b) *Conditional.* Implies that these are requirements that would enhance the software product, but would not make it unacceptable if they are absent.
- c) *Optional.* Implies a class of functions that may or may not be worthwhile. This gives the supplier the opportunity to propose something that exceeds the SRS.

#### **4.3.6 Verifiable**

An SRS is verifiable if, and only if, every requirement stated therein is verifiable. A requirement is verifiable if, and only if, there exists some finite cost-effective process with which a person or machine can check that the software product meets the requirement. In general any ambiguous requirement is not verifiable.

Nonverifiable requirements include statements such as “works well,” “good human interface,” and “shall usually happen.” These requirements cannot be verified because it is impossible to define the terms “good,” “well,” or “usually.” The statement that “the program shall never enter an infinite loop” is nonverifiable because the testing of this quality is theoretically impossible.

An example of a verifiable statement is

*Output of the program shall be produced within 20 s of event × 60% of the time; and shall be produced within 30 s of event × 100% of the time.*

This statement can be verified because it uses concrete terms and measurable quantities.

If a method cannot be devised to determine whether the software meets a particular requirement, then that requirement should be removed or revised.

#### 4.3.7 Modifiable

An SRS is modifiable if, and only if, its structure and style are such that any changes to the requirements can be made easily, completely, and consistently while retaining the structure and style. Modifiability generally requires an SRS to

- a) Have a coherent and easy-to-use organization with a table of contents, an index, and explicit cross-referencing;
- b) Not be redundant (i.e., the same requirement should not appear in more than one place in the SRS);
- c) Express each requirement separately, rather than intermixed with other requirements.

Redundancy itself is not an error, but it can easily lead to errors. Redundancy can occasionally help to make an SRS more readable, but a problem can arise when the redundant document is updated. For instance, a requirement may be altered in only one of the places where it appears. The SRS then becomes inconsistent. Whenever redundancy is necessary, the SRS should include explicit cross-references to make it modifiable.

#### 4.3.8 Traceable

An SRS is traceable if the origin of each of its requirements is clear and if it facilitates the referencing of each requirement in future development or enhancement documentation. The following two types of traceability are recommended:

- a) *Backward traceability* (i.e., to previous stages of development). This depends upon each requirement explicitly referencing its source in earlier documents.
- b) *Forward traceability* (i.e., to all documents spawned by the SRS). This depends upon each requirement in the SRS having a unique name or reference number.

The forward traceability of the SRS is especially important when the software product enters the operation and maintenance phase. As code and design documents are modified, it is essential to be able to ascertain the complete set of requirements that may be affected by those modifications.

### 4.4 Joint preparation of the SRS

The software development process should begin with supplier and customer agreement on what the completed software must do. This agreement, in the form of an SRS, should be jointly prepared. This is important because usually neither the customer nor the supplier is qualified to write a good SRS alone.

- a) Customers usually do not understand the software design and development process well enough to write a usable SRS.
- b) Suppliers usually do not understand the customer's problem and field of endeavor well enough to specify requirements for a satisfactory system.

Therefore, the customer and the supplier should work together to produce a well-written and completely understood SRS.

A special situation exists when a system and its software are both being defined concurrently. Then the functionality, interfaces, performance, and other attributes and constraints of the software are not predefined, but rather are jointly defined and subject to negotiation and change. This makes it more difficult, but no less important, to meet the characteristics stated in 4.3. In particular, an SRS that does not comply with the requirements of its parent system specification is incorrect.

This recommended practice does not specifically discuss style, language usage, or techniques of good writing. It is quite important, however, that an SRS be well written. General technical writing books can be used for guidance.

## 4.5 SRS evolution

The SRS may need to evolve as the development of the software product progresses. It may be impossible to specify some details at the time the project is initiated (e.g., it may be impossible to define all of the screen formats for an interactive program during the requirements phase). Additional changes may ensue as deficiencies, shortcomings, and inaccuracies are discovered in the SRS.

Two major considerations in this process are the following:

- a) Requirements should be specified as completely and thoroughly as is known at the time, even if evolutionary revisions can be foreseen as inevitable. The fact that they are incomplete should be noted.
- b) A formal change process should be initiated to identify, control, track, and report projected changes. Approved changes in requirements should be incorporated in the SRS in such a way as to
  - 1) Provide an accurate and complete audit trail of changes;
  - 2) Permit the review of current and superseded portions of the SRS.

## 4.6 Prototyping

Prototyping is used frequently during the requirements portion of a project. Many tools exist that allow a prototype, exhibiting some characteristics of a system, to be created very quickly and easily. See also ASTM E1340-96.

Prototypes are useful for the following reasons:

- a) The customer may be more likely to view the prototype and react to it than to read the SRS and react to it. Thus, the prototype provides quick feedback.
- b) The prototype displays unanticipated aspects of the systems behavior. Thus, it produces not only answers but also new questions. This helps reach closure on the SRS.
- c) An SRS based on a prototype tends to undergo less change during development, thus shortening development time.

A prototype should be used as a way to elicit software requirements. Some characteristics such as screen or report formats can be extracted directly from the prototype. Other requirements can be inferred by running experiments with the prototype.

## 4.7 Embedding design in the SRS

A requirement specifies an externally visible function or attribute of a system. A design describes a particular subcomponent of a system and/or its interfaces with other subcomponents. The SRS writer(s) should clearly distinguish between identifying required design constraints and projecting a specific design. Note that every requirement in the SRS limits design alternatives. This does not mean, though, that every requirement is design.

The SRS should specify what functions are to be performed on what data to produce what results at what location for whom. The SRS should focus on the services to be performed. The SRS should not normally specify design items such as the following:

- a) Partitioning the software into modules;
- b) Allocating functions to the modules;
- c) Describing the flow of information or control between modules;
- d) Choosing data structures.

#### **4.7.1 Necessary design requirements**

In special cases some requirements may severely restrict the design. For example, security or safety requirements may reflect directly into design such as the need to

- a) Keep certain functions in separate modules;
- b) Permit only limited communication between some areas of the program;
- c) Check data integrity for critical variables.

Examples of valid design constraints are physical requirements, performance requirements, software development standards, and software quality assurance standards.

Therefore, the requirements should be stated from a purely external viewpoint. When using models to illustrate the requirements, remember that the model only indicates the external behavior, and does not specify a design.

#### **4.8 Embedding project requirements in the SRS**

The SRS should address the software product, not the process of producing the software product.

Project requirements represent an understanding between the customer and the supplier about contractual matters pertaining to production of software and thus should not be included in the SRS. These normally include items such as

- a) Cost;
- b) Delivery schedules;
- c) Reporting procedures;
- d) Software development methods;
- e) Quality assurance;
- f) Validation and verification criteria;
- g) Acceptance procedures.

Project requirements are specified in other documents, typically in a software development plan, a software quality assurance plan, or a statement of work.

### **5. The parts of an SRS**

This clause discusses each of the essential parts of the SRS. These parts are arranged in Figure 1 in an outline that can serve as an example for writing an SRS.

While an SRS does not have to follow this outline or use the names given here for its parts, a good SRS should include all the information discussed here.

<b>Table of Contents</b>	
1.	Introduction
1.1	Purpose
1.2	Scope
1.3	Definitions, acronyms, and abbreviations
1.4	References
1.5	Overview
2.	Overall description
2.1	Product perspective
2.2	Product functions
2.3	User characteristics
2.4	Constraints
2.5	Assumptions and dependencies
3.	Specific requirements (See 5.3.1 through 5.3.8 for explanations of possible specific requirements. See also Annex A for several different ways of organizing this section of the SRS.)
	Appendixes
	Index

**Figure 1—Prototype SRS outline**

## 5.1 Introduction (Section 1 of the SRS)

The introduction of the SRS should provide an overview of the entire SRS. It should contain the following subsections:

- a) Purpose;
- b) Scope;
- c) Definitions, acronyms, and abbreviations;
- d) References;
- e) Overview.

### 5.1.1 Purpose (1.1 of the SRS)

This subsection should

- a) Delineate the purpose of the SRS;
- b) Specify the intended audience for the SRS.

### 5.1.2 Scope (1.2 of the SRS)

This subsection should

- a) Identify the software product(s) to be produced by name (e.g., Host DBMS, Report Generator, etc.);
- b) Explain what the software product(s) will, and, if necessary, will not do;
- c) Describe the application of the software being specified, including relevant benefits, objectives, and goals;
- d) Be consistent with similar statements in higher-level specifications (e.g., the system requirements specification), if they exist.

### **5.1.3 Definitions, acronyms, and abbreviations (1.3 of the SRS)**

This subsection should provide the definitions of all terms, acronyms, and abbreviations required to properly interpret the SRS. This information may be provided by reference to one or more appendixes in the SRS or by reference to other documents.

### **5.1.4 References (1.4 of the SRS)**

This subsection should

- a) Provide a complete list of all documents referenced elsewhere in the SRS;
- b) Identify each document by title, report number (if applicable), date, and publishing organization;
- c) Specify the sources from which the references can be obtained.

This information may be provided by reference to an appendix or to another document.

### **5.1.5 Overview (1.5 of the SRS)**

This subsection should

- a) Describe what the rest of the SRS contains;
- b) Explain how the SRS is organized.

## **5.2 Overall description (Section 2 of the SRS)**

This section of the SRS should describe the general factors that affect the product and its requirements. This section does not state specific requirements. Instead, it provides a background for those requirements, which are defined in detail in Section 3 of the SRS, and makes them easier to understand.

This section usually consists of six subsections, as follows:

- a) Product perspective;
- b) Product functions;
- c) User characteristics;
- d) Constraints;
- e) Assumptions and dependencies;
- f) Apportioning of requirements.

### **5.2.1 Product perspective (2.1 of the SRS)**

This subsection of the SRS should put the product into perspective with other related products. If the product is independent and totally self-contained, it should be so stated here. If the SRS defines a product that is a component of a larger system, as frequently occurs, then this subsection should relate the requirements of that larger system to functionality of the software and should identify interfaces between that system and the software.

A block diagram showing the major components of the larger system, interconnections, and external interfaces can be helpful.

This subsection should also describe how the software operates inside various constraints. For example, these constraints could include

- a) System interfaces;
- b) User interfaces;
- c) Hardware interfaces;
- d) Software interfaces;
- e) Communications interfaces;
- f) Memory;
- g) Operations;
- h) Site adaptation requirements.

### **5.2.1.1 System interfaces**

This should list each system interface and identify the functionality of the software to accomplish the system requirement and the interface description to match the system.

### **5.2.1.2 User interfaces**

This should specify the following:

- a) *The logical characteristics of each interface between the software product and its users.* This includes those configuration characteristics (e.g., required screen formats, page or window layouts, content of any reports or menus, or availability of programmable function keys) necessary to accomplish the software requirements.
- b) *All the aspects of optimizing the interface with the person who must use the system.* This may simply comprise a list of do's and don'ts on how the system will appear to the user. One example may be a requirement for the option of long or short error messages. Like all others, these requirements should be verifiable, e.g., "a clerk typist grade 4 can do function X in Z min after 1 h of training" rather than "a typist can do function X." (This may also be specified in the Software System Attributes under a section titled Ease of Use.)

### **5.2.1.3 Hardware interfaces**

This should specify the logical characteristics of each interface between the software product and the hardware components of the system. This includes configuration characteristics (number of ports, instruction sets, etc.). It also covers such matters as what devices are to be supported, how they are to be supported, and protocols. For example, terminal support may specify full-screen support as opposed to line-by-line support.

### **5.2.1.4 Software interfaces**

This should specify the use of other required software products (e.g., a data management system, an operating system, or a mathematical package), and interfaces with other application systems (e.g., the linkage between an accounts receivable system and a general ledger system). For each required software product, the following should be provided:

- Name;
- Mnemonic;
- Specification number;
- Version number;
- Source.

For each interface, the following should be provided:

- Discussion of the purpose of the interfacing software as related to this software product.
- Definition of the interface in terms of message content and format. It is not necessary to detail any well-documented interface, but a reference to the document defining the interface is required.

#### **5.2.1.5 Communications interfaces**

This should specify the various interfaces to communications such as local network protocols, etc.

#### **5.2.1.6 Memory constraints**

This should specify any applicable characteristics and limits on primary and secondary memory.

#### **5.2.1.7 Operations**

This should specify the normal and special operations required by the user such as

- a) The various modes of operations in the user organization (e.g., user-initiated operations);
- b) Periods of interactive operations and periods of unattended operations;
- c) Data processing support functions;
- d) Backup and recovery operations.

NOTE—This is sometimes specified as part of the User Interfaces section.

#### **5.2.1.8 Site adaptation requirements**

This should

- a) Define the requirements for any data or initialization sequences that are specific to a given site, mission, or operational mode (e.g., grid values, safety limits, etc.);
- b) Specify the site or mission-related features that should be modified to adapt the software to a particular installation.

#### **5.2.2 Product functions (2.2 of the SRS)**

This subsection of the SRS should provide a summary of the major functions that the software will perform. For example, an SRS for an accounting program may use this part to address customer account maintenance, customer statement, and invoice preparation without mentioning the vast amount of detail that each of those functions requires.

Sometimes the function summary that is necessary for this part can be taken directly from the section of the higher-level specification (if one exists) that allocates particular functions to the software product. Note that for the sake of clarity

- a) The functions should be organized in a way that makes the list of functions understandable to the customer or to anyone else reading the document for the first time.
- b) Textual or graphical methods can be used to show the different functions and their relationships. Such a diagram is not intended to show a design of a product, but simply shows the logical relationships among variables.

