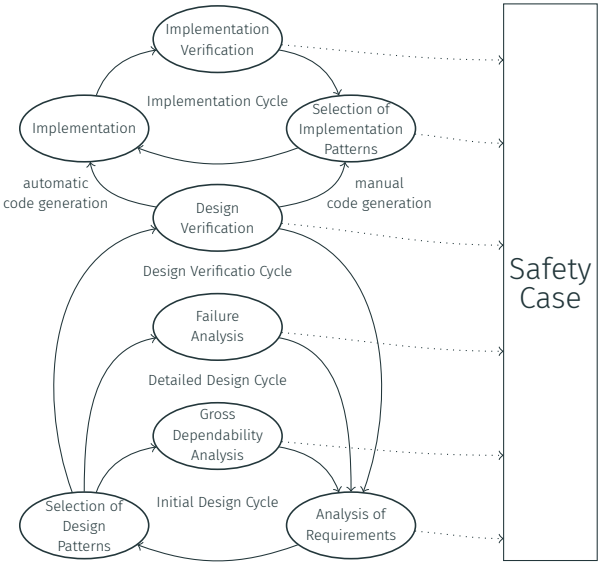
Software Safety

Verification & Validation

Prof. Dr.-Ing. Patrick Mäder, M.Sc. Martin Rabe

Contents

Software Lifecycle



Verification and Validation (V&V)

Software Verification and Validation (V&V)

Validation is the process of evaluating a project deliverable to determine whether it satisfies customer needs.

‘Are we building the **right product**?’

Verification is the process of evaluating a project deliverable to determine whether it satisfies the specifications on which it was based.

‘Are we building the **product right**?’

Landscape of V&V Activities

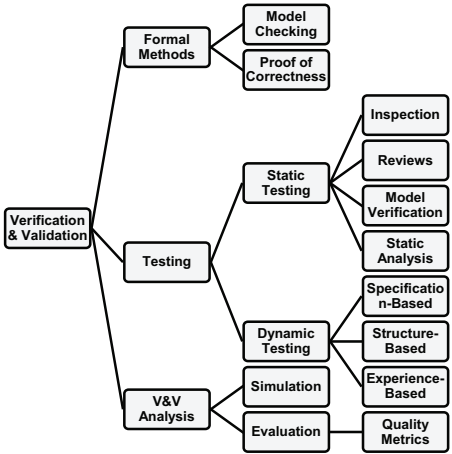


Figure A.1 — Hierarchy of Verification and Validation activities

[ISO/IEC/IEEE 29119-1:2013. Standards catalogue. International Organization for Standardization. September 2013]

Static Testing vs. Dynamic Testing

Static Testing

[Vitaly V. Kuzmin, CC BY-SA 4.0]

- **Goal**: finding defects
- **Examine the work product** for errors
- \+ Executable code not required

Dynamic Testing

[Matheus Ferrero, CCO]

- **Goal**: finding defects
- **Use the work product** to collect failures
- \- Executable code required

Why Is Dynamic Testing Not Sufficient?

- Faults can **mask other faults** at runtime
- Only **completed implementations** can be tested (esp. scalability, performance)
- **Quality attributes**, e.g., security, compliance, maintainability, are hard to test
- **Non-code artifacts** (e.g., design documents) cannot be tested

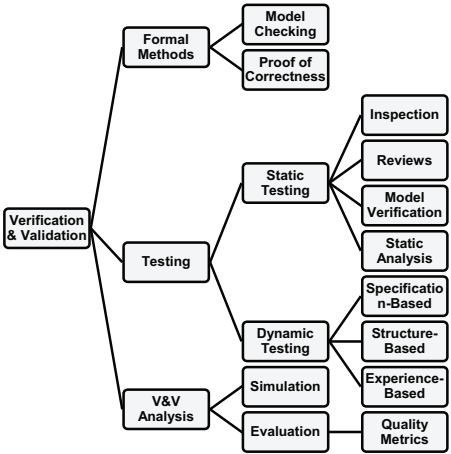
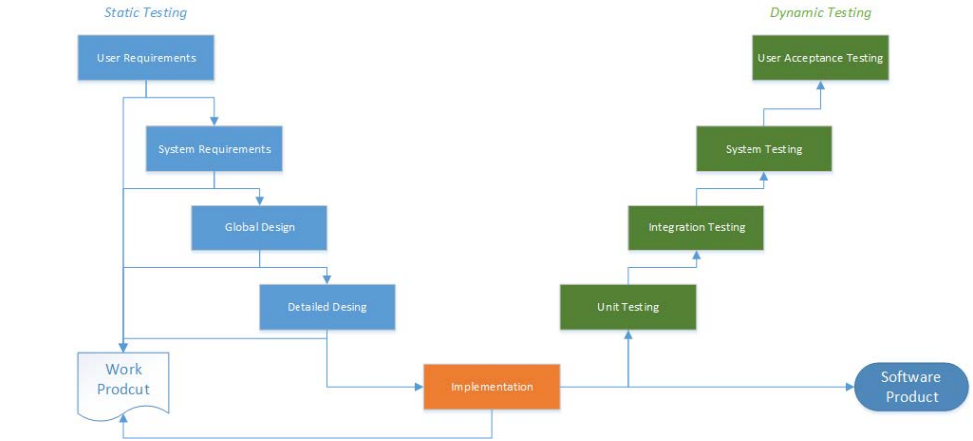


Figure A.1 — Hierarchy of Verification and Validation activities

[ISO/IEC/IEEE 29119-1:2013. Standards catalogue. International Organization for Standardization. September 2013]

Static and Dynamic Testing in a Development Project



→ Any work product can be examined by static testing activities

Reviews

Software Reviews

A **review** is an activity in which one or more persons other than the author of a work product **examine** that product with the intent of **finding defects** and **improvement opportunities**.

