

Title:
Title: A6.1-D4 Concept of component validation Final version
Author(s)/Organisation(s):
Dénes Magyari / FÖMI
Working Group:
WP6 – Component Validation Process
References:
<ul style="list-style-type: none"> A6.1-D3 Concept of component validation 3.0 Development Rules (663-development_rules-etra) HUMBOLDT Programming Guidelines (589-humboldt_programming_guidelines-etra)

Short Description:
Definition of software quality assurance terms and concept of component validation activities in HUMBOLDT for version final version components.
Keywords:
Quality Assurance, QA, Continuous Integration, CI, Build Server, Issue Tracking, Testing, Automated Testing, Unit Test, Integration Test, Code Coverage, Static Code Analysis, Metrics

History:			
Version	Author(s)	Status	Comment
001	Gyula IVÁN	new/rfc/final	
002	Gyula IVÁN, Gábor SZABÓ, László BÉRCZES		
003	Dénes Magyari	final	A6.1-D2 Concept of component validation 2.0
004	Dénes Magyari	rfc	Document restructuring. Updating CI server and issue tracking server descriptions to current state.
005	Dénes Magyari	final	Status changed to final.
006	Dénes Magyari	rfc	Previous version extended with chapter 5.3 describing the Sonar code analysis server

Table of contents

1	Introduction.....	5
1.1	Purpose and scope of A6.1-D3.....	5
1.2	Definitions.....	7
1.2.1	Acceptance testing.....	7
1.2.2	Component testing.....	7
1.2.3	Final qualification testing.....	7
1.2.4	Software integration testing.....	7
1.2.5	Software verification.....	8
1.2.6	Software validation.....	8
1.2.7	System testing.....	8
1.2.8	Test case.....	8
1.2.9	Test design.....	8
1.2.10	Test plan.....	8
1.2.11	Software verification and validation plan.....	8
1.2.12	Software quality assurance plan.....	9
1.3	Test processes.....	10
1.4	Software component validation methods.....	11
1.5	ISO 9126.....	13
2	Requirements.....	15
2.1	Terms and definitions.....	15
2.1.1	Traceability analysis-on requirements.....	15
2.1.2	Software requirement evaluation.....	15
2.1.3	Interface analysis – in requirements phase.....	17
2.1.4	System test plan analysis.....	17
2.1.5	Acceptance test plan analysis.....	17
2.1.6	Timing and sizing analysis.....	17
2.2	Characteristics of a good requirement.....	18
3	Design quality metrics and principles.....	18
3.1	Terms and definitions.....	18
3.1.1	Traceability Analysis – in design phase.....	18
3.1.2	Software design evaluation.....	19
3.1.3	Interface analysis – in design phase.....	20
3.1.4	Software final qualification test plan analysis.....	20
3.1.5	Software integration test plan analysis.....	21
3.1.6	Database analysis.....	21
3.1.7	Component test plan analysis.....	22
3.1.8	Data flow analysis.....	22
3.2	Principles of class design.....	22
3.2.1	Class cohesion.....	22
3.2.2	Single-responsibility principle.....	23
3.2.3	Open/closed principle.....	23

3.2.4	Liskov substitution principle	24
3.2.5	Dependency-inversion principle	24
3.2.6	Interface segregation principle	26
3.2.7	Law of Demeter	26
3.2.8	Reused abstractions	26
3.3	Principles of package and component design	27
3.3.1	Principles of component cohesion: granularity	27
3.3.1.1	Reuse/Release Equivalence Principle	27
3.3.1.2	Common Reuse Principle	27
3.3.1.3	Common Closure Principle	27
3.3.1.4	Package Cohesion Metrics	27
3.3.2	Principles of component coupling: stability	27
3.3.2.1	Acyclic Dependencies Principle	27
3.3.2.2	Stable-Dependencies Principle	27
3.3.2.3	Stable-abstractions principle	29
4	Code quality	31
4.1	Test driven development.....	32
4.2	Guides and rules from QA perspective	33
4.2.1	Project Folder Layout	33
4.2.2	Compiler Settings.....	33
4.2.3	Build Environment	33
4.2.4	Build Scripts	33
4.3	Static code analysis tools.....	33
4.3.1	Checkstyle.....	33
4.3.2	Findbugs	34
4.3.3	PDM	34
4.3.4	JLint.....	34
4.4	Terms and definitions	35
4.4.1	Traceability Analysis – on code	35
4.4.2	Source code and documentation	35
4.4.3	Interface analysis - code	36
4.4.4	System test case analysis.....	37
4.4.5	Software final qualification test case analysis.....	37
4.4.6	Software integration test case analysis.....	37
4.4.7	Acceptance test case analysis.....	37
4.4.8	Software integration test procedure analysis.....	37
4.4.9	Software integration test results analysis.....	37
4.4.10	Component test case analysis	38
4.4.11	System test procedure analysis	38
4.4.12	Software final qualification test procedure analysis	38
5	Integration quality	38
5.1	Version control system (SVN)	38
5.1.1	Trunk	39
5.1.1.1	States of the trunk	39
5.1.2	Tags and Branches	39
5.1.2.1	Naming Convention for Tags and Branches	39

5.1.2.2	Release Tags	40
5.1.2.3	Release Branches.....	40
5.1.2.4	Private branches	41
5.2	Continuous Integration	41
5.2.1	Concept of Continuous Integration	41
5.2.2	CI System Setup	42
5.3	Code Analysis Server.....	42
5.3.1	Sonar Overview.....	43
5.3.1.1	Code Coverage	44
5.3.1.2	Rules Compliance	45
5.3.1.3	General Metrics.....	45
5.3.2	Quality Profiles	47
5.3.3	Server Implementation	50
6	Issue Tracking.....	50
6.1	Requirements	51
7	Release Management.....	51

1 Introduction

1.1 Purpose and scope of A6.1-D3

The aim of component validation process is to ensure that the developed software components by WP8 are of sufficient quality to match the requirements formulated in WP5. Component validation process is a quality control procedure, which has to define the different characteristics of the implementation of the framework.