### 5.2.3 User characteristics (2.3 of the SRS)

This subsection of the SRS should describe those general characteristics of the intended users of the product including educational level, experience, and technical expertise. It should not be used to state specific requirements, but rather should provide the reasons why certain specific requirements are later specified in Section 3 of the SRS.

### 5.2.4 Constraints (2.4 of the SRS)

This subsection of the SRS should provide a general description of any other items that will limit the developer's options. These include

- a) Regulatory policies;
- b) Hardware limitations (e.g., signal timing requirements);
- c) Interfaces to other applications;
- d) Parallel operation;
- e) Audit functions;
- f) Control functions;
- g) Higher-order language requirements;
- h) Signal handshake protocols (e.g., XON-XOFF, ACK-NACK);
- i) Reliability requirements;
- j) Criticality of the application;
- k) Safety and security considerations.

### 5.2.5 Assumptions and dependencies (2.5 of the SRS)

This subsection of the SRS should list each of the factors that affect the requirements stated in the SRS. These factors are not design constraints on the software but are, rather, any changes to them that can affect the requirements in the SRS. For example, an assumption may be that a specific operating system will be available on the hardware designated for the software product. If, in fact, the operating system is not available, the SRS would then have to change accordingly.

### 5.2.6 Apportioning of requirements (2.6 of the SRS)

This subsection of the SRS should identify requirements that may be delayed until future versions of the system.

## 5.3 Specific requirements (Section 3 of the SRS)

This section of the SRS should contain all of the software requirements to a level of detail sufficient to enable designers to design a system to satisfy those requirements, and testers to test that the system satisfies those requirements. Throughout this section, every stated requirement should be externally perceivable by users, operators, or other external systems. These requirements should include at a minimum a description of every input (stimulus) into the system, every output (response) from the system, and all functions performed by the system in response to an input or in support of an output. As this is often the largest and most important part of the SRS, the following principles apply:

- a) Specific requirements should be stated in conformance with all the characteristics described in 4.3.
- b) Specific requirements should be cross-referenced to earlier documents that relate.
- c) All requirements should be uniquely identifiable.
- d) Careful attention should be given to organizing the requirements to maximize readability.

Before examining specific ways of organizing the requirements it is helpful to understand the various items that comprise requirements as described in 5.3.1 through 5.3.7.

### **5.3.1 External interfaces**

This should be a detailed description of all inputs into and outputs from the software system. It should complement the interface descriptions in 5.2 and should not repeat information there.

It should include both content and format as follows:

- a) Name of item;
- b) Description of purpose;
- c) Source of input or destination of output;
- d) Valid range, accuracy, and/or tolerance;
- e) Units of measure;
- f) Timing;
- g) Relationships to other inputs/outputs;
- h) Screen formats/organization;
- i) Window formats/organization;
- j) Data formats;
- k) Command formats;
- l) End messages.

### **5.3.2 Functions**

Functional requirements should define the fundamental actions that must take place in the software in accepting and processing the inputs and in processing and generating the outputs. These are generally listed as “shall” statements starting with “The system shall...”

These include

- a) Validity checks on the inputs
- b) Exact sequence of operations
- c) Responses to abnormal situations, including
  - 1) Overflow
  - 2) Communication facilities
  - 3) Error handling and recovery
- d) Effect of parameters
- e) Relationship of outputs to inputs, including
  - 1) Input/output sequences
  - 2) Formulas for input to output conversion

It may be appropriate to partition the functional requirements into subfunctions or subprocesses. This does not imply that the software design will also be partitioned that way.

### **5.3.3 Performance requirements**

This subsection should specify both the static and the dynamic numerical requirements placed on the software or on human interaction with the software as a whole. Static numerical requirements may include the following:

- a) The number of terminals to be supported;
- b) The number of simultaneous users to be supported;
- c) Amount and type of information to be handled.

Static numerical requirements are sometimes identified under a separate section entitled Capacity.

Dynamic numerical requirements may include, for example, the numbers of transactions and tasks and the amount of data to be processed within certain time periods for both normal and peak workload conditions.

All of these requirements should be stated in measurable terms.

For example,

*95% of the transactions shall be processed in less than 1 s.*

rather than,

*An operator shall not have to wait for the transaction to complete.*

NOTE—Numerical limits applied to one specific function are normally specified as part of the processing subparagraph description of that function.

### 5.3.4 Logical database requirements

This should specify the logical requirements for any information that is to be placed into a database. This may include the following:

- a) Types of information used by various functions;
- b) Frequency of use;
- c) Accessing capabilities;
- d) Data entities and their relationships;
- e) Integrity constraints;
- f) Data retention requirements.

### 5.3.5 Design constraints

This should specify design constraints that can be imposed by other standards, hardware limitations, etc.

#### 5.3.5.1 Standards compliance

This subsection should specify the requirements derived from existing standards or regulations. They may include the following:

- a) Report format;
- b) Data naming;
- c) Accounting procedures;
- d) Audit tracing.

For example, this could specify the requirement for software to trace processing activity. Such traces are needed for some applications to meet minimum regulatory or financial standards. An audit trace requirement may, for example, state that all changes to a payroll database must be recorded in a trace file with before and after values.

### 5.3.6 Software system attributes

There are a number of attributes of software that can serve as requirements. It is important that required attributes be specified so that their achievement can be objectively verified. Subclauses 5.3.6.1 through 5.3.6.5 provide a partial list of examples.

### **5.3.6.1 Reliability**

This should specify the factors required to establish the required reliability of the software system at time of delivery.

### **5.3.6.2 Availability**

This should specify the factors required to guarantee a defined availability level for the entire system such as checkpoint, recovery, and restart.

### **5.3.6.3 Security**

This should specify the factors that protect the software from accidental or malicious access, use, modification, destruction, or disclosure. Specific requirements in this area could include the need to

- a) Utilize certain cryptographical techniques;
- b) Keep specific log or history data sets;
- c) Assign certain functions to different modules;
- d) Restrict communications between some areas of the program;
- e) Check data integrity for critical variables.

### **5.3.6.4 Maintainability**

This should specify attributes of software that relate to the ease of maintenance of the software itself. There may be some requirement for certain modularity, interfaces, complexity, etc. Requirements should not be placed here just because they are thought to be good design practices.

### **5.3.6.5 Portability**

This should specify attributes of software that relate to the ease of porting the software to other host machines and/or operating systems. This may include the following:

- a) Percentage of components with host-dependent code;
- b) Percentage of code that is host dependent;
- c) Use of a proven portable language;
- d) Use of a particular compiler or language subset;
- e) Use of a particular operating system.

## **5.3.7 Organizing the specific requirements**

For anything but trivial systems the detailed requirements tend to be extensive. For this reason, it is recommended that careful consideration be given to organizing these in a manner optimal for understanding. There is no one optimal organization for all systems. Different classes of systems lend themselves to different organizations of requirements in Section 3 of the SRS. Some of these organizations are described in 5.3.7.1 through 5.3.7.7.

### **5.3.7.1 System mode**

Some systems behave quite differently depending on the mode of operation. For example, a control system may have different sets of functions depending on its mode: training, normal, or emergency. When organizing this section by mode, the outline in A.1 or A.2 should be used. The choice depends on whether interfaces and performance are dependent on mode.

### 5.3.7.2 User class

Some systems provide different sets of functions to different classes of users. For example, an elevator control system presents different capabilities to passengers, maintenance workers, and fire fighters. When organizing this section by user class, the outline in A.3 should be used.

### 5.3.7.3 Objects

Objects are real-world entities that have a counterpart within the system. For example, in a patient monitoring system, objects include patients, sensors, nurses, rooms, physicians, medicines, etc. Associated with each object is a set of attributes (of that object) and functions (performed by that object). These functions are also called services, methods, or processes. When organizing this section by object, the outline in A.4 should be used. Note that sets of objects may share attributes and services. These are grouped together as classes.

### 5.3.7.4 Feature

A feature is an externally desired service by the system that may require a sequence of inputs to effect the desired result. For example, in a telephone system, features include local call, call forwarding, and conference call. Each feature is generally described in a sequence of stimulus-response pairs. When organizing this section by feature, the outline in A.5 should be used.

### 5.3.7.5 Stimulus

Some systems can be best organized by describing their functions in terms of stimuli. For example, the functions of an automatic aircraft landing system may be organized into sections for loss of power, wind shear, sudden change in roll, vertical velocity excessive, etc. When organizing this section by stimulus, the outline in A.6 should be used.

### 5.3.7.6 Response

Some systems can be best organized by describing all the functions in support of the generation of a response. For example, the functions of a personnel system may be organized into sections corresponding to all functions associated with generating paychecks, all functions associated with generating a current list of employees, etc. The outline in A.6 (with all occurrences of stimulus replaced with response) should be used.

### 5.3.7.7 Functional hierarchy

When none of the above organizational schemes prove helpful, the overall functionality can be organized into a hierarchy of functions organized by either common inputs, common outputs, or common internal data access. Data flow diagrams and data dictionaries can be used to show the relationships between and among the functions and data. When organizing this section by functional hierarchy, the outline in A.7 should be used.

### 5.3.8 Additional comments

Whenever a new SRS is contemplated, more than one of the organizational techniques given in 5.3.7.7 may be appropriate. In such cases, organize the specific requirements for multiple hierarchies tailored to the specific needs of the system under specification. For example, see A.8 for an organization combining user class and feature. Any additional requirements may be put in a separate section at the end of the SRS.

There are many notations, methods, and automated support tools available to aid in the documentation of requirements. For the most part, their usefulness is a function of organization. For example, when organizing by mode, finite state machines or state charts may prove helpful; when organizing by object, object-oriented

analysis may prove helpful; when organizing by feature, stimulus-response sequences may prove helpful; and when organizing by functional hierarchy, data flow diagrams and data dictionaries may prove helpful.

In any of the outlines given in A.1 through A.8, those sections called “Functional Requirement  $i$ ” may be described in native language (e.g., English), in pseudocode, in a system definition language, or in four subsections titled: Introduction, Inputs, Processing, and Outputs.

## 5.4 Supporting information

The supporting information makes the SRS easier to use. It includes the following:

- a) Table of contents;
- b) Index;
- c) Appendixes.

### 5.4.1 Table of contents and index

The table of contents and index are quite important and should follow general compositional practices.

### 5.4.2 Appendixes

The appendixes are not always considered part of the actual SRS and are not always necessary. They may include

- a) Sample input/output formats, descriptions of cost analysis studies, or results of user surveys;
- b) Supporting or background information that can help the readers of the SRS;
- c) A description of the problems to be solved by the software;
- d) Special packaging instructions for the code and the media to meet security, export, initial loading, or other requirements.

When appendixes are included, the SRS should explicitly state whether or not the appendixes are to be considered part of the requirements.

## Annex A

(informative)

### SRS templates

#### A.1 Template of SRS Section 3 organized by mode: Version 1

- 3. Specific requirements
  - 3.1 External interface requirements
    - 3.1.1 User interfaces
    - 3.1.2 Hardware interfaces
    - 3.1.3 Software interfaces
    - 3.1.4 Communications interfaces
  - 3.2 Functional requirements
    - 3.2.1 Mode 1
      - 3.2.1.1 Functional requirement 1.1
      - .
      - .
      - .
      - 3.2.1.*n* Functional requirement 1.*n*
    - 3.2.2 Mode 2
      - .
      - .
      - .
    - 3.2.*m* Mode *m*
      - 3.2.*m*.1 Functional requirement *m*.1
      - .
      - .
      - 3.2.*m*.*n* Functional requirement *m*.*n*
  - 3.3 Performance requirements
  - 3.4 Design constraints
  - 3.5 Software system attributes
  - 3.6 Other requirements

#### A.2 Template of SRS Section 3 organized by mode: Version 2

- 3. Specific requirements
  - 3.1. Functional requirements
    - 3.1.1 Mode 1
      - 3.1.1.1 External interfaces
        - 3.1.1.1.1 User interfaces
        - 3.1.1.1.2 Hardware interfaces
        - 3.1.1.1.3 Software interfaces
        - 3.1.1.1.4 Communications interfaces
      - 3.1.1.2 Functional requirements
        - 3.1.1.2.1 Functional requirement 1
        - .
        - .
        - .