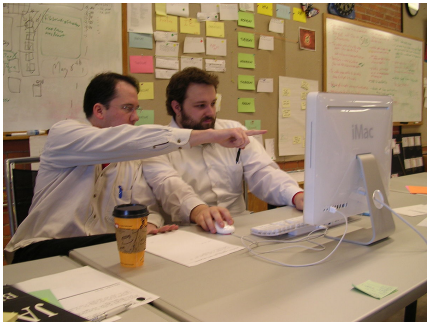
- Distinguish: **informal reviews** and **formal inspections**

[optimy]

Informal Peer Reviews

- Educating other people about the product and collecting unstructured feedback

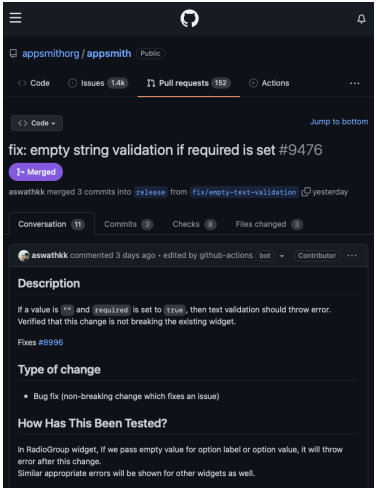
- **Types of informal reviews** [Wiegers2002]:
  - **Peer deskcheck**: asking one colleague to look over your own work product
  - **Passaround (“modern code review”)**: inviting colleagues to examine a deliverable concurrently
  - **Walkthrough**: the author describes a deliverable and solicits comments on it



[Edward, Wikipedia, CC BY 2.0]

Modern Software Reviews

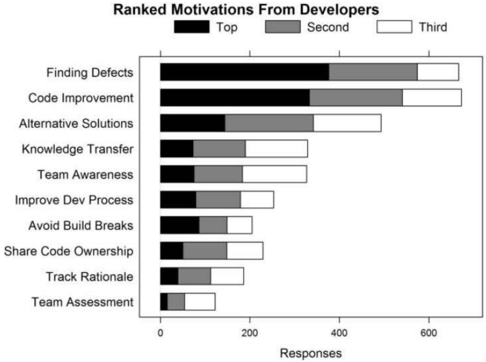
- **Tool support**: integral part of modern IDEs (e.g. MS Visual Studio) and VCS (e.g., GitHub, Bitbucket)
- **Prescribed reviews**: large tech companies like Google, Facebook, Microsoft require reviews of "any change to production code"



[GitHub pull request #9476 of appsmithorg/appsmith]

Motivations in Modern Software Reviews

- Study@Microsoft programmers' motivations for reviewing:



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 [2], [1]. In 1976, Fagan formalized a highly structured process for code reviewing [13], 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 [32].

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 developed to support code review [28]. 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 [19], Facebook [36], and in other organizations and open source software (OSS) projects [40].

This trend raises questions, such as: 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 the challenges faced by practitioners 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 exploratory 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 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. On the other hand, 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.

978-1-4673-3076-3/13/$31.00 © 2013 IEEE 712 ICSE 2013, San Francisco, CA, USA

[A. Bacchelli, C. Bird: **Expectations, outcomes, and challenges of modern code review**. ICSE 2013, 712–721]

→ **Rich set of benefits and motivations beyond defect finding**

- e.g., Google: introduced review to "force developers to write code that other developers could understand" [Sadowski et al. 2018]

Issues of Modern Software Reviews

- **Low quality of code reviews**
  - Reviewers look for easy errors, as formatting issues
  - → Miss serious errors
- **Understanding is the main challenge**
  - Understanding the reason for a change
  - Understanding the code and its context
  - → Feedback channels to ask questions often needed
- **No quality assurance on the outcome**
  - → The review process itself should also be monitored

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 [2], [1]. In 1976, Fagan formalized a highly structured process for code reviewing [13], 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 [32].

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 developed to support code review [28]. 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 [19], Facebook [36], and in other organizations and open source software (OSS) projects [40].

This trend raises questions, such as: 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 the challenges faced by practitioners 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 exploratory 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 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. On the other hand, 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.

978-1-4673-3076-3/13/$31.00 © 2013 IEEE 712 ICSE 2013, San Francisco, CA, USA

Authorized licensed use limited to: TU Ilmenau. Downloaded on November 04,2020 at 10:56:07 UTC from IEEE Xplore. Restrictions apply.

[A. Bacchelli, C. Bird: **Expectations, outcomes, and challenges of modern code review**. ICSE 2013, 712–721]

Inspection

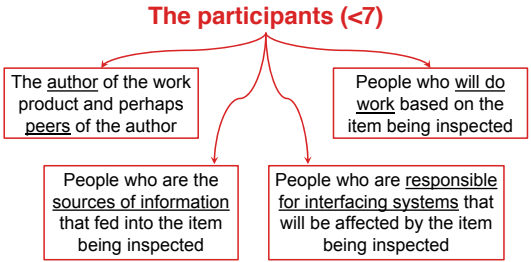
Inspection (Formal Review)

An inspection is a type of **formal peer review** that involves a trained team of individuals who follow a well-defined process to **examine** a **work product carefully for defects**.