For the development of such a process different materials and knowledge bases must be taken into account:

- Evaluation of different standards on software quality, verification and testing,
- Experiences of software manufacturers of our consortium in software quality control and management,
- Experiences of public sector in software verification and validation,
- Experiences on reaction of software users.

Component Validation Process has many connection points with other workpackages within HUMBOLDT project. Validation process has to cover the implementation of framework development (WP8). It is not explicitly written in project documentation, but data harmonization (WP7) has a great influence on validation process too. As above mentioned, the main input of component validation process will be formulated in WP5 (Framework Interface).

Both WP6 and WP10 work on different aspects of software quality. The distribution of work between WP 6 and WP 10 is:

- WP6: Technical validation: all technical characteristics of software quality.
- WP10: Usability testing: anything involving interaction with users.

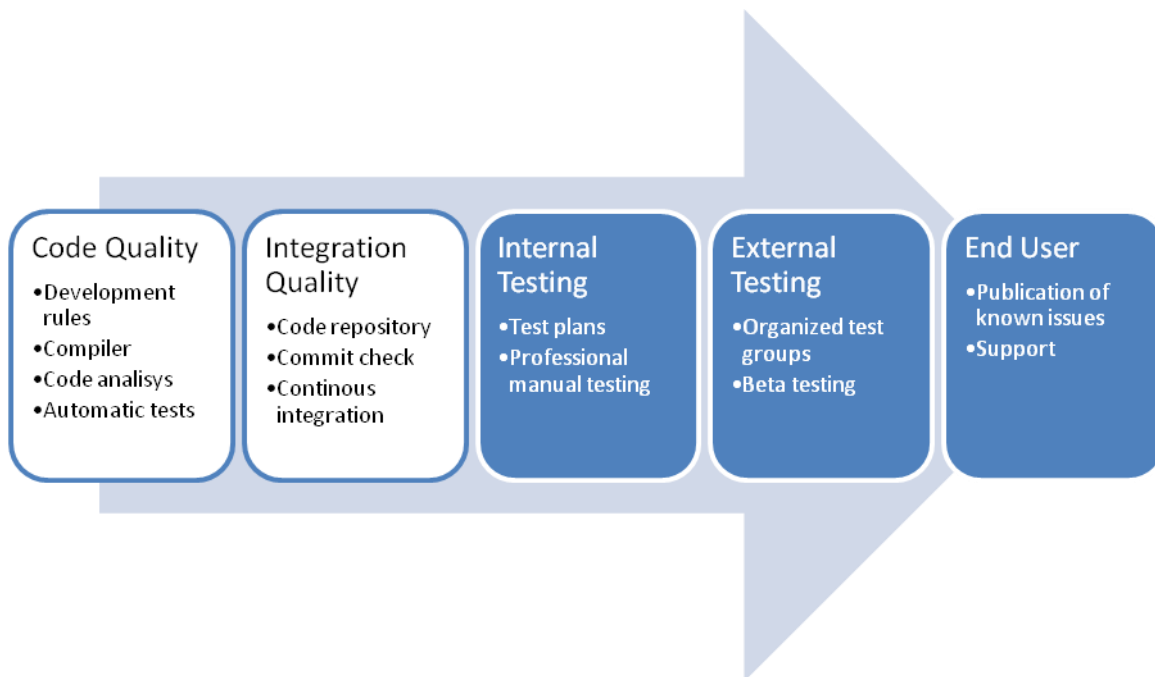


Figure 1: Quality aspects of software production

The outline of the current document follows the process of software production from individuals writing code, through integrating code and testing within and outside of the project.

This document follows the path of software production, addressing quality assurance aspects of each stage.

The first two stages are main concern for developers and the source code itself. Employed techniques are dealing with aspects of the source code i.e. stylistic, syntax, semantic and architectural. This part of the quality assurance procedure is white box approach, because it focuses on the implementation details.

The second half strictly deals with functionality thus it is a black box approach. Procedures in this part are not interested in implementation details only the functionality is tested and inspected here. In these stages the procedures should be carried out persons who are not involved in the development process itself, rather domain experts and competent users of the product.

The goal of each stage is to prevent a bug or error to advance to the next stage. Each bug crossing a stage involves significantly more effort to correct and handle. The discovery of an issue at any stage bounces back to the first section i.e. the code must be corrected. The cost of uncovering an issue grows significantly each time a bug crosses a barrier.