- 3.1.1.2.*n* Functional requirement *n*
- 3.1.1.3 Performance
- 3.1.2 Mode 2
  - .
  - .
  - .
- 3.1.*m* Mode *m*
- 3.2 Design constraints
- 3.3 Software system attributes
- 3.4 Other requirements

### A.3 Template of SRS Section 3 organized by user class

- 3. Specific requirements
- 3.1 External interface requirements
  - 3.1.1 User interfaces
  - 3.1.2 Hardware interfaces
  - 3.1.3 Software interfaces
  - 3.1.4 Communications interfaces
- 3.2 Functional requirements
  - 3.2.1 User class 1
    - 3.2.1.1 Functional requirement 1.1
    - .
    - .
    - .
  - 3.2.1.*n* Functional requirement 1.*n*
  - 3.2.2 User class 2
    - .
    - .
    - .
  - 3.2.*m* User class *m*
    - 3.2.*m*.1 Functional requirement *m*.1
    - .
    - .
    - .
  - 3.2.*m*.*n* Functional requirement *m*.*n*
- 3.3 Performance requirements
- 3.4 Design constraints
- 3.5 Software system attributes
- 3.6 Other requirements

### A.4 Template of SRS Section 3 organized by object

- 3. Specific requirements
- 3.1 External interface requirements
  - 3.1.1 User interfaces
  - 3.1.2 Hardware interfaces
  - 3.1.3 Software interfaces
  - 3.1.4 Communications interfaces
- 3.2 Classes/Objects
  - 3.2.1 Class/Object 1

- 3.2.1.1 Attributes (direct or inherited)
  - 3.2.1.1.1 Attribute 1
  - .
  - .
  - .
  - 3.2.1.1.*n* Attribute *n*
- 3.2.1.2 Functions (services, methods, direct or inherited)
  - 3.2.1.2.1 Functional requirement 1.1
  - .
  - .
  - .
  - 3.2.1.2.*m* Functional requirement 1.*m*
- 3.2.1.3 Messages (communications received or sent)
- 3.2.2 Class/Object 2
  - .
  - .
  - .
  - 3.2.*p* Class/Object *p*
- 3.3 Performance requirements
- 3.4 Design constraints
- 3.5 Software system attributes
- 3.6 Other requirements

## A.5 Template of SRS Section 3 organized by feature

- 3. Specific requirements
  - 3.1 External interface requirements
    - 3.1.1 User interfaces
    - 3.1.2 Hardware interfaces
    - 3.1.3 Software interfaces
    - 3.1.4 Communications interfaces
  - 3.2 System features
    - 3.2.1 System Feature 1
      - 3.2.1.1 Introduction/Purpose of feature
      - 3.2.1.2 Stimulus/Response sequence
      - 3.2.1.3 Associated functional requirements
        - 3.2.1.3.1 Functional requirement 1
        - .
        - .
        - .
        - 3.2.1.3.*n* Functional requirement *n*
    - 3.2.2 System feature 2
      - .
      - .
      - .
    - .
    - .
    - 3.2.*m* System feature *m*
      - .
      - .
      - .
  - 3.3 Performance requirements
  - 3.4 Design constraints
  - 3.5 Software system attributes
  - 3.6 Other requirements

## A.6 Template of SRS Section 3 organized by stimulus

- 3. Specific requirements
  - 3.1 External interface requirements
    - 3.1.1 User interfaces
    - 3.1.2 Hardware interfaces
    - 3.1.3 Software interfaces
    - 3.1.4 Communications interfaces
  - 3.2 Functional requirements
    - 3.2.1 Stimulus 1
      - 3.2.1.1 Functional requirement 1.1
      - .
      - .
      - .
      - 3.2.1.*n* Functional requirement 1.*n*
    - 3.2.2 Stimulus 2
      - .
      - .
      - .
      - 3.2.2.*m* Stimulus *m*
      - 3.2.2.1 Functional requirement *m.1*
      - .
      - .
      - .
      - 3.2.2.*m.n* Functional requirement *m.n*
  - 3.3 Performance requirements
  - 3.4 Design constraints
  - 3.5 Software system attributes
  - 3.6 Other requirements

## A.7 Template of SRS Section 3 organized by functional hierarchy

- 3. Specific requirements
  - 3.1 External interface requirements
    - 3.1.1 User interfaces
    - 3.1.2 Hardware interfaces
    - 3.1.3 Software interfaces
    - 3.1.4 Communications interfaces
  - 3.2 Functional requirements
    - 3.2.1 Information flows
      - 3.2.1.1 Data flow diagram 1
        - 3.2.1.1.1 Data entities
        - 3.2.1.1.2 Pertinent processes
        - 3.2.1.1.3 Topology
      - 3.2.1.2 Data flow diagram 2
        - 3.2.1.2.1 Data entities
        - 3.2.1.2.2 Pertinent processes
        - 3.2.1.2.3 Topology
      - .
      - .
      - 3.2.1.*n* Data flow diagram *n*

- 3.2.1.n.1 Data entities
- 3.2.1.n.2 Pertinent processes
- 3.2.1.n.3 Topology
- 3.2.2 Process descriptions
  - 3.2.2.1 Process 1
    - 3.2.2.1.1 Input data entities
    - 3.2.2.1.2 Algorithm or formula of process
    - 3.2.2.1.3 Affected data entities
  - 3.2.2.2 Process 2
    - 3.2.2.2.1 Input data entities
    - 3.2.2.2.2 Algorithm or formula of process
    - 3.2.2.2.3 Affected data entities
  - .
  - .
  - 3.2.2.m Process *m*
    - 3.2.2.m.1 Input data entities
    - 3.2.2.m.2 Algorithm or formula of process
    - 3.2.2.m.3 Affected data entities
- 3.2.3 Data construct specifications
  - 3.2.3.1 Construct 1
    - 3.2.3.1.1 Record type
    - 3.2.3.1.2 Constituent fields
  - 3.2.3.2 Construct 2
    - 3.2.3.2.1 Record type
    - 3.2.3.2.2 Constituent fields
  - .
  - .
  - 3.2.3.p Construct *p*
    - 3.2.3.p.1 Record type
    - 3.2.3.p.2 Constituent fields
- 3.2.4 Data dictionary
  - 3.2.4.1 Data element 1
    - 3.2.4.1.1 Name
    - 3.2.4.1.2 Representation
    - 3.2.4.1.3 Units/Format
    - 3.2.4.1.4 Precision/Accuracy
    - 3.2.4.1.5 Range
  - 3.2.4.2 Data element 2
    - 3.2.4.2.1 Name
    - 3.2.4.2.2 Representation
    - 3.2.4.2.3 Units/Format
    - 3.2.4.2.4 Precision/Accuracy
    - 3.2.4.2.5 Range
  - .
  - .
  - 3.2.4.q Data element *q*
    - 3.2.4.q.1 Name
    - 3.2.4.q.2 Representation
    - 3.2.4.q.3 Units/Format
    - 3.2.4.q.4 Precision/Accuracy
    - 3.2.4.q.5 Range

- 3.3 Performance requirements
- 3.4 Design constraints
- 3.5 Software system attributes
- 3.6 Other requirements

## A.8 Template of SRS Section 3 showing multiple organizations

- 3. Specific requirements
  - 3.1 External interface requirements
    - 3.1.1 User interfaces
    - 3.1.2 Hardware interfaces
    - 3.1.3 Software interfaces
    - 3.1.4 Communications interfaces
  - 3.2 Functional requirements
    - 3.2.1 User class 1
      - 3.2.1.1 Feature 1.1
        - 3.2.1.1.1 Introduction/Purpose of feature
        - 3.2.1.1.2 Stimulus/Response sequence
        - 3.2.1.1.3 Associated functional requirements
      - 3.2.1.2 Feature 1.2
        - 3.2.1.2.1 Introduction/Purpose of feature
        - 3.2.1.2.2 Stimulus/Response sequence
        - 3.2.1.2.3 Associated functional requirements
      - .
      - .
      - .
      - 3.2.1.m Feature 1.m
        - 3.2.1.m.1 Introduction/Purpose of feature
        - 3.2.1.m.2 Stimulus/Response sequence
        - 3.2.1.m.3 Associated functional requirements
    - 3.2.2 User class 2
      - .
      - .
      - .
    - .
    - .
    - .
    - 3.2.n User class n
      - .
      - .
      - .
  - 3.3 Performance requirements
  - 3.4 Design constraints
  - 3.5 Software system attributes
  - 3.6 Other requirements

## Annex B

(informative)

### Guidelines for compliance with IEEE/EIA 12207.1-1997

#### B.1 Overview

The Software Engineering Standards Committee (SESC) of the IEEE Computer Society has endorsed the policy of adopting international standards. In 1995, the international standard, ISO/IEC 12207, Information technology—Software life cycle processes, was completed. The standard establishes a common framework for software life cycle processes, with well-defined terminology, that can be referenced by the software industry.

In 1995 the SESC evaluated ISO/IEC 12207 and decided that the standard should be adopted and serve as the basis for life cycle processes within the IEEE Software Engineering Collection. The IEEE adaptation of ISO/IEC 12207 is IEEE/EIA 12207.0-1996. It contains ISO/IEC 12207 and the following additions: improved compliance approach, life cycle process objectives, life cycle data objectives, and errata.

The implementation of ISO/IEC 12207 within the IEEE also includes the following:

- IEEE/EIA 12207.1-1997, IEEE/EIA Guide for Information Technology—Software life cycle processes—Life cycle data;
- IEEE/EIA 12207.2-1997, IEEE/EIA Guide for Information Technology—Software life cycle processes—Implementation considerations; and
- Additions to 11 SESC standards (i.e., IEEE Stds 730, 828, 829, 830, 1012, 1016, 1058, 1062, 1219, 1233, 1362) to define the correlation between the data produced by existing SESC standards and the data produced by the application of IEEE/EIA 12207.1-1997.

NOTE—Although IEEE/EIA 12207.1-1997 is a guide, it also contains provisions for application as a standard with specific compliance requirements. This annex treats 12207.1-1997 as a standard.

#### B.1.1 Scope and purpose

Both IEEE Std 830-1998 and IEEE/EIA 12207.1-1997 place requirements on a Software Requirements Description Document. The purpose of this annex is to explain the relationship between the two sets of requirements so that users producing documents intended to comply with both standards may do so.

#### B.2 Correlation

This clause explains the relationship between IEEE Std 830-1998 and IEEE/EIA 12207.0-1996 and IEEE/EIA 12207.1-1997 in the following areas: terminology, process, and life cycle data.

##### B.2.1 Terminology correlation

Both this recommended practice and IEEE/EIA 12207.0-1996 have similar semantics for the key terms of software, requirements, specification, supplier, developer, and maintainer. This recommended practice uses

the term “customer” where IEEE/EIA 12207.0-1996 uses “acquirer,” and this recommended practice uses “user” where IEEE/EIA 12207.0-1996 uses “operator.”

### B.2.2 Process correlation

IEEE/EIA 12207.0-1996 uses a process-oriented approach for describing the definition of a set of requirements for software. This recommended practice uses a product-oriented approach, where the product is a Software Requirements Description (SRD). There are natural process steps, namely the steps to create each portion of the SRD. These may be correlated with the process requirements of IEEE/EIA 12207.0-1996. The difference is that this recommended practice is focused on the development of software requirements whereas IEEE/EIA 12207.0-1996 provides an overall life cycle view and mentions Software Requirements Analysis as part of its Development Process. This recommended practice provides a greater level of detail on what is involved in the preparation of an SRD.

### B.2.3 Life cycle data correlation

IEEE/EIA 12207.0-1996 takes the viewpoint that the software requirements are derived from the system requirements. Therefore, it uses the term, “description” rather than “specification” to describe the software requirements. In a system in which software is a component, each requiring its own specification, there would be a System Requirements Specification (SRS) and one or more SRDs. If the term Software Requirements Specification had been used, there would be a confusion between an SRS referring to the system or software requirements. In the case where there is a stand-alone software system, IEEE/EIA 12207.1-1997 states “If the software is a stand-alone system, then this document should be a specification.”

## B.3 Content mapping

This clause provides details bearing on a claim that an SRS complying with this recommended practice would also achieve “document compliance” with the SRD described in IEEE/EIA 12207.1-1997. The requirements for document compliance are summarized in a single row of Table 1 of IEEE/EIA 12207.1-1997. That row is reproduced in Table B.1 of this recommended practice.

**Table B.1—Summary of requirements for an SRD  
excerpted from Table 1 of IEEE/EIA 12207.1-1997**

Information item	IEEE/EIA 12207.0-1996 Clause	Kind	IEEE/EIA 12207.1-1997 Clause	References
Software Requirements Description	5.1.1.4, 5.3.4.1, 5.3.4.2	Description (See note for 6.22.1 of IEEE/EIA 12207.1-1997.)	6.22	IEEE Std 830-1998; EIA/IEEE J-STD-016, F.2.3, F.2.4; MIL-STD 961D. Also see ISO/IEC 5806, 5807, 6593, 8631, 8790, and 11411 for guidance on use of notations.

The requirements for document compliance are discussed in the following subclauses:

- B.3.1 discusses compliance with the information requirements noted in column 2 of Table B.1 as prescribed by 5.1.1.4, 5.3.4.1, and 5.3.4.2 of IEEE/EIA 12207.0-1996.

- B.3.2 discusses compliance with the generic content guideline (the “kind” of document) noted in column 3 of Table B.1 as a “description”. The generic content guidelines for a “description” appear in 5.1 of IEEE/EIA 12207.1-1997.
- B.3.3 discusses compliance with the specific requirements for a Software Requirements Description noted in column 4 of Table B.1 as prescribed by 6.22 of IEEE/EIA 12207.1-1997.
- B.3.4 discusses compliance with the life cycle data objectives of Annex H of IEEE/EIA 12207.0-1996 as described in 4.2 of IEEE/EIA 12207.1-1997.

### **B.3.1 Compliance with information requirements of IEEE/EIA 12207.0-1996**

The information requirements for an SRD are those prescribed by 5.1.1.4, 5.3.4.1, and 5.3.4.2 of IEEE/EIA 12207.0-1996. The requirements are substantively identical to those considered in B.3.3 of this recommended practice.

### **B.3.2 Compliance with generic content guidelines of IEEE/EIA 12207.1-1997**

According to IEEE/EIA 12207.1-1997, the generic content guideline for an SRD is generally a description, as prescribed by 5.1 of IEEE/EIA 12207.1-1997. A complying description shall achieve the purpose stated in 5.1.1 and include the information listed in 5.1.2 of IEEE/EIA 12207.1-1997.