- Originally developed by Michael Fagan at IBM (1976)
- Software industry best practice
- Best-established type of formal peer review
- Considered most effective approach to find bugs – typically 60–90% of bugs

Inspection Team and Roles

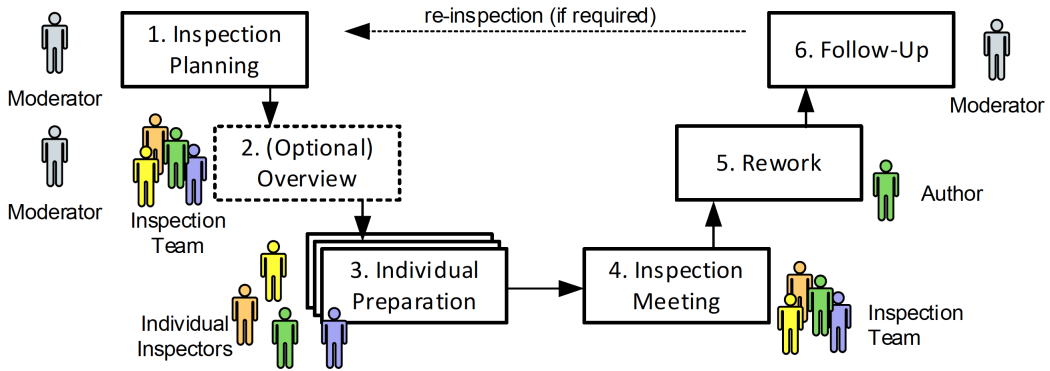
- Inspection roles
  - **Author**: passive role: listens to comments, answers questions
  - **Moderator**: plans inspection with author, coordinates activities and meeting
  - **Reader**: presents the code, model elements, paraphrased requirements
  - **Recorder/scribe**: documents issues and defects



[Dagmar Monett: **Methods for Validating and Testing Software Requirements (lecture slides)**, Europe Week 2015.]

→ They look for defects and improvement opportunities

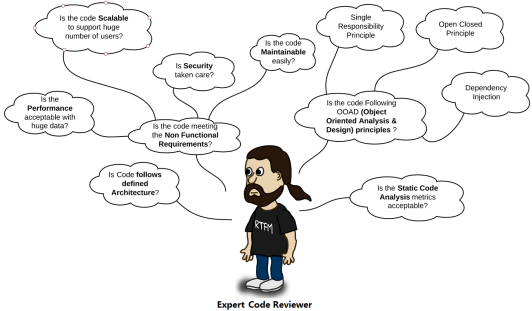
Inspection Process



[D. Winkler, J. Musil, A. Musil, S. Biffl: **Collective Intelligence-Based Quality Assurance: Combining Inspection and Risk Assessment to Support Process Improvement in Multi-Disciplinary Engineering**, EuroSPI '16]

Defect Checklist

- Helps reviewers looking for typical errors
- Include issues detected in the past
- Preferably focus on few important items
- Checklist examples:
  - Are all variables initialized before use?
  - Are all variables used?
  - Is the condition of each if/while statement correct?
  - Does each loop terminate?
  - Do function parameters have the right types and appear in the right order?
  - Are linked lists efficiently traversed?
  - Is dynamically allocated memory released?
  - Can unexpected inputs cause corruption?
  - Have all possible error conditions been handled?
  - Are strings correctly sanitized?



[Code Review Checklist – To Perform Effective Code Reviews, EvokeTechnologies]

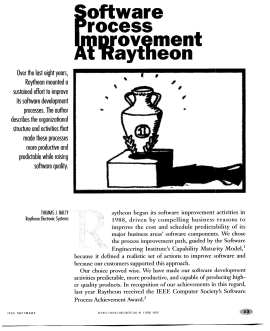
Inspection Process Considerations

- **Author**: does not explain or defend the code – not objective
  - Author $\neq$ moderator, $\neq$ scribe, $\neq$ reader
  - Should join the meeting to observe questions and misunderstandings and to clarify
- **Reader** [optional] walks through the code line by line, explaining it
  - Reading the code aloud requires deeper understanding
  - Verbalizes interpretations, thus observing differences in interpretation
- **Prevent social issues in the inspection process**
  - Identify defects, not alternatives and do not criticize authors
  - Avoid defending code; avoid discussions of solutions/alternatives
  - Avoid style discussions if there are no guidelines

  → Author decides how to resolve fault

Inspection Benefits – Studies and Claims

- [Raytheon]
  - Reduced “rework” from 41% of costs to 20%
  - Reduced integration effort by 80%
- [Paulk et al.]: costs to fix a space shuttle software
  - 1$ if found in inspection
  - 13$ during system test
  - 92$ after delivery
- [IBM]
  - 1h of inspection saves 20h of testing
- [R. Grady]: efficiency data from HP
  - System use: 0.21defects/h
  - Black box testing: 0.28defects/h
  - White box testing: 0.32defects/h
  - Reading/inspection: 1.06defects/h



Software Process Improvement At Raytheon

Over the last eight years, Raytheon mounted a sustained effort to improve its software development processes. The author describes the organizational structure and activities that made these processes more productive and predictable while raising software quality.