Infrastructure for quality assurance:

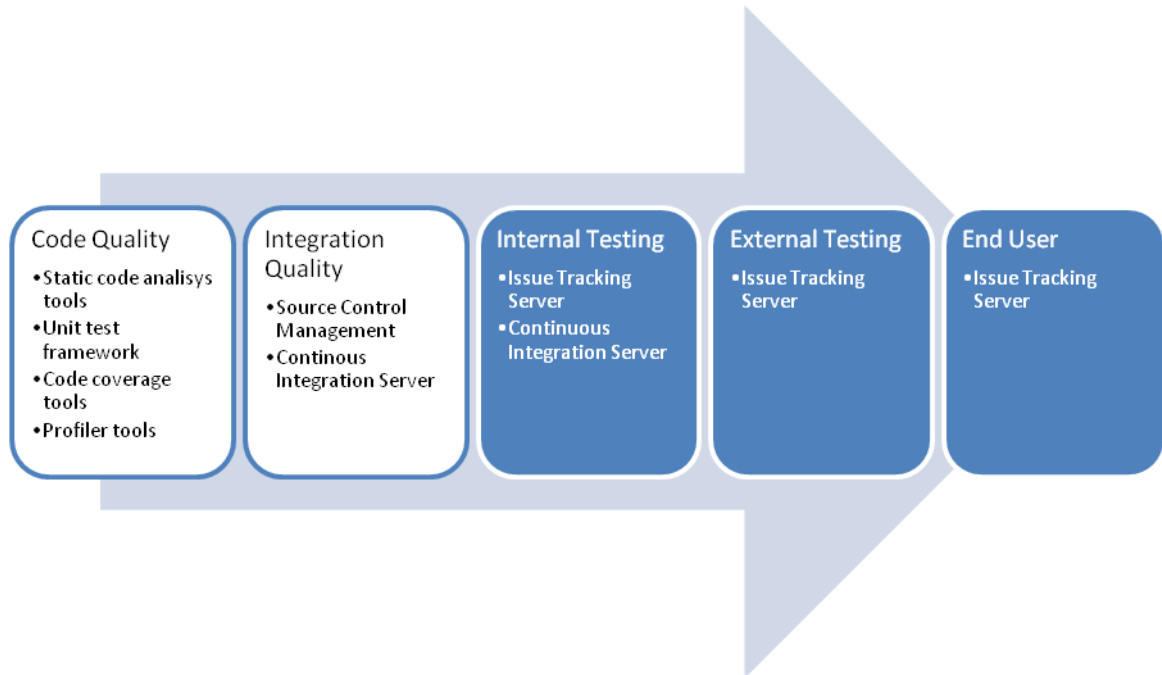


Figure 2: Tools supporting each phase of quality assurance

1.2 Definitions

1.2.1 Acceptance testing

Acceptance testing is testing conducted in an operational environment to determine whether a system satisfies its acceptance criteria (i.e. initial requirements and current needs of its user). Acceptance testing enables the customer to determine whether to accept the system.

1.2.2 Component testing

Component testing verifies the correct implementation of the design and compliance with program requirements for one software element (e.g. unit, module).

1.2.3 Final qualification testing

Final qualification testing the complete software program verifies that the software meets all of the software requirements and is ready to be integrated with system hardware.

1.2.4 Software integration testing

Software integration testing is an orderly progression of testing of incremental pieces of the software program in which software elements are combined and tested to show the compliance with the software design, capabilities and requirements. Software integration testing is also performed to verify operation under off-nominal and stress conditions.

1.2.5 Software verification

Software verification is the process, in which certified, that the software in a development phase satisfies all the requirements, which has been specified in the previous phase.

Verification step-by-step theoretically is enough to certify equivalency between the starting and finishing phase. But, because the whole process generally is not exact, a separate final verification is necessary, which have to be executed between the specification and the end product. If there was a mistake or defect in specification, the end-product will not satisfy the user requirements, therefore a separate control process is needed, in which the product is controlled from original function point of view. This process is validation

1.2.6 Software validation

Software validation is the process of evaluating software at the end of its software development process to ensure compliance with software requirements. This process ensures that the software system performs to the customer's expectations under operational conditions.

1.2.7 System testing

Systems testing is an orderly progression of testing of incremental pieces of the system in which both hardware and software elements are combined and tested until the entire system has been integrated. System testing verifies the requirements of the system and validates whether the system meets its original objectives.

1.2.8 Test case

Test case is a documentation that specifies inputs, predicted results, and sets of execution conditions for a test item

1.2.9 Test design

Test design is a documentation that specifies the details of the test approach for a software feature or combination of software features. The test design identifies the associated tests.

1.2.10 Test plan

Test plan is a documentation that specifies the scope, approach, resources and schedule if intended test activities.

1.2.11 Software verification and validation plan

Software verification and validation plan (V&V plan) fixes the regulations, which related to the control activities carried out during the software development. V&V plan provides a guideline for the checking the accomplishment of requirements including in software specification, and tests must be executed within the development phases.

The V&V plan describes the principles and methods of checks be used during the development, in order to provide, that the software satisfies the quality requirements.

The V&V plan unambiguously determines the liability, responsibility of testers and their contacts with the developers. This part must be refresh continuously during the project period.

The V&V plan must contain the followings:

- The location of V&V within the development, its organization method and the list of persons, who are accredited to V&V,
- Necessary resources,
- The persons, who are responsible for the execution of tasks,
- The instruments and methods are used during the testing,
- The system of documentation of results of testing, the contents of reports,
- Order of problem indication and solution, criterions of execution of retesting,
- Conditions and order of deviations from V&V plan,
- Systematisation and storing of documents, utilities, tools, mailing derived from testing,
- Standards, inner regulations, practices complied during testing activities.

Besides these content, the V&V plan appoints for each development phase separately:

- Connected checking (testing) tasks,
- Methods must be followed,
- Acceptance and decision criterions must be used,
- Input and Output document list of the phase,
- Time available for the execution of test,
- List of necessary instruments, equipments, professionals etc.,
- Risks derived from failure of the phase and handling of them,
- Role and responsibility of members of testing group.