The purpose of a description is:

IEEE/EIA 12207.1-1997, subclause 5.1.1: Purpose: Describe a planned or actual function, design, performance, or process.

An SRD complying with this recommended practice would achieve the stated purpose.

Any description or specification complying with IEEE/EIA 12207.1-1997 shall satisfy the generic content requirements provided in 5.1.2 of that standard. Table B.2 of this recommended practice lists the generic content items and, where appropriate, references the clause of this recommended practice that requires the same information.

**Table B.2—Coverage of generic description requirements by IEEE Std 830-1998**

IEEE/EIA 12207.1-1997 generic content	Corresponding clauses of IEEE Std 830-1998	Additions to requirements of IEEE Std 830-1998
a) Date of issue and status	—	Date of issue and status shall be provided.
b) Scope	5.1.1 Scope	—
c) Issuing organization	—	Issuing organization shall be identified.
d) References	5.1.4 References	—
e) Context	5.1.2 Scope	—
f) Notation for description	4.3 Characteristics of a good SRS	—
g) Body	5. The parts of an SRS	—
h) Summary	5.1.1. Overview	—
i) Glossary	5.1.3 Definitions	—
j) Change history	—	Change history for the SRD shall be provided or referenced.

### B.3.3 Compliance with specific content requirements of IEEE/EIA 12207.1-1997

The specific content requirements for an SRD in IEEE/EIA 12207.1-1997 are prescribed by 6.22 of IEEE/EIA 12207.1-1997. A compliant SRD shall achieve the purpose stated in 6.22.1 of IEEE/EIA 12207.1-1997.

The purpose of the SRD is:

IEEE/EIA 12207.1-1997, subclause 6.22.1: Purpose: Specify the requirements for a software item and the methods to be used to ensure that each requirement has been met. Used as the basis for design and qualification testing of a software item.

An SRS complying with this recommended practice and meeting the additional requirements of Table B.3 of this recommended practice would achieve the stated purpose.

An SRD compliant with IEEE/EIA 12207.1-1997 shall satisfy the specific content requirements provided in 6.22.3 and 6.22.4 of that standard. Table B.3 of this recommended practice lists the specific content items and, where appropriate, references the clause of this recommended practice that requires the same information.

An SRD specified according the requirements stated or referenced in Table B.3 of this recommended practice shall be evaluated considering the criteria provided in 5.3.4.2 of IEEE/EIA 12207.0-1996.

**Table B.3—Coverage of specific SRD requirements by IEEE Std 830-1998**

IEEE/EIA 12207.1-1997 specific content	Corresponding clauses of IEEE Std 830-1998	Additions to requirements of IEEE Std 830-1998
a) Generic description information	See Table B.2	—
b) System identification and overview	5.1.1 Scope	—
c) Functionality of the software item including: – Performance requirements – Physical characteristics – Environmental conditions	5.3.2 Functions 5.3.3 Performance requirements	Physical characteristics and environmental conditions should be provided.
d) Requirements for interfaces external to software item	5.3.1 External interfaces	—
e) Qualification requirements	—	The requirements to be used for qualification testing should be provided (or referenced).
f) Safety specifications	5.2.4 Constraints	—
g) Security and privacy specifications	5.3.6.3 Security	—
h) Human-factors engineering requirements	5.2.3 User characteristics 5.2.1.2 User interfaces	—
i) Data definition and database requirements	5.3.4 Logical data base requirements	—
j) Installation and acceptance requirements at operation site	5.2.1.8 Site adaptation requirements	Installation and acceptance requirements at operation site
k) Installation and acceptance requirements at maintenance site	—	Installation and acceptance requirements at maintenance site
l) User documentation requirements	—	User documentation requirements
m) User operation and execution requirements	5.2.1.7 Operations	User execution requirements

**Table B.3—Coverage of specific SRD requirements by IEEE Std 830-1998 (continued)**

<b>IEEE/EIA 12207.1-1997 specific content</b>	<b>Corresponding clauses of IEEE Std 830-1998</b>	<b>Additions to requirements of IEEE Std 830-1998</b>
n) User maintenance requirements	5.3.6.4 Maintainability	—
o) Software quality characteristics	5.3.6 Software system attributes	—
p) Design and implementation constraints	5.2.4 Constraints	—
q) Computer resource requirements	5.3.3 Performance requirements	Computer resource requirements
r) Packaging requirements	—	Packaging requirements
s) Precedence and criticality of requirements	5.2.6 Apportioning of requirements	—
t) Requirements traceability	4.3.8 Traceable	—
u) Rationale	5.2.5 Assumptions and dependencies	—
Items a) through f) below are from 6.22.4	—	Support the life cycle data objectives of Annex H of IEEE/EIA 12207.0-1996
a) Support the life cycle data objectives of Annex H of IEEE/EIA 12207.0-1996	4.3 Characteristics of a good SRS	—
b) Describe any function using well-defined notation	4.3 Characteristics of a good SRS	—
c) Define no requirements that are in conflict	4.3 Characteristics of a good SRS	—
d) User standard terminology and definitions	5.1.3 Definition	—
e) Define each unique requirement one to prevent inconsistency	4.3 Characteristics of a good SRS	—
f) Uniquely identify each requirement	4.3 Characteristics of a good SRS	—

### B.3.4 Compliance with life cycle data objectives

In addition to the content requirements, life cycle data shall be managed in accordance with the objectives provided in Annex H of IEEE/EIA 12207.0-1996.

## B.4 Conclusion

The analysis suggests that any SRS complying with this recommended practice and the additions shown in Table B.2 and Table B.3 also complies with the requirements of an SRD in IEEE/EIA 12207.1-1997. In addition, to comply with IEEE/EIA 12207.1-1997, an SRS shall support the life cycle data objectives of Annex H of IEEE/EIA 12207.0-1996.

# On Non-Functional Requirements

Martin Glinz

*Department of Informatics, University of Zurich, Switzerland*

*glinz@ifi.uzh.ch*

## Abstract

*Although the term ‘non-functional requirement’ has been in use for more than 20 years, there is still no consensus in the requirements engineering community what non-functional requirements are and how we should elicit, document, and validate them. On the other hand, there is a unanimous consensus that non-functional requirements are important and can be critical for the success of a project.*

*This paper surveys the existing definitions of the term, highlights and discusses the problems with the current definitions, and contributes concepts for overcoming these problems.*

## 1. Introduction

If you want to trigger a hot debate among a group of requirements engineering people, just let them talk about non-functional requirements. Although this term has been in use for more than two decades, there is still no consensus about the nature of non-functional requirements and how to document them in requirements specifications.

This paper is an attempt to work out and discuss the problems that we have with the notion of non-functional requirements and to contribute concepts for overcoming these problems. The focus is on system (or product) requirements; the role of non-functional requirements in the software process is not discussed [16].

The paper is organized as follows. Section 2 surveys typical definitions for the terms ‘functional requirement’ and ‘non-functional requirement’. The problems with these definitions are discussed in Section 3. Section 4 presents concepts about how these problems can be overcome or at least alleviated. The paper ends with a discussion of these concepts.

## 2. Defining the term

In every current requirements classification (for example [7], [11], [12]), we find a distinction between

requirements concerning the functionality of a system and other requirements.

There is a rather broad consensus about how to define the term ‘*functional requirements*’. The existing definitions follow two threads that coincide to a large extent. In the first thread, the emphasis is on functions: a functional requirement specifies “a function that a system (...) must be able to perform” [6], “what the product must do” [18], “what the system should do” [20]. The second thread emphasizes behavior: functional requirements “describe the behavioral aspects of a system” [1]; behavioral requirements are “those requirements that specify the inputs (stimuli) to the system, the outputs (responses) from the system, and behavioral relationships between them; also called functional or operational requirements.” [3].

Wiegers as well as Jacobson, Rumbaugh and Booch try a synthesis: “A statement of a piece of required functionality or a behavior that a system will exhibit under specific conditions.” [21]; “A requirement that specifies an action that a system must be able to perform, without considering physical constraints; a requirement that specifies input/output behavior of a system.” [10].

There is only one semantic difference that may arise between the different definitions: timing requirements may be viewed as behavioral, while they are not functional. However, most publications in RE consider timing requirements to be performance requirements which in turn are classified as non-functional requirements.

On the other hand, there is no such consensus for *non-functional requirements*. Table 1 gives an overview of selected definitions from the literature or the web, which – in my opinion – are representative of the definitions that exist.

## 3. Where is the problem?

The problems that we currently have with the notion of non-functional requirements can be divided into *definition problems*, *classification problems* and *representation problems*.

**Table 1.** Definitions of the term ‘non-functional requirement(s)’ (listed in alphabetical order of sources)

Source	Definition
Antón [1]	Describe the nonbehavioral aspects of a system, capturing the properties and constraints under which a system must operate.
Davis [3]	The required overall attributes of the system, including portability, reliability, efficiency, human engineering, testability, understandability, and modifiability.
IEEE 610.12 [6]	Term is not defined. The standard distinguishes design requirements, implementation requirements, interface requirements, performance requirements, and physical requirements.
IEEE 830-1998 [7]	Term is not defined. The standard defines the categories functionality, external interfaces, performance, attributes (portability, security, etc.), and design constraints. Project requirements (such as schedule, cost, or development requirements) are explicitly excluded.
Jacobson, Booch and Rumbaugh [10]	A requirement that specifies system properties, such as environmental and implementation constraints, performance, platform dependencies, maintainability, extensibility, and reliability. A requirement that specifies physical constraints on a functional requirement.
Kotonya and Sommerville [11]	Requirements which are not specifically concerned with the functionality of a system. They place restrictions on the product being developed and the development process, and they specify external constraints that the product must meet.
Mylopoulos, Chung and Nixon [16]	“... global requirements on its development or operational cost, performance, reliability, maintainability, portability, robustness, and the like. (...) There is not a formal definition or a complete list of nonfunctional requirements.”
Ncube [17]	The behavioral properties that the specified functions must have, such as performance, usability.
Robertson and Robertson [18]	A property, or quality, that the product must have, such as an appearance, or a speed or accuracy property.
SCREEN Glossary [19]	A requirement on a service that does not have a bearing on its functionality, but describes attributes, constraints, performance considerations, design, quality of service, environmental considerations, failure and recovery.
Wieggers [21]	A description of a property or characteristic that a software system must exhibit or a constraint that it must respect, other than an observable system behavior.
Wikipedia: Non-Functional Requirements [22]	Requirements which specify criteria that can be used to judge the operation of a system, rather than specific behaviors.
Wikipedia: Requirements Analysis [23]	Requirements which impose constraints on the design or implementation (such as performance requirements, quality standards, or design constraints).

### 3.1. Definition problems

When analyzing the definitions in Table 1, we find not only terminological, but also major conceptual discrepancies. Basically, all definitions build on the following terms: *property* or *characteristic*, *attribute*, *quality*, *constraint*, and *performance*. However, there is no consensus about the concepts that these terms denote. There are also cases where the meaning is not clear, because terms are used without a definition or a clarifying example.

*Property* and *characteristic* seem to be used in their general meaning, i.e. they denote something that the system must have, which typically includes specific qualities such as usability or reliability, but excludes any functional quality. There is no consensus whether constraints also are properties: Jacobson, Booch and Rumbaugh [10] include them, others, e.g. Wieggers [21] and Antón [1] exclude them.

*Attribute* is a term that is used both with a broad and a narrow meaning. In IEEE 830-1998 [7], attributes are a collection of specific qualities, excluding perform-

ance and constraints. On the other hand, in the definition by Davis [3], every non-functional requirement is an attribute of the system.

Every requirement (including all functional ones) can be regarded as a *quality*, because, according to ISO 9000:2000 [8], quality is the “degree to which a set of inherent characteristics fulfils requirements”. Similarly, every requirement can be regarded as a *constraint*, because it constrains the space of potential solutions to those that meet this requirement.

Hence, in all definitions that mention the term *quality*, its meaning is restricted to a set of specific qualities other than functionality: usability, reliability, security, etc.

Correspondingly, it is clear that *constraint* in the context of non-functional requirements must have a restricted meaning. However, there is no consensus among the existing definitions what precisely this restriction should be. For example, IEEE 830-1998 [7] or Jacobson, Booch and Rumbaugh [10] restrict the meaning of constraint to design constraints and physi-

cal constraints. Others, for example the definition in the Wikipedia article on requirements analysis [23], do not treat constraints as a sub-category of non-functional requirements, but consider every non-functional requirement to be a constraint. Davis [3] does not mention constraints at all when discussing non-functional requirements. Robertson and Robertson [18] have a rather specific view of constraints: they consider operational, cultural, and legal constraints to be non-functional properties, whereas design constraints are regarded as a concept that is different from both functional and non-functional requirements.

*Performance* is treated as a quality or attribute in many definitions. Others, e.g. IEEE 830-1998 [7] and Wikipedia [23] consider it as a separate category.

Another discrepancy exists in the *scope* of non-functional requirements. Some definitions emphasize that non-functional requirements have, by definition, a *global scope*: “global requirements” (Mylopoulos, Chung and Nixon [16]), “overall attributes” (Davis [3]). Accordingly, proponents of this global view separate functional and non-functional requirements completely. For example, in the Volere template [18] they are documented in two separate top-level sections. On the other hand, Jacobson, Booch and Rumbaugh [10] emphasize that there are both local non-functional requirements (e.g. most performance requirements) and global requirements such as security or reliability.