THOMAS J. HALEY
Raytheon Electronic Systems

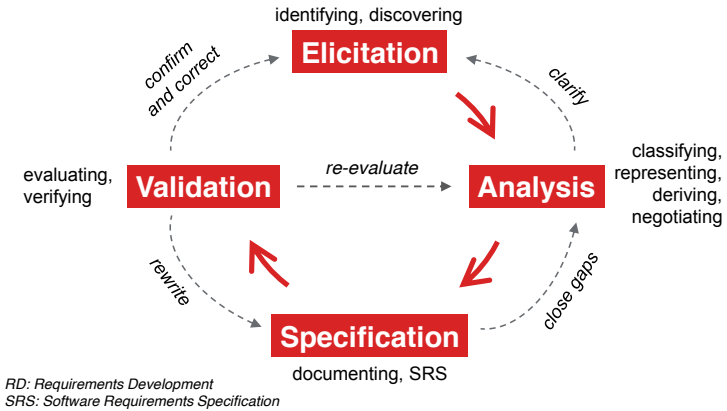
Raytheon began its software improvement activities in 1988, driven by compelling business reasons to improve the cost and schedule predictability of its major business areas' software components. We chose the process improvement path, guided by the Software Engineering Institute's Capability Maturity Model,[1] because it defined a realistic set of actions to improve software and because our customers supported this approach.

Our choice proved wise. We have made our software development activities predictable, more productive, and capable of producing high-er quality products. In recognition of our achievements in this regard, last year Raytheon received the IEEE Computer Society's Software Process Achievement Award.[2]

[T. Haley: Software Process Improvement at Raytheon, IEEE Software, 1996]

Requirements Validation and Management

Requirements Validation



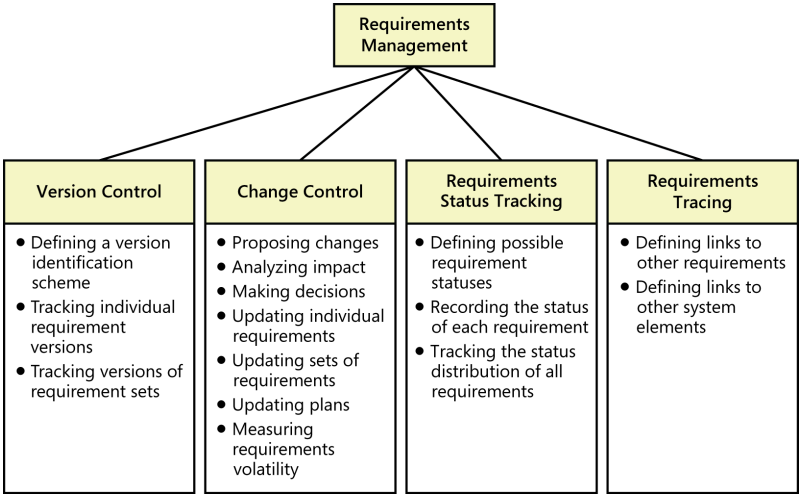
[Dagmar Monett: Methods for Validating and Testing Software Requirements (lecture slides), Europe Week 2015.]

Requirements validation confirms that you have the correct set of requirements that will enable developers to build a solution that satisfies the business objectives.

Validation: Key Actions

Developing acceptance tests and criteria to confirm that a product based on the requirements would meet customer needs and achieve the business objectives.

Requirements Management Activities



[Wiegers&Beatty: Software Requirements, 3rd Edition, Microsoft Press, 2014, p.458.]

Version Control

Version Control: Single Requirements

- Every time a requirement is altered, its version number changes
  - E.g., “0.1, ... 1.0, ... 2.3”
- The version number does not need to be continuous
  - E.g., it is possible to jump directly from 0.1 to 1.0

- Best practices
  - Whole version numbers (e.g., “1.0, 2.0, 3.0”) indicate validated, inspected and accepted statuses of a requirement
  - Increments (e.g., “0.5, 1.1”) indicate unvalidated statuses

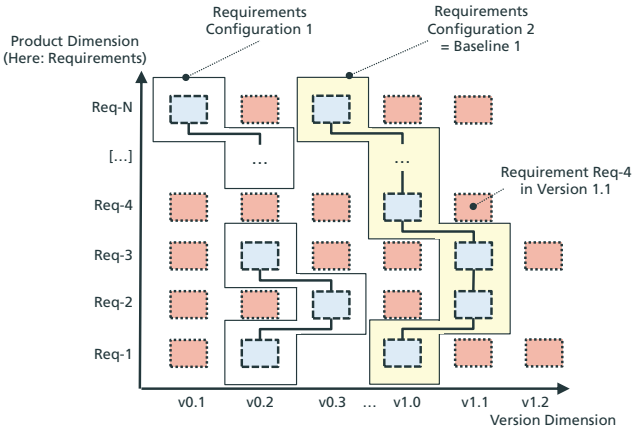
Version Control: Requirements Documents

- **Requirements configuration**: particular version of a requirements document
- Versions of single requirements within a requirements configuration can differ
  - E.g., configuration contains RQ-1 in ver. 0.5 and RQ-2 in ver. 1.3
- New requirements configuration does not have to contain the latest version of the individual requirements

- **Configuration properties**
  - **Logical connections** between the requirements in the configuration
  - **Consistency** among the requirements in the configuration
  - **Unique identification** of the configuration (ID number)
  - **Immutable** (stable) state of the specification and its requirements
  - **Basis for rollbacks** if changes of requirements must be undone

Requirements Baseline

- **Configuration** of, typically stable, requirements
  - represent **basis for development and release planning**
  - can be used to **estimate the effort** needed to realize a system release
  - can be used to **compare** the planned system to competing systems