1.2.12 Software quality assurance plan

Software quality assurance plan synthesizes the activities and rules related to quality assurance during the whole development cycle of the software. Software quality assurance plan must contain at least the followings:

- Aims of quality assurance plan,
- Catalogue of referenced documents,
- Method of project management and organization,
- Documents to be delivered during the development, and their contents,
- Standards, rules and conventions to be taken into account,

- Testing activities and order of testing,
- Scheme of problem indication and solution, personal responsibility,
- Used instruments, methods and methodologies,
- Archiving of source code, version management, handling of program (management of software configuration),
- Procedures against unauthorized access and physical damages,
- Methods of storing and systematisation of documentation.

1.3 Test processes

Test process is worth executing in sections, where the testing can be carried out incrementally accordantly with the implementation of the system.

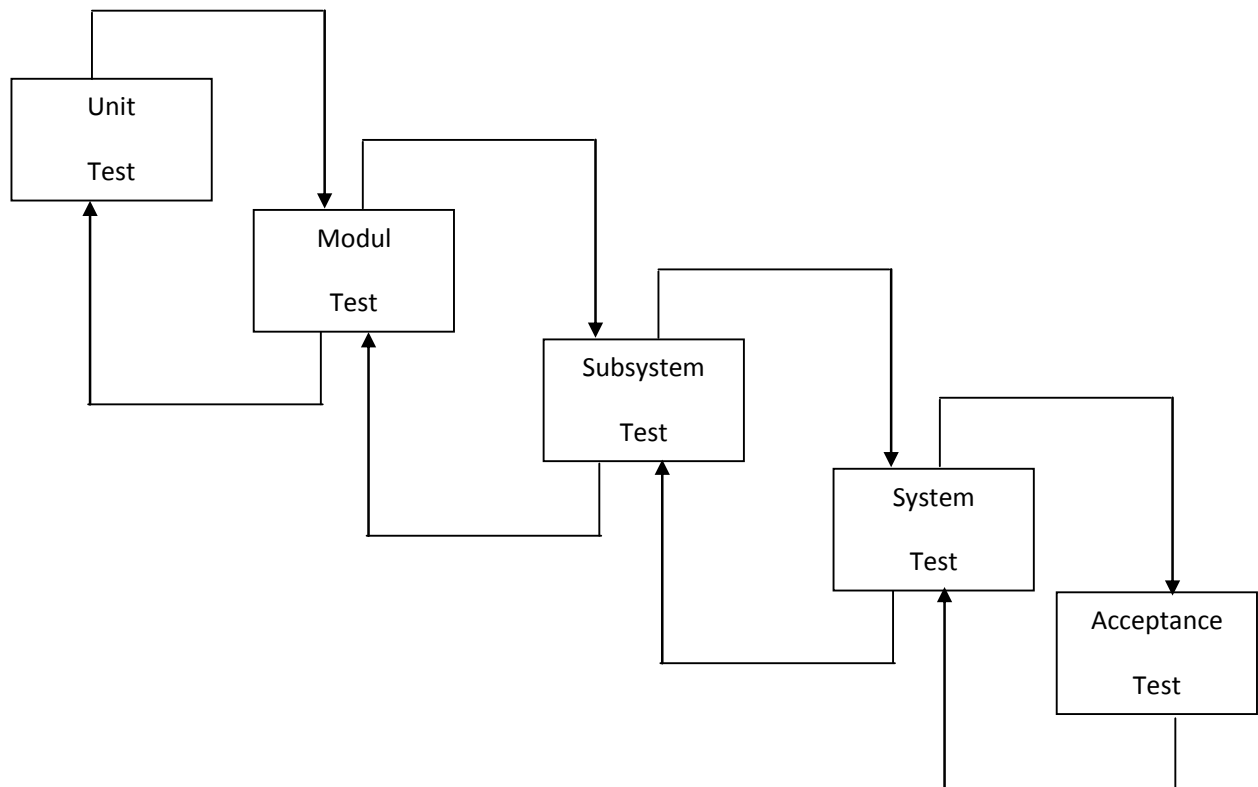


Figure 3: Test process

Sections of test process:

Unit Test: The individual components have to be tested independently of the others, and be provided the faultless operating.

Modul Test: The modul is a group of interdependent components, the moduls also can be tested independently of the others

Subsystem test: The tests of moduls, which compose a subsystem. This test process focuses on the interface-errors of moduls, because the most problems are derived from the wrong connections of interfaces.

System test: This test phase deals with the errors, derived from the unanticipated interactions between the subsystems and their interfaces, and is concerned with validation too.

Acceptance test: The system is tested with the data of end-user, not with test data. Errors can be detected in real situation. Problems can be arisen, which show the system properties do not fulfil the user requirements.

Unit and modul test is generally the task of the programmer, who develops the component. The later sections of testing are the responsibility of independent groups of testers, based on test plans. Figure 4. shows the different testing phases in software developments.