Most definitions refer to *system requirements* (also called *product requirements*) only and exclude *project* and *process requirements* either explicitly [7] or implicitly. However, in the definition by Kotonya and Sommerville [11], project requirements are considered to be non-functional requirements.

Finally, the variety and divergence of concepts used in the definitions of non-functional requirements also lead to definitions containing elements that are obviously misconceived. For example, the SCREEN glossary [19] lists failure and recovery as non-functional properties, while behavior in the case of failure and recovery from failures are clearly functional issues. Ncube [17] even defines non-functional requirements as “the behavioral properties...”. Although this is more likely a severe typo than a misconception, the fact that this error went unnoticed in a PhD thesis illustrates the fuzziness of the current notions of non-functional requirements and the need for a clear and concise definition. The worst example is the definition in the Wikipedia article on non-functional requirements [22], which is so vague that it could mean almost anything.

## 3.2. Classification problems

When analyzing the definitions given in Table 1, we also find rather divergent concepts for sub-classifying

non-functional requirements. Davis [3] regards them as qualities and uses Boehm’s quality tree [2] as a sub-classification for non-functional requirements. The IEEE standard 830-1998 on Software Requirements Specifications [7] sub-classifies non-functional requirements into external interface requirements, performance requirements, attributes and design constraints, where the attributes are a set of qualities such as reliability, availability, security, etc. The IEEE Standard Glossary of Software Engineering Terminology [6] distinguishes functional requirements on the one hand and design requirements, implementation requirements, interface requirements, performance requirements, and physical requirements on the other. Sommerville [20] uses a sub-classification into product requirements, organizational requirements and external requirements.

More classification problems arise due to mixing three concepts that should better be separated. These are the concepts of *kind* (should a given requirement be regarded as a function, a quality, a constraint, etc.), *representation* (see below), and *satisfaction* (hard vs. soft requirements). For an in-depth discussion of this problem, see [5].

## 3.3. Representation problems

As long as we regard any requirement that describes a function or behavior as a functional requirement, the notion of non-functional requirements is *representation-dependent*. Consider the following example: A particular security requirement could be expressed as “The system shall prevent any unauthorized access to the customer data”, which, according to all definitions given in Table 1 is a non-functional requirement. If we represent this requirement in a more concrete form, for example as “The probability for successful access to the customer data by an unauthorized person shall be smaller than  $10^{-5}$ ”, this is still a non-functional requirement. However, if we refine the original requirement to “The database shall grant access to the customer data only to those users that have been authorized by their user name and password”, we have a functional requirement, albeit it is still a security requirement. In a nutshell, the kind of a requirement depends on the way we represent it.

A second representational problem is the lack of consensus *where to document* non-functional requirements. As discussed above, some authors recommend the documentation of functional and non-functional requirements in separate chapters of the software requirements specification. The Volere template [18] is a prominent example for this documentation style. In IEEE 830-1998 [7], seven of the eight proposed SRS templates also separate the functional requirements

completely from the non-functional ones. On the other hand, when documenting requirements in a use-case-oriented style according to the Rational Unified Process [10], non-functional requirements are attached to use case as far as possible. Only the remaining (global) non-functional requirements are documented separately as so-called supplementary requirements.

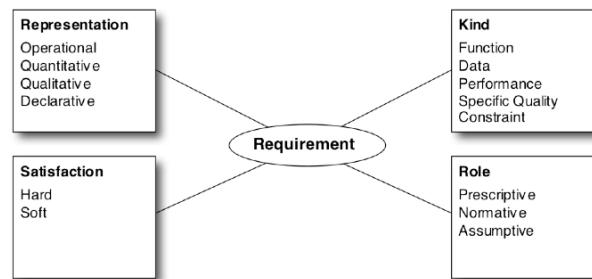
However, there are also cases where a non-functional requirement affects neither a single functional requirement nor the system as a whole. Instead, it pertains to a specific set of functional requirements. For example, some subset of the set of all use cases may need secure communication, while the other use cases do not. Such a case cannot be documented adequately with classic requirements specification templates and must be handled by setting explicit traceability links. As we will see in the next chapter, aspect-orientation provides a solution for this problem.

## 4. Elements of a solution

### 4.1. A faceted classification

In [5], I have proposed that the classification problems, and – in a radical sense – also the definition problems be overcome by introducing a faceted classification for requirements (Fig. 1) where the terms ‘functional requirement’ and ‘non-functional requirement’ no longer appear.

Separating the concepts of *kind*, *representation*, *satisfaction*, and *role* has several advantages, for example: (i) It is the representation of a requirement (not its kind!) that determines the way in which we can verify that the system satisfies a requirement (Table 2). (ii) Decoupling the kind and satisfaction facets reflects the fact that functional requirements are not always hard and non-functional requirements not always soft. (iii) The role facet allows a clear distinction of (prescriptive) system requirements and (normative or assumptive) domain requirements.



**Figure 1.** A faceted classification of requirements (from [5])

However, when using this classification it turned out that the idea of getting rid of the terms ‘functional

requirement’ and ‘non-functional requirement’ is too radical. When practicing requirements engineering, *there is a need* to distinguish functional concerns from other, “non-functional” concerns and there is also a need for a sub-classification of the “non-functional” concerns in a clear and comprehensible way. In the next section, such a definition is presented.

**Table 2.** Representation determines verification [5]

Representation	Type of verification
Operational	Review, test or formal verification
Quantitative	Measurement (at least on an ordinal scale)
Qualitative	No direct verification. May be done by subjective stakeholder judgment of deployed system, by prototypes or indirectly by goal refinement or derived metrics
Declarative	Review

### 4.2. A definition based on concerns

We define a taxonomy of terms that is based on the concept of *concerns*, which makes it independent of the chosen representation. We assume a requirements engineering context, where *system* is the entity whose requirements have to be specified.

Furthermore, the taxonomy concentrates on *system requirements*. As project and process requirements are conceptually different from system requirements, they should be distinguished at the root level and not in a sub-category such as non-functional requirements.

**DEFINITION.** A *concern* is a matter of interest in a system. A concern is a *functional* or *behavioral* concern if its matter of interest is primarily the expected behavior of a system or system component in terms of its reaction to given input stimuli and the functions and data required for processing the stimuli and producing the reaction. A concern is a *performance* concern if its matter of interest is timing, speed, volume or throughput. A concern is a *quality concern* if its matter of interest is a quality of the kind enumerated in ISO/IEC 9126 [9].

**DEFINITION.** The set of all requirements of a system is partitioned into *functional requirements*, *performance requirements*, *specific quality requirements*, and *constraints*.

A *functional requirement* is a requirement that pertains to a functional concern.

A *performance requirement* is a requirement that pertains to a performance concern.

A *specific quality requirement* is a requirement that pertains to a quality concern other than the quality of meeting the functional requirements.

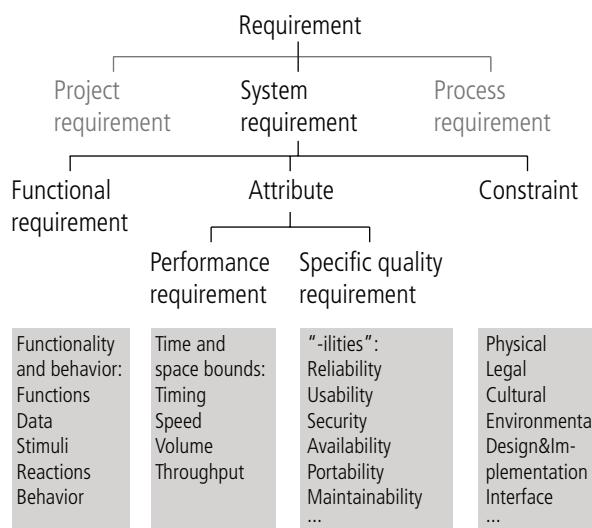
A *constraint* is a requirement that constrains the solution space beyond what is necessary for meeting

the given functional, performance, and specific quality requirements.

An *attribute* is a performance requirement or a specific quality requirement.

The taxonomy defined above is visualized in Figure 2. In my opinion, this structure is sufficient and useful both for theoretical and practical work. For persons who do not want to dispose of the term ‘non-functional requirement’, we can define this term additionally as follows.

**DEFINITION.** A *non-functional requirement* is an attribute of or a constraint on a system.



**Figure 2.** A concern-based taxonomy of requirements

*Performance* is a category of its own in our taxonomy (as well as in the IEEE standards [6] [7]) because performance requirements are typically treated separately in practice. This is probably due to the fact that measuring performance is not difficult *a priori*, while measuring other attributes is: there is a broad consensus to measure performance in terms of time, volume, and volume per unit of time. For any other attribute, there are no such generally agreed measures, which means that the task of eliciting specific qualities always implies finding an agreement among the stakeholders how to measure these qualities.

*Interfaces*, which are a separate category in the IEEE standards [6] [7], no longer appear in the terminology defined above. An interface requirement is classified according to the concern it pertains to as functional, performance, specific quality, or constraint.

Due to the systematic construction of our taxonomy, the task of *classifying* a given requirement becomes easier and less ambiguous. The often heard rule ‘What

the system does → functional requirement; How the system behaves → non-functional requirement’ [4] is too coarse and leads to mis-classifications when attributes and constraints are represented functionally or when they are very important<sup>1</sup>.

Table 3 gives the classification rules for the taxonomy laid out in Figure 2.

**Table 3.** Classification rules

No <sup>1</sup>	Question	Result
	Was this requirement stated because we need to specify...	
1	... some of the system’s behavior, data, input, or reaction to input stimuli – regardless of the way how this is done?	Functional
2	... restrictions about timing, processing or reaction speed, data volume, or throughput?	Performance
3	... a specific quality that the system or a component shall have?	Specific quality
4	... any other restriction about what the system shall do, how it shall do it, or any prescribed solution or solution element?	Constraint

<sup>1</sup> Questions must be applied in this order

### 4.3. Aspect-oriented representation

As soon as we structure the functional requirements specification systematically (by using a text template or by modeling requirements), the question about structuring the non-functional requirements comes up. Structuring attributes and constraints into sub-kinds according to a documentation template is helpful here, but clearly this is not enough; in particular when we have attributes and constraints that are neither completely local nor fully global, but pertain to some specific parts of the system.

An aspect-oriented representation of requirements, in particular of attributes and constraints, helps overcome this problem. A multi-dimensional separation of concerns [15] allows every concern to be modeled separately. Thus, all concerns are treated equally, which looks clean and elegant from an academic viewpoint. However, in practice, there is almost always a dominant concern, which is typically a functional one. This concern is crosscut by other concerns, both functional and non-functional ones.

In my research group, we have developed an aspect-oriented-extension for a hierarchical modeling lan-

<sup>1</sup> For example, in particle physics, the detectors of contemporary accelerators produce enormous amounts of data in real time. When asked a ‘what shall the system do’ question about the data processing software for such a detector, one of the first answers a physicist typically would give would be that the system must be able to cope with the data volume. However, this is a performance requirement.

guage [13], [14] where we identify a dominant functional concern and decompose the system model hierarchically according to the structure of this concern. All other concerns are modeled as aspects of this primary model. Aspect composition is supported by formal model weaving semantics.

Thus we can document attributes and constraints as separate entities, but, at the same time, attach them systematically to those elements in the primary model hierarchy where they apply. Global attributes and constraints are attached to the root of the decomposition hierarchy, while an attribute or constraint that restricts only some parts of the model is attached exactly to these parts by modeling join relationship from the aspect to the affected parts of the primary model.

## 5. Discussion

The analysis of the definitions given in Table 1 reveals the deficiencies of the current terminology. The *new definitions* proposed in this paper

- are more systematically constructed than any existing definitions that I am aware of,
- are representation-independent; i.e. the kind of a requirement is always the same, regardless of its representation,
- make it easier to classify a given requirement: with the classification criteria given in Table 3, the classification is much less ambiguous than with traditional definitions,
- better support the evolution of a requirements specification, because the classification of a requirement remains invariant under refinement and change as long as the concern to which the requirements pertains remains the same.