[Matthias Koch: Requirements Engineering Course 2015/2016, Fraunhofer IESE adapted from Conradi and Westfechel, 1998]

Change Control

Requirements Changes (1/2)

- Over the course of a project, **specifications continuously evolve**
  - New requirements are added
  - Existing requirements are altered
  - Existing requirements are removed
- Many possible **causes**, e.g.:
  - Requirements were wrong, misunderstood or incomplete
  - Desires of the stakeholders have changed
  - New insights about functions, qualities or restrictions
  - Changes in law, technologies, market trends, business processes
- Changes can affect single requirements or entire requirements documents

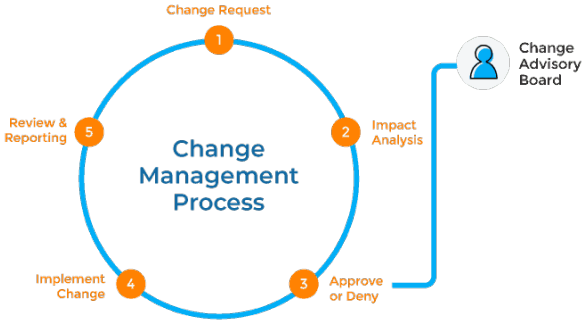
Requirements Changes (2/2)

- Changes are usually good and mostly unavoidable
- The **problem** of requirements changes is not the changes themselves, but **improperly dealing** with them
- Too few changes indicates low stakeholder interest
- Too many changes over a short period of time:
  - indicates inadequately performed requirements engineering activities (e.g., elicitation and negotiation techniques)
  - makes it nearly impossible to develop a system that all stakeholders agree to
  - takes up a lot of resources

→ **Faulty change management can easily ruin a good requirements development**

Change Management

- Describes a **systematic dealing with requirements changes**
- **Core properties**
  - Change request
  - Change Control Board (CCB)
  - Change process
- **Core rules**
  - Nobody changes requirements without approval, including project leader and requirements analyst
  - Change processes only apply to validated configurations/baselines



[PLUTORA: Change Advisory Board (CAB) and Release Management: The Connection, 2020]

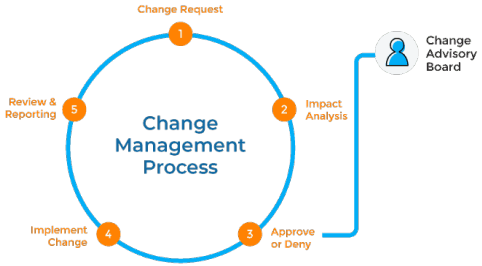
Change Request

- Is a documented request

- Describes the desired change from the view of the requester
  1. What should be changed?
     - Title (summary)
     - Description
  2. Why?
     - Justification
  3. How important is this change?
  4. Who requests the change?
  5. General data (identifier, date filed, which system release)

Activities of the Change Control Board (CCB)

1. Receive change requests
2. Perform an **impact analysis** (consequences, effort, costs)
3. **Review** the change request accordingly
4. **Accept or rejects** the change request
5. Identify the necessary **change measures** (corrective, adaptive, hotfix)
6. **Define** requirement changes or new requirements
7. **Prioritize** and implements/plans the change
8. **Control and validate** the applied changes



[PLUTORA: Change Advisory Board (CAB) and Release Management: The Connection, 2020]

Composition of the Change Control Board

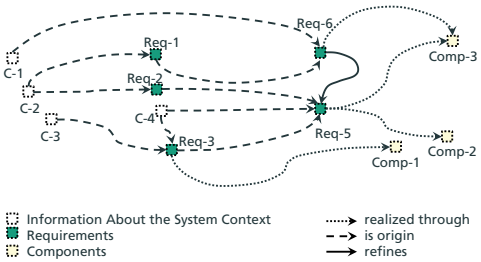
- Change manager
  - Responsible for changes, conflict mediation, negotiation, communicating and documenting decisions
- Contractor
- Architect
- Developer
- Configuration manager
- Product manager
- Project manager
- Quality assurance representative
- Requirements engineer
- Representative of the clients/users

Requirements Tracing

Requirements Traceability

The traceability of a requirement is the ability to trace the requirements over the course of the entire life cycle of the system.

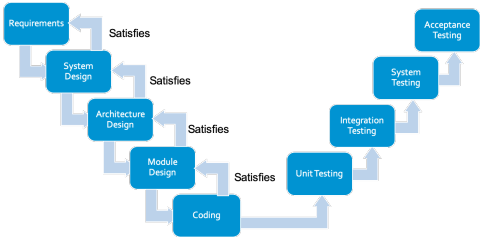
- Establishing the relationships of a requirement with other requirements or other development artifacts
- Purpose-driven approach: **choose the information to be recorded with respect to the purpose** that it will serve
  - Recording only traces that are required
  - Structure traces according the purpose they fulfill



[Matthias Koch: Requirements Engineering Course 2015/2016, Fraunhofer IESE]

Traceability and V&V

- Traceability ensures that steps fit together
  - Starting point for most V&V
- Forward Traceability:
  - Next step in process has everything in current step
  - “Nothing got left out”
- Backward Traceability
  - Previous step in process provoked current step
  - “Nothing spurious included/no gold plating”
- Traceability is an audit
  - Doesn’t prove correctness if tracing is OK
  - But, problems are there if tracing fails



[Ricardo Camacho: Requirements Management and the Traceability Matrix, Parasoft, 2020]