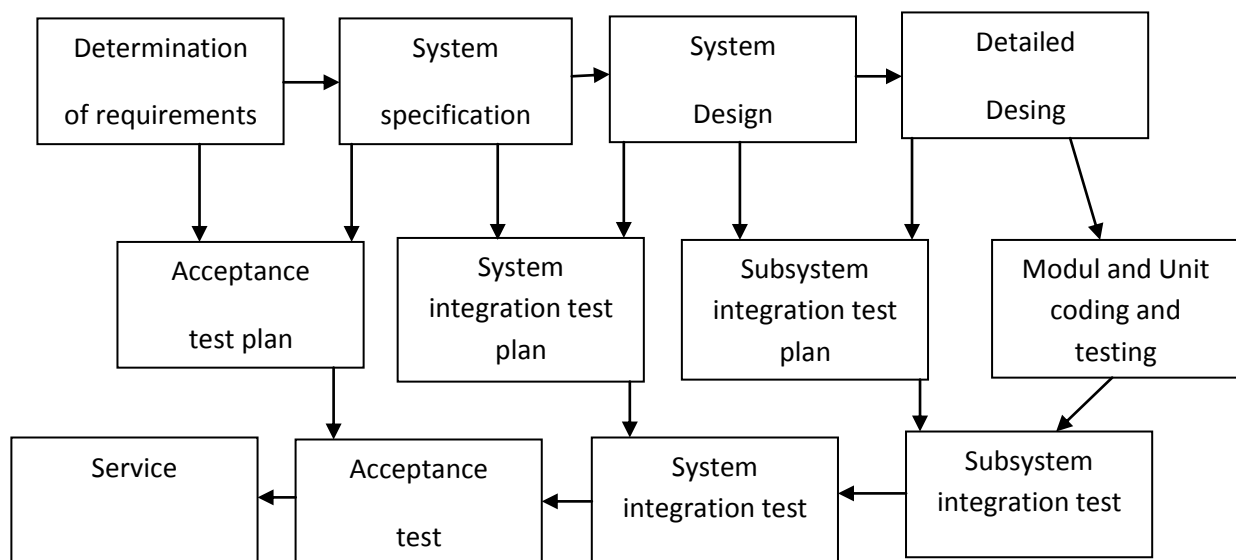


Figure 4: Testing phases in software development

1.4 Software component validation methods

There are many different software validation methods for components. They could be classified into two classes:

- Black-box validation methods (also known as functional testing methods)
 - These methods refer to the systematic techniques for testers to design and generate test cases and data to achieve a certain test adequacy criteria for a component based on its component specifications.
- White-box validation methods (also known as program-based testing methods, or structure-based testing methods)
 - These methods refer to the systematic techniques for testers to design and generate test cases and data to achieve a certain test adequacy criteria for a component based on its component program and structure,

Black-box method's base idea is a testing of a component as a black box. It focuses only on the external accessible behaviors and functions. It checks a component's outputs for selected component inputs.

Black-box testing methods for components can be classified into three groups:

- Usage-based black-box testing techniques – focusing on user oriented accesses
 - User-operation scenario testing,
 - Random testing,
 - Statistical testing,
- Error-based black-box testing techniques – focusing on error-prone points
 - Equivalence partitioning testing,
 - Category-partitioning testing,
 - Boundary-value analysis,
 - Decision table-based testing,
- Fault-based black-box testing techniques – focusing on targeted fault points
 - Mutation testing,
 - Fault injection method.

White-box testing methods are focusing on the internal structures of a component, on the internal component logics and behaviors of a component. These testing methods can be used in two purposes:

- Component developers use white-box testing methods to discover program-oriented errors in a component development process,
- Component users use white-box testing methods to check program errors in a component based software development process in the following cases:
 - Creating and checking newly created components,
 - Validating the extended and customized parts of adopted components.

From white-box testing methods for components can be classified into three types of testing:

- Path testing (based on program flow graph),
- Data flow testing,
- Object-oriented testing.

Path testing:

- Exercise program control flows of a component to check its outputs based on the given inputs
 - Use program flow graph as a test model,
 - Generate test cases based on the test model,

- Apply test coverage criteria based on the test model.

Data flow testing:

- Focuses on program errors relating to incorrect data definition and usage
 - Program flow graphs are used as test model,
 - Use program flow graph to identify various data flow paths, such as data define and use path,
 - Generate test cases to exercise various data flow paths to achieve the selected data flow test coverage criteria.

Object-oriented testing:

- Testing object-oriented program by focusing on object-oriented program structures and features,
- Generate test cases based on selected object-oriented test models to achieve pre-defined test coverage criteria,
- Basic approaches:
 - Object state-based testing,
 - Class relation-based testing, inheritance testing,
 - Class function-based dependency testing,
- Test models:
 - Object state chart,
 - Class relation graph,
 - Class function dependency graph.

1.5 ISO 9126

The quality model established in the first part of the standard, ISO 9126-1, classifies software quality in a structured set of characteristics and sub-characteristics as follows:

- Functionality - A set of attributes that bear on the existence of a set of functions and their specified properties. The functions are those that satisfy stated or implied needs.
 - Suitability
 - Accuracy
 - Interoperability
 - Compliance
 - Security
- Reliability - A set of attributes that bear on the capability of software to maintain its level of performance under stated conditions for a stated period of time.

- Maturity
- Recoverability
- Fault Tolerance
- Usability - A set of attributes that bear on the effort needed for use, and on the individual assessment of such use, by a stated or implied set of users.
 - Learnability
 - Understandability
 - Operability
- Efficiency - A set of attributes that bear on the relationship between the level of performance of the software and the amount of resources used, under stated conditions.
 - Time Behaviour
 - Resource Behaviour
- Maintainability - A set of attributes that bear on the effort needed to make specified modifications.
 - Stability
 - Analyzability
 - Changeability
 - Testability
- Portability - A set of attributes that bear on the ability of software to be transferred from one environment to another.
 - Installability
 - Replaceability
 - Adaptability
 - Conformance (similar to compliance, above, but here related specifically to portability, e.g. conformance to a particular database standard)

Some of these attributes come from the design, specification and the set of formulated requirements of the system:

- Usability
- Maintainability
- Portability

Some attributes are come from the architecture of the system:

- Functionality
- Reliability

- Maintainability
- Portability

While others from the implementation quality itself:

- Functionality
- Reliability
- Efficiency

The role of WP6 activity in terms of ISO 9126 focuses mostly on the last group.