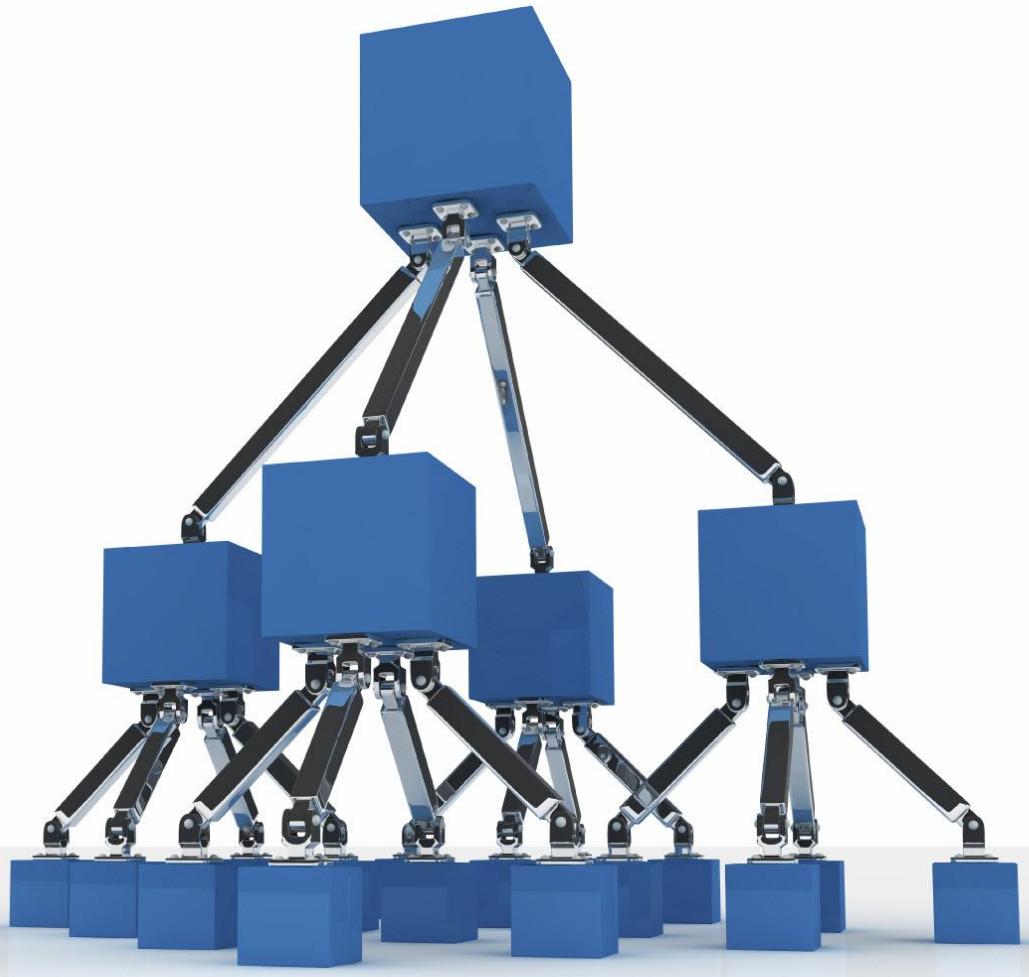
With an aspect-oriented documentation of attributes and constraints, the *documentation* and *traceability* problems of non-functional requirements are alleviated.

As this is mainly theoretical work, the *validation* of the theoretical soundness and usefulness of these ideas will be the extent to which other researchers find these ideas useful and adopt or build upon them.

The systematic exploration of the *practical usefulness* is a topic for further investigation.

## References

- [1] A. Antón (1997). *Goal Identification and Refinement in the Specification of Information Systems*. PhD Thesis, Georgia Institute of Technology.
- [2] B. Boehm et al. (1976). Quantitative Evaluation of Software Quality. *Proc. 2nd IEEE International Conference on Software Engineering*, 592-605.
- [3] A. Davis (1993). *Software Requirements: Objects, Functions and States*. Prentice Hall.
- [4] X. Franch (1998). Systematic Formulation of Non-Functional Characteristics of Software. *Proc. 3rd Int'l Conf. Requirements Engineering (ICRE'98)*, 174-181.
- [5] M. Glinz (2005). Rethinking the Notion of Non-Functional Requirements. *Proc. Third World Congress for Software Quality (3WCSQ 2005)*, Munich, Germany, Vol. II, 55-64.
- [6] IEEE (1990). *Standard Glossary of Software Engineering Terminology*. IEEE Standard 610.12-1990.
- [7] IEEE (1998). *IEEE Recommended Practice for Software Requirements Specifications*. IEEE Std. 830-1998.
- [8] ISO 9000 (2000). *Quality Management Systems – Fundamentals and Vocabulary*. International Organization for Standardization.
- [9] ISO/IEC 9126-1 (2001). *Software Engineering – Product Quality – Part 1: Quality Model*. International Organization for Standardization.
- [10] I. Jacobson, G. Booch, and J. Rumbaugh (1999). *The Unified Software Development Process*. Reading, Mass.: Addison Wesley.
- [11] G. Kotonya, I. Sommerville (1998). *Requirements Engineering: Processes and Techniques*. John Wiley & Sons.
- [12] A. van Lamsweerde (2001). Goal-Oriented Requirements Engineering: A Guided Tour. *Proc. 5th International Symposium on Requirements Engineering (RE'01)*, Toronto, 249-261.
- [13] S. Meier, T. Reinhard, C. Seybold, and M. Glinz (2006). Aspect-Oriented Modeling with Integrated Object Models. *Proc. Modellierung 2006*, Innsbruck, Austria, 129-144.
- [14] S. Meier, T. Reinhard, R. Stoiber, M. Glinz (2007). Modeling and Evolving Crosscutting Concerns in ADORA. *Proc. Early Aspects at ICSE: Workshop in Aspect-Oriented Requirements Engineering and Architecture Design*.
- [15] A. Moreira, A. Rashid, and J. Araújo (2005). Multi-Dimensional Separation of Concerns in Requirements Engineering. *Proc. 13th IEEE International Requirements Engineering Conference (RE'05)*, 285-296.
- [16] J. Mylopoulos, L. Chung, B. Nixon (1992). Representing and Using Nonfunctional Requirements: A Process-Oriented Approach. *IEEE Transactions on Software Engineering* **18**, 6 (June 1992), 483-497.
- [17] C. Ncube (2000). *A Requirements Engineering Method for COTS-Based Systems Development*. PhD Thesis, City University London.
- [18] S. Robertson and J. Robertson (1999). *Mastering the Requirements Process*. ACM Press.
- [19] SCREEN (1999). *Glossary of EU SCREEN Project*. <http://cordis.europa.eu/infowin/acts/rus/projects/screen/glossary/glossary.htm> (visited 2007-07-05)
- [20] I. Sommerville (2004). *Software Engineering*, Seventh Edition. Pearson Education.
- [21] K. Wiegers (2003). *Software Requirements*, 2nd edition. Microsoft Press.
- [22] Wikipedia: *Non-Functional Requirements* [http://en.wikipedia.org/wiki/Non-functional\\_requirements](http://en.wikipedia.org/wiki/Non-functional_requirements) (visited 2007-07-05)
- [23] Wikipedia: *Requirements Analysis* [http://en.wikipedia.org/wiki/Requirements\\_analysis](http://en.wikipedia.org/wiki/Requirements_analysis) (visited 2007-07-05)



# Using statement-level templates to improve the quality of requirements

An Integrate white paper

October 2012



# Using statement-level templates to improve the quality of requirements

## Abstract

Much has been written on the subject of what constitutes an acceptable requirement statement, and tools are available to check the quality of a requirement in terms of grammar and vocabulary after it has been written. Using a statement-level template (or “boilerplate”) is a means of creating an acceptable requirement as it is written, rather than after the event.

There have been a number of papers published on using templates and structured language for improving requirements quality, and for relating textual requirements to formal specifications of various kinds. This paper focuses on the use of statement-level templates to aid in the expression of textual requirements in general, including stakeholder requirements at levels of abstraction above the specification. The aim of the approach is to provide grammatical structures that encapsulate the accepted rules for writing singular, unambiguous, verifiable requirements.

The paper surveys a number of tools for supporting statement-level templates, and describes some experiences of applying the use of templates to projects.

## Introduction

It is received wisdom in the discipline of requirements management that textual requirements should possess a number of characteristics, such as singularity, non-ambiguity and verifiability. A synthesis of such properties can be found in [1] and in more detail in [12]. A number of tools are available, such as LEXIOR [2], and RQA (Requirements Quality Analyzer) [3], that analyze textual requirements, and identify where they fail to possess such characteristics.

This paper reviews an approach to assuring that such characteristics are present in the requirements as they are written, rather than assessing them after the event. The approach encourages the author to select a pre-determined textual template [18] specific to the type of requirement to be expressed, and to fill out place-holders to instantiate it.

There is a body of published work in the area of requirements templates (or, more precisely, patterns) associated with formal specifications, which work is briefly reviewed below. The main motivation of such work is to make the specification of a system more approachable through the use of structured text that has a precise relationship with one or more underlying formal models.

The approached described in this paper is slightly different. We do not attempt to tie the templates to any underlying formal model, and argue that there are still significant benefits to be gained, as enumerated below. What's more, the

approach can address a very broad range of kinds of requirement at all levels of abstraction, including high-level stakeholder goals, aspirations and requirements.

Early work on the use of this kind of requirement template dates from 1998. Although not formally published, the basic approach can be found in [18] and at [4]. Each template is an English language sentence with place-holders for specific words that define the particular requirement. No constraints are placed on the kinds of words that can fill the place-holders; only encouragement to use consistent vocabulary.

More recent work [15], [19] has applied the approach in a very specific domain and attempted to quantify the benefits. Later development makes use of semantic parsing with underlying domain-specific ontologies to guide the author in the selection of appropriate terms, units and values.

The benefits of using this approach to author requirements include:

- *An aid in articulation.* Being able to draw upon a palette of classified templates helps in finding effective ways of expressing particular kinds of requirement or constraint.
- *Uniformity of grammar.* By always considering the reuse of existing templates, the same kinds of requirement will always be expressed in the same kinds of way, leading to a consistent use of language in expressing requirements.

- *Uniformity of vocabulary.* Each placeholder in a template can be controlled through application of ontology. For instance, if the place-holder is expected to contain a "mode of operation", the ontology can articulate the allowable vocabulary.
- *Ensuring essential characteristics.* The use of a template aids in assuring that the requirement statement is unambiguous, quantified, testable and other essential characteristics.
- *Easier identification of repeated and conflicting requirements.* When requirements are expressed in uniform way, automatic detection of potentially repeated or similar, requirements becomes possible. Likewise, candidates for contradictory requirements can be identified.
- *One-stop control over expression.* With effective tool support, global changes in the way requirements are expressed can be effected by just changing the template in one place. E.g. changing "will" to "shall".
- *Protection of classified information.* Often the only part of a classified requirement is the quantity associated with it – e.g. the speed of the vessel, the rate of fire. Using templates, the classification can be focused on just the restricted attributes, allowing the rest of the requirement to be viewed for non-restricted use.

## Terminology

We wish here to make a distinction between "templates", "boilerplates" and "patterns". The term "template" is most often used in the field of requirements management to refer to the heading structure of a requirements document. The use of such templates helps ensure that the authors consider the complete range of concerns when collating a set of requirement statements. Because of this use of the word "template", early work [18] in the subject of this paper coined the term "boilerplate" to refer to a grammatical structure for an individual textual requirement. In this paper, we use the term "statement-level template" to refer to this concept.

"Patterns" usually refer to something broader than an individual statement. Design patterns [21], for instance, are bundles of artefacts that frequent occur together when defining a particular kind of design. One could imagine requirements patterns as sets of related requirements that together provide a complete specification of a particular feature. This paper is not about such patterns.

## Related work

Existing work on the concept of requirements templates falls into three categories:

- The use of controlled English grammar to express properties that have an underlying formal semantics.
- The use of templates in tabular rather than textual form.
- The use of statement-level templates.

### Approaches related to formal models

Examples of the use of statement-level templates and patterns associated with formal notations include:

- (1999) *Finite state automata and Petri Nets.* In [6], specification patterns are related to a formal semantics based on finite state automata. This work is later extended by [7] to cover real-time aspects. Here the patterns are given projections into three different formalisms.
- (2001) *Safety.* Finite state automata in the safety domain are considered by [8].
- (2008) *Probabilistic verification techniques.* The work of [5] applies the concept of specification patterns to make it easier for authors to formulate probabilistic properties of systems, improving the acceptability of formal verification techniques.
- (2009) *Requirements Documents generation.* The use of formally declared requirements patterns, as in [13] allows semi-automatic techniques to create software requirements documents.

The frequently cited objective of using such templates or patterns is to allow the specification author to specify system properties in familiar language controlled in such a way that there is an underlying formal semantics. The advantages of this are clear: automated formal verification and model-checking techniques can be brought to bear on the specification despite the requirements being expressed in natural language. The most relevant drawback has to do with the difficulty to use a strongly formal approach to write requirements.

### Approaches that use tabular templates

Examples of approaches that use tabular templates or patterns without formal semantics are:

- *Planguage.* In [9], tabular templates with keywords are used to specify requirements rather than relying on grammatical sentences.

One of the key motivations for this is to ensure that an appropriate set of performance requirements and constraints are considered for each functional requirement, thus ensuring completeness. The consistent use of keywords also ensures uniform and unambiguous language.

- *Volere*. In [10], a form of tabular template is proposed called a “snow card”, which ensures that certain information, such as rationale and requirement type, is associated with each requirement statement.

These approaches are not primarily textual-requirement-centric, attempting to provide semi-formal representation schemas.

### Approaches based on statement-level templates

In this paper, we make no attempt to associate the statement-level templates with a formal semantics. We simply control the language used to reduce the risk of ambiguity or inconsistency, and to maintain certain important characteristics of requirements statements we focus on the expression of singular textual requirements. Nor do we address the completeness of sets or sub sets of related requirements.

Other work that is most similar to this approach includes:

- *(2002) Requirements Boilerplates*. The ideas presented in Chapter 7 of [18] are the most direct predecessor of this work.
- *(2009) Easy Approach to Requirements Syntax (EARS) and Adv-EARS*. In [15] and [16], a mapping of the grammatical constructs of English has been used to formalize requirements for a specific domain: aero-engine control systems.
- *(2010) The Requirements Specification Language (RSL) of CESAR*. The CESAR research project, funded by the European Union AREMIS program, reviewed work on the use of boilerplates [14] with a view to extending and applying the approach to a number of safety-critical domains, with discussions of how to formalise the approach using ontologies.

All three of these references will be covered in greater detail in the next section.