Analyses Based on Traceability

- **Coverage analysis**: verify if a requirement has been implemented
- **Justification analysis**: identify gold-plated solutions
  - Does each element of the implementation contribute to the realization of a requirement?
  - Does each requirement contribute to a system goal?
- **Impact analysis** of the effects of changes
- Opportunities to **reuse** requirements in other projects
- Accountability: retroactive assignment of development efforts
- Simplifies maintenance because the relations have already been drawn

Representations of Requirements Traceability (1/2)

- Individual traceability
  - **Text-based references** (static)
    - Implicit: a business process refers to an activity as “A5”. A following chapter provides an explanation of A5.
    - Explicit; “This business process is refined through activities A3, A4, A5, and A6.“
  - **Hyperlinks** provide the additional possibility to jump to the appropriate section by clicking
- Disadvantage
  - No general overview of all relations

Representations of Requirements Traceability (2/2)

- General traceability
  - **Trace matrices** (tables)
    - Initial artifact IDs in rows, target artifact IDs in columns
    - The table cells visualize the relations, either by marking that a relation exists (“X”), or indicating the type of traceability link
  - **Trace graphs**
    - Requirements are nodes in a graph
    - Edges represent relationships (showing the type of relation)
    - Helps understand transitive relations
- Disadvantage
  - Become difficult to maintain as the number of requirements increase

derived	F-01	F-02	F-03	F-04	F-05	F-06	F-07	F-08	F-09	F-10
F-01		×								
F-02			×							
F-03					×					
F-04	×					×		×		
F-05										
F-06			×	×						
F-07	×	×							×	
F-08						×				
F-09										
F-10								×		

[Matthias Koch: **Requirements Engineering Course 2015/2016**, Fraunhofer IESE]

Requirements Attributes, Views and Tools

Requirements Attributes (1/2)

- Describe information about a requirement in a **structured manner**
  - Information of the same type can always be found in the same position
  - It is harder to overlook important information

- **Frequently used attribute types**
  - Identifier (ID)
  - Name
  - Description
  - Version
  - Author
  - Source
  - Stability
  - Criticality (e.g., SIL, ASIL)
  - Priority

Requirements Attributes (2/2)

- Additional attribute types
  - Requirement type
  - Person responsible
  - Status
    - Regarding the content
    - Regarding the validation
    - Regarding the agreement
  - Effort
  - Release
  - Legal obligation
  - Cross references
  - General information

Example: Attributes of a Requirements

Attribute	
Identifier	F-04
Name	Change travel data
Description	The system should provide the Travel Management with the ability to change travel data.
Version	1.0
Author	R. the Requirements Engineer
Source	T. Travelmanager
Stability	fixed
Priority	must-have
Person Responsible	D. Developer
Status Content	concept
Status Validation	in validation
X-Ref	F-01; F-06; F-08

[Matthias Koch: **Requirements Engineering Course 2015/2016**, Fraunhofer IESE]

Attribute Scheme

- The **set of all defined attributes** for a class of requirements
  - E.g., a use case has attributes such as “Name” and “Pre-condition”
- Each class of requirement can have a tailored set of attribute types, depending on:
  - specific properties of the project
  - constraints of the organization
  - properties and regulations of the application domain
  - constraints and restrictions of the development process
- For each requirement, an (default) attribute value is provided
  - E.g., an attribute value for “priority” could be “high”

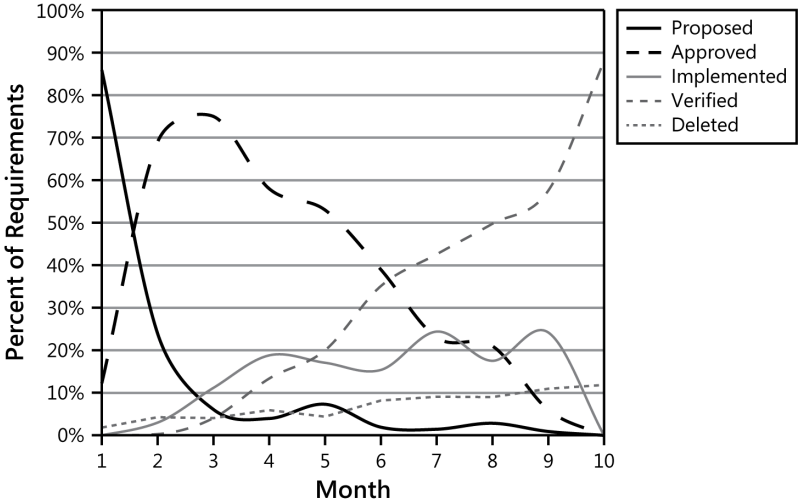
Attribute Structures

- Table structure (template)
  - Simplest way
  - Manual
- Information models (model-based)
  - Usually when employing tools for requirements management
  - Allows for determining relations between attribute types of different attribute schemes (relational database)
  - Requirement dependencies help maintaining consistency
  - Templates can be generated

Requirements Views: Selective and Condensed

- Views keep the complexity of the requirements manageable
- Views require the use of information models
- Selective views
  - Select particular requirements and/or mask certain attributes
  - Create a view depending on role or sub-activities of that role, e.g.:
    - views for architects, programmers, project managers, testers
    - managing only those requirements that one is responsible for
    - selecting or viewing requirements by an attribute value
- Condensed views
  - Aggregate or generate data to obtain statistics, e.g.:
    - calculating the percentage of high-priority requirements
    - generating graphs by summing up attribute values

Requirements View Example: Status Tracking



[Wiegers&Beatty: Software Requirements, 3rd Edition, Microsoft Press, 2014, p.466.]