2 Requirements

The main input for the component validation process is the formulation of requirements. Validating the functionality of components is testing their capability to fulfil their requirements applying various techniques.

2.1 Terms and definitions

2.1.1 Traceability analysis-on requirements

Tracing the software requirements (software specifications, interface specifications) to system requirements (concept) and system requirements to the software requirements is necessary. In this analysis the following actions should be carried out:

- **Correctness:** Validation of correctness of the relationships between each software requirement and its system requirement,
- **Consistency:** Verification of that, the relationship between the software and system requirements are specified to a consistent level of detail,
- **Completeness:** In completeness analysis should:
 - Verify that every software requirement is traceable to a system requirement with sufficient detail to show compliance with the system requirement,
 - Verify that all system requirements related to software are traceable to software requirements,
- **Accuracy:** Validation of that, the system performance and operating characteristics are accurately specified by the traced software requirements.

2.1.2 Software requirement evaluation

In software requirement evaluation the requirements (e.g. functional, capability, interface, qualification, safety, security, human factors, data definitions, user documentation, installation and acceptance, user operation and user maintenance) of system requirements and interface requirements specification should be investigated for correctness, consistency, completeness, accuracy, readability, and testability. Task criteria are the follows:

- **Correctness:**
 - Verify and validate that the software requirements satisfy the system requirements allocated to software within the assumptions and constraints of the system,
 - Verify that the software requirements comply with applicable standards, references, regulations, policies and business rules,
 - Validate that the flow of data and control satisfy functionality and performance requirements,
 - Validate data usage and format,
- **Consistency:**
 - Verify that all terms and concepts are documented consistently,
 - Verify that the function interactions and assumptions are consistent and satisfy system requirements and acquisition needs,
 - Verify that there is internal consistency between the software requirements and external consistency with the system requirements,
- **Completeness:**
 - Verify that the following elements are in software requirements and interface requirements specifications within the assumptions and constraints of the system:
 - Functionality (e.g. algorithms, state/mode definitions, input/output validation, exception handling, reporting and logging),
 - Process definition and scheduling,
 - Hardware, software and user interface descriptions,
 - Performance criteria (e.g. timing, sizing, speed, capacity, accuracy, precision, safety, and security),
 - Critical configuration data,
 - System, device, and software control (e.g. initialisation, transaction and state monitoring, self.testing),
 - Verify that the system requirements and interface requirements specifications satisfy specified configuration management procedures,
- **Accuracy:**
 - Validate that the logic, computational, and interface precision (e.g. truncation and rounding) satisfy the requirements in the system environment,
- **Readability:**
 - Verify that the documentation is legible, understandable, and unambiguous to the intended audience,
 - Verify that the documentation defines all acronyms, mnemonics, abbreviations, terms and symbols,

- Testability:
 - Verify that there are objective acceptance criteria for validation the requirements of the system requirements and interface requirements specifications.

2.1.3 Interface analysis – in requirements phase

In interface analysis (in requirements phase) verification and validation of that, the requirements for software interfaces with hardware, user, operator and other systems are correct, consistent, complete, accurate and testable is needed. Criteria are the follows:

- Correctness: Validate the external and internal system and software interface requirements,
- Consistency: Verify that the interface descriptions are consistent between the system requirements and interface requirements specifications,
- Completeness: Verify that each interface is described and includes data format and performance criteria (e.g. timing, bandwidth, accuracy, safety and security),
- Accuracy: Verify that each interface provides information with the required accuracy,
- Testability: Verify that there are objective acceptance criteria for validating the interface requirements.

2.1.4 System test plan analysis

In system test plan analysis the verification of that, the system test plan conforms to HUMBOLDT defined test document purpose, format and content is needed. The criteria are the follows:

- Test coverage of system requirements,
- Appropriateness of test methods and standards used,
- Feasibility of system qualification testing,
- Feasibility and testability of operation and maintenance requirements.

2.1.5 Acceptance test plan analysis

In acceptance test plan analysis the verification of that, the acceptance test plan complies with HUMBOLDT defined test document purpose, format and content is necessary. Criteria are the follows:

- Test coverage of system requirements,
- Feasibility of operation and maintenance (e.g. capability to be operated and maintained in accordance with user needs).

2.1.6 Timing and sizing analysis

In this step collection and analysis of data about the software functions and resource utilizations are needed to determine if system and software requirements for speed and capacity are satisfied. The types of functions and resource utilization issues include, but are not limited to the following:

- CPU Load,

- RAM and secondary storage,
- Network speed and capacity,
- Input and output speed.

Sizing and timing analysis is started at software design and iterated through acceptance testing.

2.2 Characteristics of a good requirement

A requirement needs to meet several criteria to be considered a “good requirement” Good requirements should have the following characteristics:

- Unambiguous
- Testable (verifiable)
- Clear (concise, terse, simple, precise)
- Correct
- Understandable
- Feasible (realistic, possible)
- Independent
- Atomic
- Necessary
- Implementation-free (abstract)

Besides these criteria for individual requirements, three criteria apply to the set of requirements.

The set should be:

- Consistent
- Nonredundant
- Complete

3 Design quality metrics and principles

3.1 Terms and definitions

3.1.1 Traceability Analysis – in design phase

In this step the analysis of trace relationships between design elements (System and Interface Design) and requirements (system and interface requirements) for correctness, consistency and completeness is needed. The criteria are the follows:

- **Correctness:** Validate the relationships between each design element and software requirement,
- **Consistency:** Verify that the relationships between the design elements and the software requirements are specified to a constant level of detail,
- **Completeness:**
 - Verify that all design elements are traceable from the software requirements,
 - Verify that all software requirements are traceable to the design elements.