## Project Experience

The purpose of this section is to report on a number of projects that have defined and applied statement-level templates. Some of these projects were completed with no formal publication of their results; others do have related

publications, and these are cited; yet others are on-going experiments.

### Future Surface Combatant

In 1998, the concept of requirement statement templates was applied to a UK defence project in which requirements were being elicited for a naval warship, Future Surface Combatant (FSC). It was observed that requirement authors had difficulty in articulating certain kinds of requirement, such as timeliness and periodicity. Once shown a good example of how to do it, however, the difficulty was largely overcome. A palette of “boilerplate” requirements was established for the project containing examples of many kinds of requirement, carefully phrased to avoid ambiguity and to ensure proper quantification. (The term “boilerplate” was used to distinguish the concept from requirements document templates, which consist purely of a heading structure.)

The palette was organised into types of property or constraint, and each boilerplate consisted of a complete sentence with place-holder words that the author could replace with specific terms. The placeholders were named consistently across all the boilerplates, so that, for instance, “<stakeholder>” and “<unit of volume>” might appear in multiple boilerplates where the same concept was referenced.

The requirements writing process was defined as follows:

1. Consider what kind of requirement or property you wish to articulate.
2. Select the most appropriate boilerplate from the palette.
3. Fill out the place-holders in the boilerplate to form the specific requirement.

As enthusiasm for the concept grew, tool support was provided that supported the requirements author in steps 2 and 3. The tool encouraged the author to select to use consistent vocabulary across requirements by offering terms already used for the same placeholder in other requirements. Beyond this, no attempt at formalising an ontology or glossary was implemented.

With tool support, a requirement statement became, in effect, a boilerplate and a set of values assigned to the placeholders. By representing the requirements in this fashion, it became possible centrally to control and evolve the precise way in which different types of requirement were expressed. Simply by changing the global boilerplate, the text of all requirements that used the boilerplate was also updated.

It also became possible to abstract from the requirements all the values that had been used for the placeholders. For instance, the placeholder "<function>" gave a list of all the functions mentioned, and "<user>" gave a list of all users mentioned.

Table 1 show some examples of the boilerplates that were used. In the left-hand column are the

boilerplates hierarchically organised into clauses that may be combined to construct complete requirement statements. The right-hand column shows two things: the classification of the combined clauses, and a complete example instantiation.

**Table 1: Example boilerplates from FSC**

<i>BOILERPLATE</i>	<i>CLASSIFICATION &amp; EXAMPLE INSTANTIATION</i>
While <operational condition>,	<b>Mode</b> While <i>disconnected from a source of power, ...</i>
... the <user> shall able to <capability>	<b>Mode/Capability</b> While <i>disconnected from a source of power, the photographer shall able to capture still images</i>
... at least <quantity> times per <time unit>	<b>Mode/Capability/Rapidity (Maximise, Exceed)</b> While <i>disconnected from a source of power the photographer shall able to capture still images at least 10 times per minute</i>
... for a sustained period of at least <quantity> <time unit>	<b>Mode/Capability/Rapidity/Sustainability (Maximise, Exceed)</b> While <i>disconnected from a source of power the photographer shall able to capture still images at least 10 times per minute for a sustained period of at least 4 hours</i>
... the <system function> shall able to <action> <entity>	<b>Mode/Function</b> While <i>in winds up to Beaufort scale 8 the missile launcher shall able to fire missiles</i>
... within <quantity> <time unit>s from <event>	<b>Mode/Function/Timeliness (Minimise, Do not exceed)</b> While <i>in winds up to Beaufort scale 8 the missile launcher shall able to fire missiles within 5 seconds from receipt of warning signal</i>

The application of this approach in the project had a positive effect on the confidence of the requirements authors in expressing the requirements. At the end of the exercise, the project had developed a palette of about 120 boilerplates for different kinds of requirement. This palette represented an intellectual asset to the organisation to pass on to future projects.

### Baggage Handling

A similar approach to the above was applied by the British Airports Authority (BAA) in gathering

requirements for the London Heathrow Terminal 5 Baggage Handling System, but without specific tool support. Their boilerplates were referred to as "statement level templates" (as opposed to document templates), and templates were provided at the stakeholder level for capabilities, constraints, assumptions and definitions. A different style of template is used at the application-specific level, where more structure can be prescribed.

Table 2 shows an example capability template used in the baggage handling project.

**Table 2: Example capability template from baggage handling**

<b>Templates</b>
<Actor> shall be able to <Act> <Action_Subject> whilst <Qualification> <Qualification_Subject>.
<Actor> shall be able to <Dispatch_Act> <Action_Subject> from <Dispatch_Object>.
where
<Actor> ..... the user (person or external system) who interacts with the system to be developed in order to perform the Act.
<Act> ..... the act the actor needs to perform to achieve some business goal.
<Action_Subject> ..... the part of the application domain which will be affected by the Act.
<Qualification> ..... a description of the circumstances under which the Actor should be able to perform the specified Act.
<Qualification_Subject> ..... the part of the application domain over which the Qualification is defined.
<Dispatch_Act> ..... the dispatch act the Actor needs to perform to achieve some business goal. (A dispatch act engages in a different grammatical construct, because it is expected to require an indirect object, the Dispatch_Object).
<Dispatch_Object>..... the location from which the dispatch takes place.

As an example how the capability template was used, an original, poorly expressed user requirement was re-expressed as shown in Table 3. By being constrained to use provided templates, requirements authors are guided into

expressing requirements in a uniform style at the appropriate level of abstraction. Hence the emphasis is shifted from solution to capability.

**Table 3: Re-expression of user requirement**

<b>Original requirement</b>
Each check-in desk shall be equipped with weigh, label and take-away conveyors, feeding collector conveyors.
<b>Re-expression as a set of capability requirements</b>
<i>Check-in clerks shall be able to weigh std baggage items whilst stationed at the check-in desk.</i>
<i>Check-in clerks shall be able to label std baggage items whilst stationed at the check-in desk.</i>
<i>Check-in clerks shall be able to despatch to a take-away conveyor std baggage items from the check-in desk.</i>
<i>Check-in clerks shall be able to despatch to a feeding collector conveyor std baggage items from the check-in desk.</i>

Requirement author reaction to this approach was mixed: some complained that they felt the templates restricted their freedom of expression. However, the project managers persisted with the approach because they felt the benefits of uniformity of expression outweighed the constraints.

## Easy Approach to Requirements Syntax (EARS)

Reported in [15] and [19] are some experiments in apply the statement-level template approach to the specification of properties of aero-engine

control systems. Because the domain is so specific, and applied at a technical system level rather than user level, the experiment was able to use just 4 statement templates to cover the majority of requirements. These are shown in Table 4.

**Table 4: Templates for control systems**

<b>Ubiquitous requirements</b> The <system name> shall <system response>	<b>Example</b> The <i>control system</i> shall <i>prevent engine overspeed</i>
<b>Event-driven requirements</b> WHEN <optional preconditions/trigger> the <system name> shall <system response>	<b>Example</b> WHEN <i>continuous ignition is commanded by the aircraft</i> , the <i>control system</i> shall <i>switch on continuous ignition</i>
<b>Unwanted behaviours</b> IF <optional preconditions/trigger>, THEN the <system name> shall <system response>	<b>Example</b> IF <i>the computed airspeed fault flag is set</i> , THEN the <i>control system</i> shall <i>use modelled airspeed</i>
<b>State-driven requirements</b> WHILE <in a specific state> the <system name> shall <system response>	<b>Example</b> WHILE <i>the aircraft is in-flight</i> , the <i>control system</i> shall <i>Maintain engine fuel flow above XXlbs/sec</i>

Note the use here of capitalised key words to make it clear exactly which template is in use.

Reference [15] reports an improvement in requirement quality when judged by a small set of criteria. The later reference [19] evolves the approach, adds some more templates, and uses more refined criteria for assessing the benefits accrued.

## Context RDS

Context RDS is an experimental toolkit that develops the concept of statement templates strongly towards the use of domain-specific ontologies. The tool is being applied to projects within the aero-engine division of Rolls-Royce where templates are being used to frame requirements for engine control systems, a domain that lends itself well to stylised requirements.

In RDS, statement-level templates contain placeholders that may be of the following types:

- Process Defined
- Value by Reference
- User Defined Term
- User Defined Text

Placeholders that obtain their value by reference can only be populated by terms of one or more type, defined in a project-specific process model. For example, "While <Condition> is....." has a placeholder that can be constrained to allow references to atomic condition or complex conditions; atomic conditions have further constraints. User Defined Terms have templates with similar placeholders, thus allow terms to reference other terms.

Figure 1 shows an example template structure, and figure 2 shows an example instantiation.

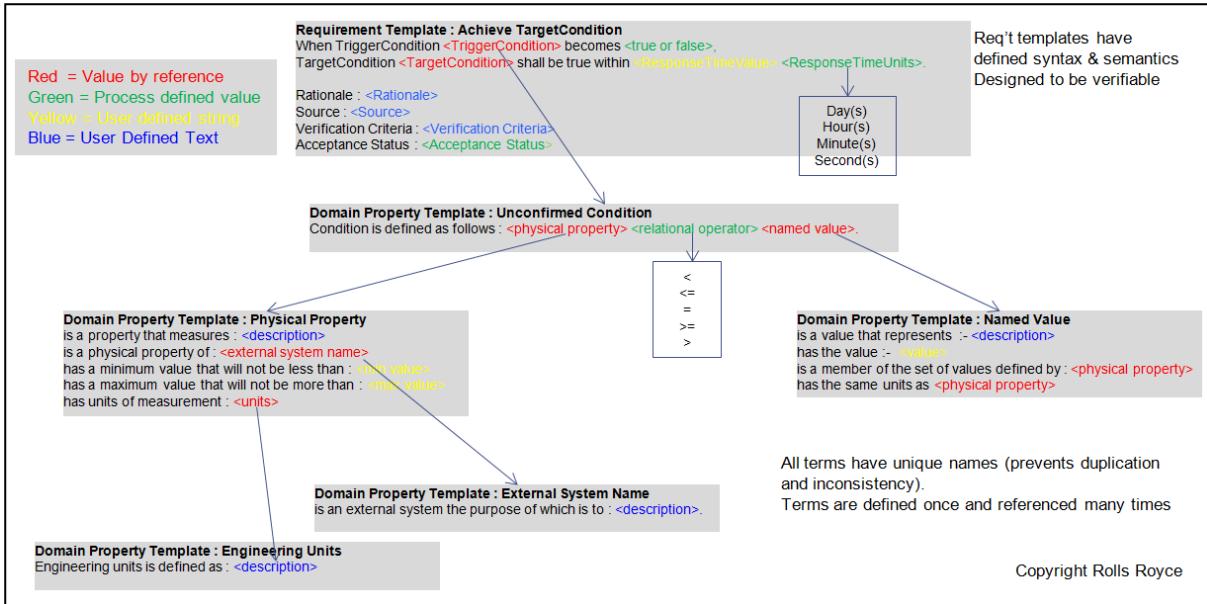


Figure 1: A Context RDS Template Structure

The configuration of the tool constrains how authors are able to instantiate the templates. This ensures, for instance, the consistent use of terms and units, and prevents nonsense requirements created by meaningless instantiation.

It is considered too early in the experience of using Context RDS to be able to report specific feedback. However, it is evident that considerably more up-front work and cost is engendered by this approach; it is hoped that the investment will return dividends later in the project.

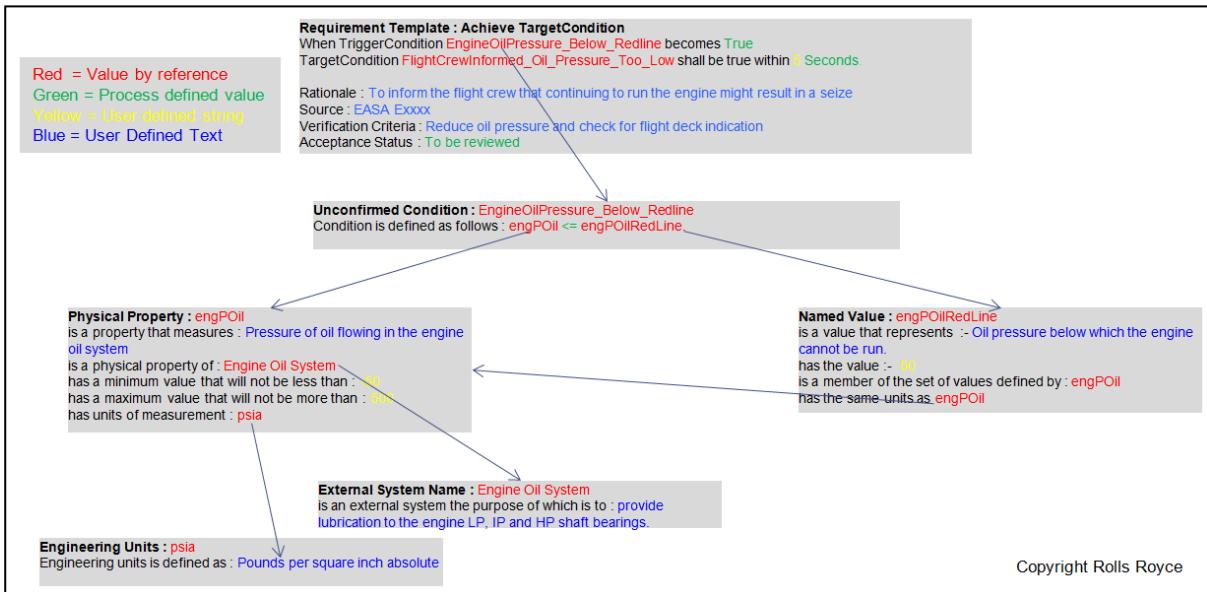


Figure 2: An instantiated template in Context RDS

### The Requirements Specification Language (RSL) of the CESAR project

A main goal of the European-funded CESAR project was to consider way of improving the development process for a number of safety-critical domains, such as aerospace and rail. In

their analysis of opportunities in requirements engineering, the use of boilerplates was identified as candidate for bringing about improvement.