Requirements Status Scheme Example

Status	Definition
Proposed	The requirement has been requested by an authorized source.
In Progress	A business analyst is actively working on crafting the requirement.
Drafted	The initial version of the requirement has been written.
Approved	The requirement has been analyzed, its impact on the project has been estimated, and it has been allocated to the baseline for a specific release. The key stakeholders have agreed to incorporate the requirement, and the software development group has committed to implement it.
Implemented	The code that implements the requirement has been designed, written, and unit tested. The requirement has been traced to the pertinent design and code elements. The software that implemented the requirement is now ready for testing, review, or other verification.
Verified	The requirement has satisfied its acceptance criteria, meaning that the correct functioning of the implemented requirement has been confirmed. The requirement has been traced to pertinent tests. It is now considered complete.
Deferred	An approved requirement is now planned for implementation in a later release.
Deleted	An approved requirement has been removed from the baseline. Include an explanation of why and by whom the decision was made to delete it.
Rejected	The requirement was proposed but was never approved and is not planned for implementation in any upcoming release. Include an explanation of why and by whom the decision was made to reject it.

[Wiegers&Beatty: Software Requirements, 3rd Edition, Microsoft Press, 2014.]

Requirements Management Tools

- **Requirements management tools** typically focus on the support and implementation of requirements management and quality assurance tasks, e.g.:
  - attribute-based documentation
  - category and view creation
  - traceability
  - versioning
  - change management and impact analysis
  - (prioritization)

Types of RE/M Tools (1/3)

- Specialized requirements management tools
  - Typically UI + databases with multi-user / multi-tenant capabilities
  - E.g., Rational DOORS (IBM), Rational Requisite Pro (IBM), CaliberRM (Borland), Reqtify (Dassault Systemes)
- System development tools
  - Designed to support integration with other development activities
  - Test management or configuration management
    - E.g., HP Quality Center, bug tracking (JIRA, PTC, Polarion)

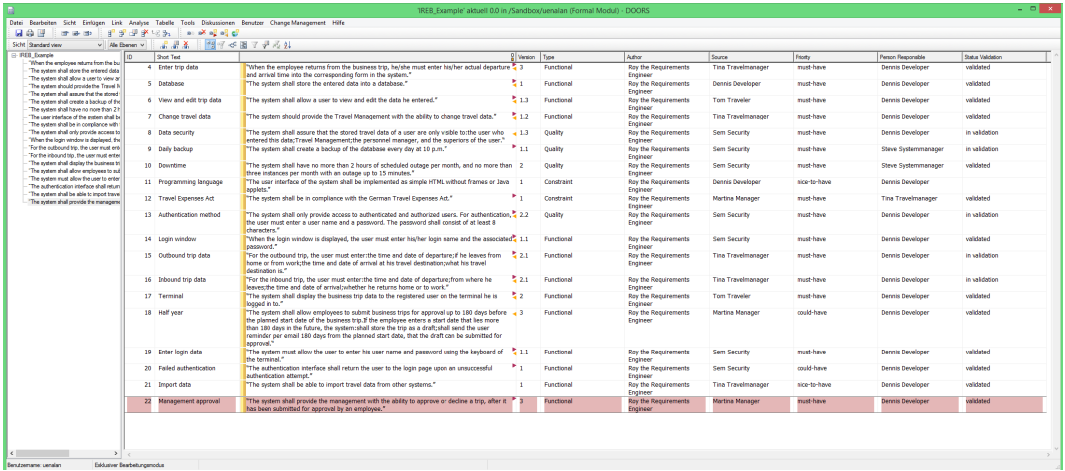
Types of RE/M Tools (2/3)

- System development tools
  - Simulation and visualization tools
    - E.g., mind maps, GUI prototyping
    - Useful for clarification and early validation
  - Modeling tools
    - E.g., Enterprise Architect
    - Support different views of requirements, e.g., use cases, behavior models, data models, test cases
    - Support model-based syntactical checking of requirements, but require traceability between requirements for all representations

Types of RE/M Tools (3/3)

- Wikis
  - Support simultaneous elicitation
- Office software
  - E.g., Word, Excel, Visio, Outlook
  - Widespread and easy to use
  - Requires templates and workarounds for use with RE practices (e.g., traceability, versioning)

Example: IBM Rational DOORS



[Matthias Koch: **Requirements Engineering Course 2015/2016**, Fraunhofer IESE]

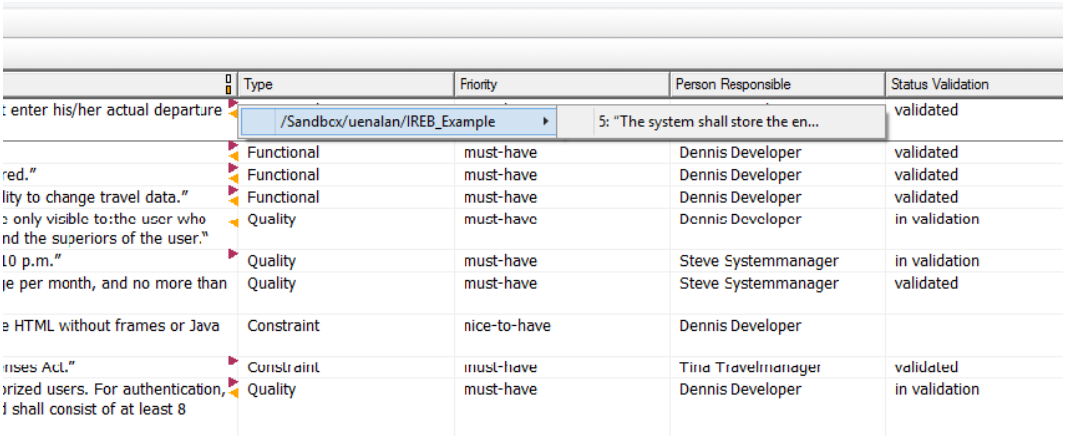
Common Features of RM Tools (1/2)