3.1.2 Software design evaluation

In this section the evaluation of design elements for correctness, consistency, completeness, accuracy, readability and testability is required. Criteria are the follows:

- **Correctness:** Verify and validate that the software design satisfies the software requirements. Verify that the software design complies with applicable standards, references, regulations, policies and business rules. Validate the software design sequences of states and state changes using logic and data flows coupled with domain expertise, prototyping results, engineering principles and other basis. Validate that the flow of data and control satisfy functionality and performance requirements. Validate data usage and format. Assess to appropriateness of design methods and standards,
- **Consistency:** Verify that all terms and design concepts are documented consistently. Verify that there is internal consistency between the design elements and external consistency with architectural design,
- **Completeness:** Verify that the following elements are in the System Design, within the assumptions and constraints of the system:
 - Functionality (e.g. algorithms, state/mode definitions, input/output validation, exception handling, reporting and logging),
 - Process definition and scheduling,
 - Hardware, software and user interface descriptions,
 - Performance criteria (e.g. timing, sizing, speed, capacity, accuracy, precision, safety, and security),
 - Critical configuration data,
 - System, device, and software control (e.g. initialisation, transaction and state monitoring, self-testing),

Verify that the System Design and Interface Design satisfy specified configuration management procedures.

- **Accuracy:** Validate that the logic, computational, and interface precision (e.g. truncation and rounding) satisfy the requirements in the system environment.
- **Readability:**

- Verify that the documentation is legible, understandable, and unambiguous to the intended audience,
- Verify that the documentation defines all acronyms, mnemonics, abbreviations, terms and symbols, and design language if any.
- Testability:
 - Verify that there are objective acceptance criteria for validating each software design element and the system design. Verify that each software design element is testable to objective acceptance criteria.

3.1.3 Interface analysis – in design phase

In interface analysis (in design phase) verification and validation of that, the software design interfaces with hardware, user, operator and other systems are correct, consistent, complete, accurate and testable is needed. Criteria are the follows:

- Correctness: Validate the external and internal software interface design in the context of system requirements,
- Consistency: Verify that the interface design is consistent between the system design and interface design,
- Completeness: Verify that each interface is described and includes data format and performance criteria (e.g. timing, bandwidth, accuracy, safety and security),
- Accuracy: Verify that each interface provides information with the required accuracy,
- Testability: Verify that there are objective acceptance criteria for validating the interface design.

3.1.4 Software final qualification test plan analysis

In this step the verification of the final qualification test plan is carried out, that it complies with HUMBOLDT defined test document purpose, format and content. Criteria are the follows:

- Traceable to the software requirements,
- External consistency with the software requirements,
- Internal consistency,
- Test coverage of the software requirements,
- Appropriateness of test standards and methods used,
- Feasibility of software qualification testing,
- Feasibility on operation and maintenance (e.g. capability to be operated and maintained in accordance with user needs).

3.1.5 Software integration test plan analysis

In this phase the verification of integration test plan is necessary, that it complies with HUMBOLDT defined test purpose, format and content.

Criteria are the follows:

- Compliance with increasingly larger set of functional requirements at each stage of integration,
- Assessment of timing, sizing and accuracy,
- Performance at boundaries and under stress conditions,
- Traceable to the software requirements,
- External consistency with the software requirements,
- Internal consistency,
- Test coverage of the software requirements,
- Measures of requirements test coverage and software reliability,
- Appropriateness of test standards and methods used,
- Feasibility of software qualification testing,
- Feasibility on operation and maintenance (e.g. capability to be operated and maintained in accordance with user needs).

3.1.6 Database analysis

Evaluation of database design includes the following:

- Physical limitations analysis: Identify the physical limitations of the database such as maximum number of records, maximum record length, largest numeric value, smallest numeric value, maximum array length in a data structure and compare them to designed values,
- Index vs. Storage Analysis: Analyse the use of multiple indexes compared to the volume of stored data to determine if the proposed approach meets the requirements for data retrieval performance and site constraints,
- Data Structure Analysis: Some DBMS have specific data structures within a record, such as arrays, tables and date formats. Review the use of these structures for potential impact on requirements for data storage and retrieval,
- Backup and Disaster Recovery Analysis: Review the methods employed for backup against the requirements for data recovery and system disaster recovery and identify deficiencies.

3.1.7 Component test plan analysis

In this phase the verification of component test plan is necessary, that it complies with HUMBOLDT defined test purpose, format and content.

Criteria are the follows:

- Compliance with design requirements,
- Assessment of timing, sizing and accuracy,
- Performance at boundaries and interfaces and under stress and error conditions,
- Traceable to the software requirements and design,
- External consistency with the software requirements and design,
- Internal consistency between unit requirements,
- Test coverage of requirements in each unit,
- Measures of requirements test coverage and software reliability and maintainability,
- Feasibility of software integration and testing,
- Feasibility on operation and maintenance (e.g. capability to be operated and maintained in accordance with user needs).

3.1.8 Data flow analysis

Evaluation of data flow is including the followings:

- Symbology consistency check. Verify that each symbol is used consistently,
- Flow Balancing. Compare the output data from each process block to the data inputs and the data derived within the process to ensure the data is available when required. This process does not specifically examine timing or sequence considerations,
- Confirmation of Derived Data. Examine the data derived within a process for correctness and format. Data designed to be entered into a process by operator action should be confirmed to ensure availability,

Keys to Index Comparison. Compare the data keys used to retrieve data from data stores within a process to the database index design to confirm that no invalid keys have been used and the uniqueness properties are consistent.