A key CESAR report [14] considered two things together: a Requirements Specification Language (RSL) largely based on the boilerplate concept,

and a Requirements Meta Model (RMM) consisting of a general and domain-specific ontology for supporting requirements vocabulary.

The report notes that such ontology provides the semantic base for which requirements can be analysed, validated and verified.

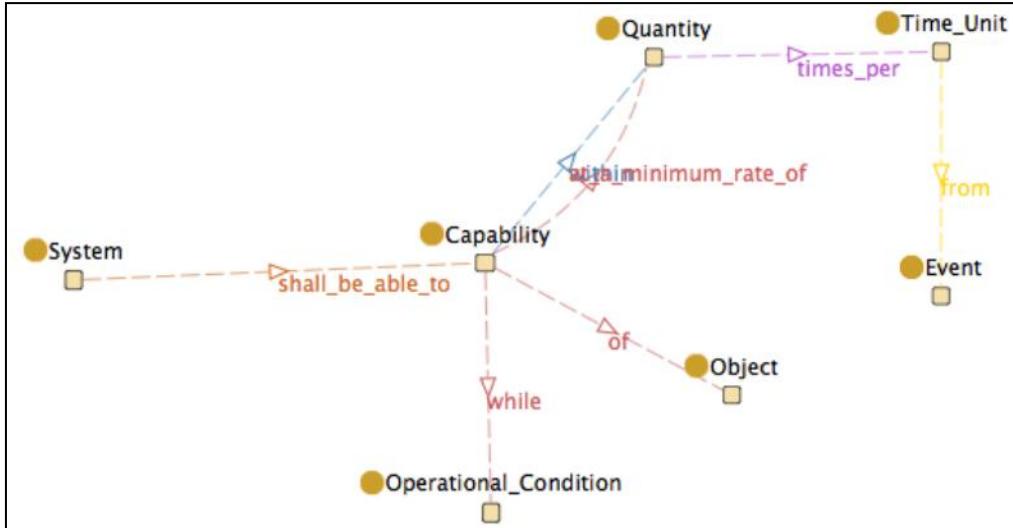


Figure 3: An example ontology underpinning requirement statements

Figure 3 shows an example ontology, from which a variety of statement-level templates could be generated by traversing parts of the structure as appropriate. The nodes are typed entities that form the placeholders, and the arcs describe the relationships between those entities.

### Requirements Authoring Tool (RAT)

Another project relates to a recently released product, the Requirements Authoring Tool (RAT) from The Reuse Company [17]. RAT allows the user to start typing a requirement, and simultaneously determines which of a palette of statement-level templates are applicable, guiding the user on permissible terminology. If the requirement that the user is typing eventually matches a template, the user is informed of the fact.

To achieve this, it combines three things:

- A set of pre-determined templates.
- A semantic parsing engine.
- A strong underlying domain-specific ontology.

The combination of these properties allows the requirements author to write requirements in an assisted way, as a counterpart to the “fill the gap” approach to placeholders. The possibility of not selecting a-priori a pattern before typing a requirement leads to the tool being considered as an assistant, as well as a coach, and users may feel their creativity is less constrained.

At present, the tool comes equipped with about 70 statement-level templates that can be combined in various ways. RAT allows the

evolution of the set of templates and associated ontology to be managed as part of the enterprise’s knowledge base. In doing this, it introduces the specialist role of “knowledge manager”, in addition to that of “author”. If authors find that the tool is able to match templates to the requirements they are trying to express, they can apply to the knowledge manager to introduce new ones, with associated terms. Such a role requires an understanding of the purpose of ontology, and how to use the tool to define it.

### Future Directions

This section presents some ideas for the future development of the approach.

#### Extension to artefacts other than requirements

Works so far has focused on controlling the expression of requirements. Other artefacts closely related to requirements may benefit from the same approach. For instance:

- Descriptions of V&V activities, their criteria and expected outcomes.
- Requirements flow down structures and associated decomposition rationale. (See the concept of rich traceability [22].)

The two aspects just mentioned relate to two key relationships that are typically articulated during requirements management activities: the decomposition (or satisfaction) relationship between layers of requirements, and the association of test activities with requirements. In both cases, a particular type of requirement will

typically require similar patterns of decomposition and testing. This gives rise to the idea of extending the statement level templates into design patterns that bundle a requirement with its decomposition structure and test structure.

### Conceptual representation of requirements

In the templating approach described here, the textual requirement is the primary representation, because there is no underlying formal semantics. In the other approaches described in section 2, the textual representation is merely a projection of a formal construct. The absence of a formal semantics makes the approach applicable to the authoring of requirements that are at a level of abstraction that does not lend itself to formal modelling.

However, it may be possible to abstract away from the textual representation by using conceptual graphs in the style of [11]. More general than any specific formal specification language, conceptual graphs are a candidate for a representation of requirements and system properties from which textual requirements could be projections. From a single conceptual representation of a requirement, a textual statement could be projected in any chosen natural language. Projections from conceptual graphs into modelling notations, such as SysML may also be possible.

### Conclusion

We have attempted to describe hitherto unpublished work on the use of templates – or boilerplates – for textual requirements statements. We have described the general approach, with several specific applications, and a number of tools that offer support. The only available quantitative analysis of the benefits of such an approach is based on small studies [15] [19]. Real practical experience does strongly suggest, however, that the use of templates offers benefits to authors of requirements and managers of projects.

### References

- [1] Guide for Writing Requirements, INCOSE Product INCOSE-TP-2010-006-01, V. 1, April 2012 (requires login)  
<https://connect.incose.org/products/SAWG%20Shared%20Documents/Technical%20Resources/SE%20Practice%20-%20Technical%20Mgmt/Guide%20for%20Writing%20Requirements%202012-0417%20RWG%20TO%20Approved.pdf>
- [2] LEXIOR from Cortim,  
<http://www.cortim.com/pageLibre0001010b.html>
- [3] Requirements Quality Analyser from The Reuse Company,  
[http://www.reusecompany.com/index.php?option=com\\_content&view=category&layout=blog&id=171&Itemid=75&lang=en](http://www.reusecompany.com/index.php?option=com_content&view=category&layout=blog&id=171&Itemid=75&lang=en)
- [4] [www.requirementsengineering.info/boilerplates](http://www.requirementsengineering.info/boilerplates)
- [5] Grunske, L.: Specification patterns for probabilistic quality properties. In: ICSE, New York, ACM (2008) 31-40
- [6] Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: ICSE, New York, ACM (1999) 411-420
- [7] N. Abid, S. Dal Zilio, and D. le Botlan. A Real-Time Specification Patterns Language. Technical Report 11364, LAAS, 2011. <http://hal.archives-ouvertes.fr/hal-00593965>
- [8] Friedemann Bitsch, Safety Patterns - The Key to Formal Specification of Safety Requirements, Proceedings of the 20th International Conference on Computer Safety, Reliability and Security, p.176-189, September 26-28, 2001
- [9] T. Gilb. *A Handbook for Systems and Software Engineering using Planguage*, Addison Wesley 2001
- [10] Suzanne Robertson and James C. Robertson. *Mastering the Requirements Process*, 2<sup>nd</sup> Edition, Addison Wesley 2006.
- [11] John Sowa, *Conceptual structures: Information processing in mind and machine*, Addison Wesley, 1983
- [12] Genova, G. Fuentes, J. Llorens, J. Hurtado, O. Moreno, V. *A framework to measure and improve the quality of textual requirements*, Requirements Engineering, Springer, Url: <http://dx.doi.org/10.1007/s00766-011-0134-z>, Doi: 10.1007/s00766-011-0134-z
- [13] Renault, S. Mendez, O. Franch, X. Quer C. A Pattern-based Method for building Requirements Documents in Call-for-tender Processes. International Journal of Computer Science & Applications (IJCSA), 6(5): 175-202 (2009)
- [14] Definition and Exemplification of RSL and RMM, CESAR Project Technical Report D\_SP2\_R2.1\_M1:  
[http://www.cesarproject.eu/fileadmin/user\\_upload/CESAR\\_D\\_SP2\\_R2.1\\_M1\\_v1.000\\_PU.pdf](http://www.cesarproject.eu/fileadmin/user_upload/CESAR_D_SP2_R2.1_M1_v1.000_PU.pdf)
- [15] Alistair Mavin, Philip Wilkinson, Adrian Harwood, Mark Novak, Easy Approach to Requirements Syntax (EARS), 2009, 17th IEEE International Requirements Engineering Conference, Atlanta, Georgia, USA, August 31-September 04
- [16] Dipankar Majumdar, Sabnam Sengupta, Ananya Kanjilal, Swapan Bhattacharya, "Adv-EARS: A Formal Requirements Syntax for Derivation of Use Case Models", Proc of the First International

Conference on Advances in Computing and Information Technology, July, 15-17, 2011, Chennai, India. pp 40-48.

[17] Requirements Authoring Tool (RAT) from The Reuse Company,

[http://www.reusecompany.com/index.php?option=com\\_content&view=category&layout=blog&id=216&Itemid=187&lang=en](http://www.reusecompany.com/index.php?option=com_content&view=category&layout=blog&id=216&Itemid=187&lang=en)

[18] E. Hull, K. Jackson and J. Dick: Requirements Engineering. First Edition, Springer 2002.

[19] Alistair Mavin, Philip Wilkinson: BIG EARS (The Return of "Easy Approach to Requirements

Syntax", 2010, 18th IEEE International Requirements Engineering Conference, Sydney, Australis, September 27 - October 1

[21] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[22] "Rich Traceability", Jeremy Dick, Proc. 1st International Workshop on Requirements Traceability, Edinburgh, Sept 2002

(all hyperlinks accessed November 2012)

## About the authors

Dr. Jeremy Dick was educated at London University's Imperial College in Computing Science, majoring in formal methods of software development. He went on to pursue academic and industrial research in this field for a number of years. From 1996 to 2006, he worked for software tool supplier Telelogic (now IBM) in their UK professional services organization, as a principal consultant in tool-supported requirements management. This role has afforded him a broad exposure to requirements management practices and issues across many industry sectors. In this role, he developed the concept of Rich Traceability. He co-authored the Springer book entitled "Requirements Engineering," and toured internationally giving seminars in this subject. In 2006, he moved to UK-based consultancy Integrate Systems Engineering Ltd, where he has been helping to develop and promote the concept of Evidence-based Development. He is a past Chairman of INCOSE's Requirements Working Group.

Dr Juan Llorens received his MS degree in Industrial Engineering from the ICAI Polytechnic School at the UPC University in Madrid in 1986, and his PhD in Industrial Engineering and Robotics at the Carlos III University of Madrid in 1996. In 1987 he started his own company dealing with Artificial Intelligence applied to Database systems, named Knowledge Engineering SL (KE). Dr. Llorens worked as technical director in KE and as Marketing Director in Advanced Vision Technologies until 1992, when he joined the Carlos III University. Since 2003 he has been Professor of the Informatics Department of the University. His main subject is Software and Knowledge Reuse, which he teaches in the Informatics and Information Science graduate studies department at the University. Dr. Llorens is the leader of the KR Group (Knowledge Reuse Group) within the University. In 1998 he was invited to the Högskolan på Åland (HÅ) (Åland, Finland). From 1998 to 2008 he split his educational activities between Madrid's University and the HÅ, where he taught Object Oriented Software design. Since 2008, based on an agreement with his University, he shares his academic duties with the position of R+D officer in The Reuse Company (a brand of the University's spin-off organization named CISET) that commercializes systems engineering quality and reuse technology. Since 1999 he has been the board of CISET.

His current research involves the study of Knowledge technologies and System Engineering techniques integration towards Software and Information quality measurement and reuse. He is member of INCOSE's (International Council on Systems Engineering, [www.incose.org](http://www.incose.org)) Requirements Working Group.

His CV is presented at <http://www.kr.inf.uc3m.es/juanllorens.htm>

## About Integrate

**integrate**'s vision is for companies to align strategy and execution, drive quality and ensure compliance through using systems engineering techniques as the bridge between the domains of engineering and management. Our consultancy practice and supporting technologies help clients deliver this vision. We use systems engineering as a vehicle to support informed management and direction, and to optimise solutions against both technical and business constraints. Our approach is intensely pragmatic, firmly focused on our clients' delivery needs, and founded on a thorough understanding of the underlying problem.

For more information go to <http://www.integrate.biz> or email [info@integrate.biz](mailto:info@integrate.biz)

This is an edited version of a paper presented at ICSSEA 2012, the 24th International Conference on Software & Systems Engineering and their Applications

### Find out more

If you require further information, please contact

**Sales Manager**

t: +44 (0)1225 859991

e: [sales@integrate.biz](mailto:sales@integrate.biz)

**integrate systems engineering ltd**

251-253 London Road East

Bathaston

Bath BA1 7RL

United Kingdom

[www.integrate.biz](http://www.integrate.biz)