- Manage various information types
  - E.g., natural language, models, sketches, project plans, change requests
- Manage logical relationships between information
  - Horizontal and vertical traceability of requirements
- Identify artifacts uniquely
  - Unique ID for traceability
- Make information accessible flexibly and securely
  - Access control, multi-user, configuration and version management

Common Features of RM Tools (2/2)

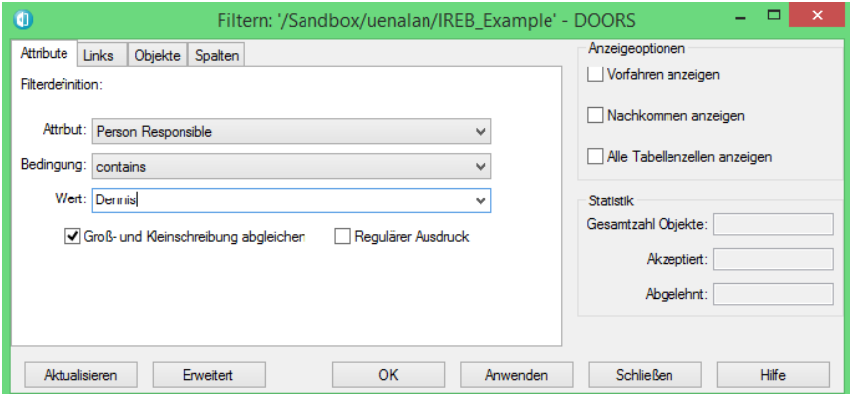
- Support different views on requirements
  - E.g., user / role specific, system specific, requirements types
- Organize information
  - E.g., attribute assignment or hierarchies, grouping, annotations
- Generate reports or summaries over the information
  - E.g., change requests, modification history, traceability matrix
- Generate documents from the information
  - Requirements specification for system release XYZ

Example: Requirements Traces in DOORS



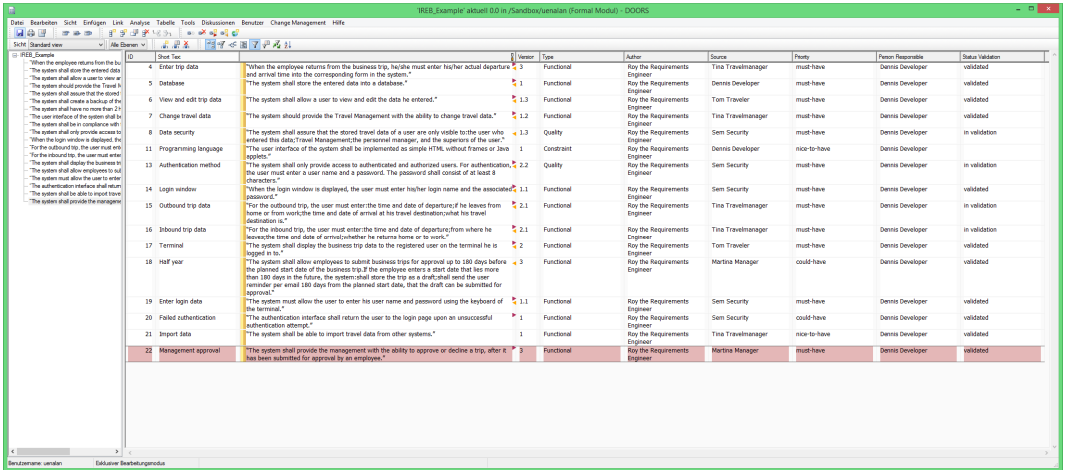
[Matthias Koch: Requirements Engineering Course 2015/2016, Fraunhofer IESE]

Example: Role-Specific View in DOORS (1/2)



[Matthias Koch: Requirements Engineering Course 2015/2016, Fraunhofer IESE]

Example: Role-Specific View in DOORS (2/2)



[Matthias Koch: **Requirements Engineering Course 2015/2016**, Fraunhofer IESE]

Specialized RM Features

- Manage requirements and attributes based on information models
- Organize requirements by hierarchy and type levels
- Configuring and version management on requirement level
- Define requirement baselines
- User management (including access rights and control)
- Traceability management
- Consolidate requirements (e.g., view generation)
- Change management (change control and impact analysis)
- Import / export data

Summary

Summary

- Goals of requirements management
  - Maintaining a persistent availability of the documented requirements
  - Structuring the information in a sensible manner
  - Allow for selective access to the information
- Techniques for requirements management
  - Assigning attributes to requirements
  - Prioritizing requirements
  - Traceability of requirements
  - Versioning of requirements
  - Management of requirements changes
- Tools can support requirements management

Failed Feedback …

Dilbert
© Scott Adams
At http://dilbert.com/strip/2013-02-25/
(Educational/Classroom usage permission is granted by Universal Uclick. All Rights Reserved)

Questions?