3.2 Principles of class design

3.2.1 Class cohesion

Class design should reduce the need to edit multiple classes when making changes to application logic. A fundamental goal of OO design is to place the behaviour (methods) as close to the data they operate on (attributes), so that changes are less likely to propagate across multiple classes.

Metrics

$$\text{Lack of Cohesion of Methods (LCOM)} = \frac{(\sum R(A))/A - M}{1 - M}$$

Where

- A is the number of attributes
- M is the number of methods
- each attribute A is accessed by R(A) methods

3.2.2 Single-responsibility principle

The definition of responsibility in this context is reason to change. A class should have only one reason to change.

Separation of concerns is very important for flexible design. For example if a class or interface has the main responsibility of some domain logic, it is tempting to mix infrastructural features such as logging or data handling into the same class.

Nontrivial example is a simple modem interface with the following methods:

- Dial
- HangUp
- Send
- Receive

Separation of data communication might be desirable to be able to use the implementer class in more generic communication situations.

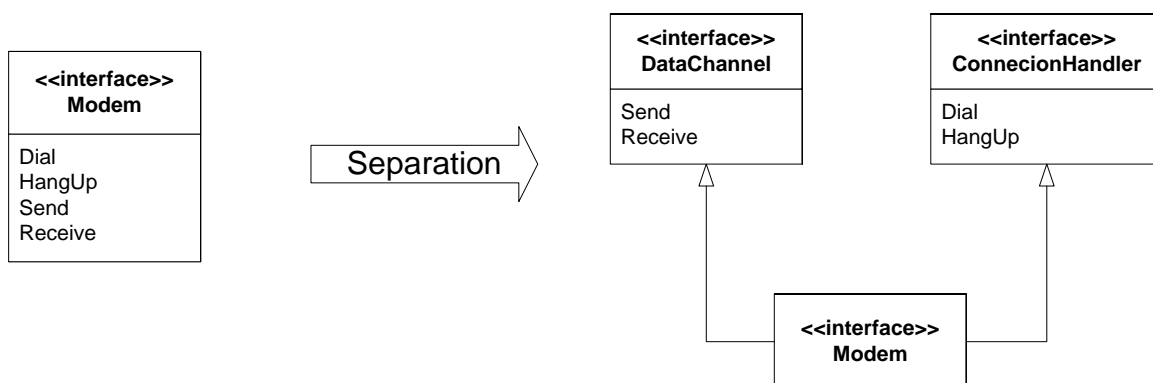


Figure 5: Example of separation of concerns

Metrics

Possible metric is responsibility / class, but identifying a responsibility might be difficult until the evaluation of the design requires the change itself.

3.2.3 Open/closed principle

Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification.

- Open for extension means that the behaviour of the module can be extended. As the requirements of the application change, we can extend the module with new behaviors that satisfy those changes. In other words, we are able to change what the module does.
- Closed for modification means that extending the behavior of a module does not result in changes to the source, or binary, code of the module. The binary executable version of the module whether in a linkable library, a jar, or a .exe file remains untouched.

Typical technique to satisfy this principle is application of the template method design pattern.

3.2.4 Liskov substitution principle

Subtypes must be substitutable for their base types.

Dynamic polymorphism is a powerful mechanism that allows us to invert dependencies, reducing duplication and making change much easier. All OO design principles depend upon polymorphism, but we must ensure that any type can be substituted for any of its subtypes at run-time without having any adverse effect on the client. Subtypes must obey all of the rules that apply to their super-types pre-conditions for calling methods, post-conditions of methods called, and invariants that always apply between method calls.

Typical violations come from the generally accepted term “is-a” for inheritance. In real life “is-a” relations are often more constraining for the subtype not a more generalized version of it. Typical examples are square is a rectangle and circle is an ellipse. Both examples are candidate for violation this principle. The correct term for OO design is “behaves like”. A square does not behave like a rectangle because it is more constrained.

Metrics

Every class should pass all of the unit tests for all of its super-types.

3.2.5 Dependency-inversion principle

- High-level modules should not depend on low-level modules. Both should depend on abstractions.
- Abstractions should not depend upon details. Details should depend upon abstractions.

Much of the duplication in code comes from client objects knowing about all sorts of specialized suppliers, that from the clients perspective do similar things but in different ways. Polymorphism is a powerful mechanism that underpins OO design. It allows us to bind to an abstraction, and then we don't need to know what concrete classes we are collaborating with. This makes it much easier to plug in new components with no need to change the client code.

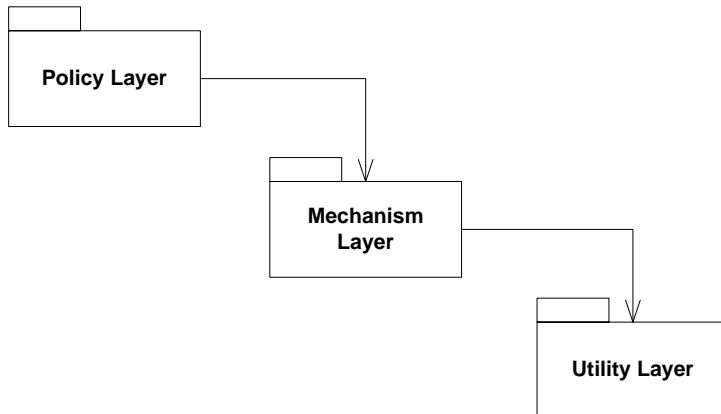


Figure 6: Trivial implementation of layered architecture

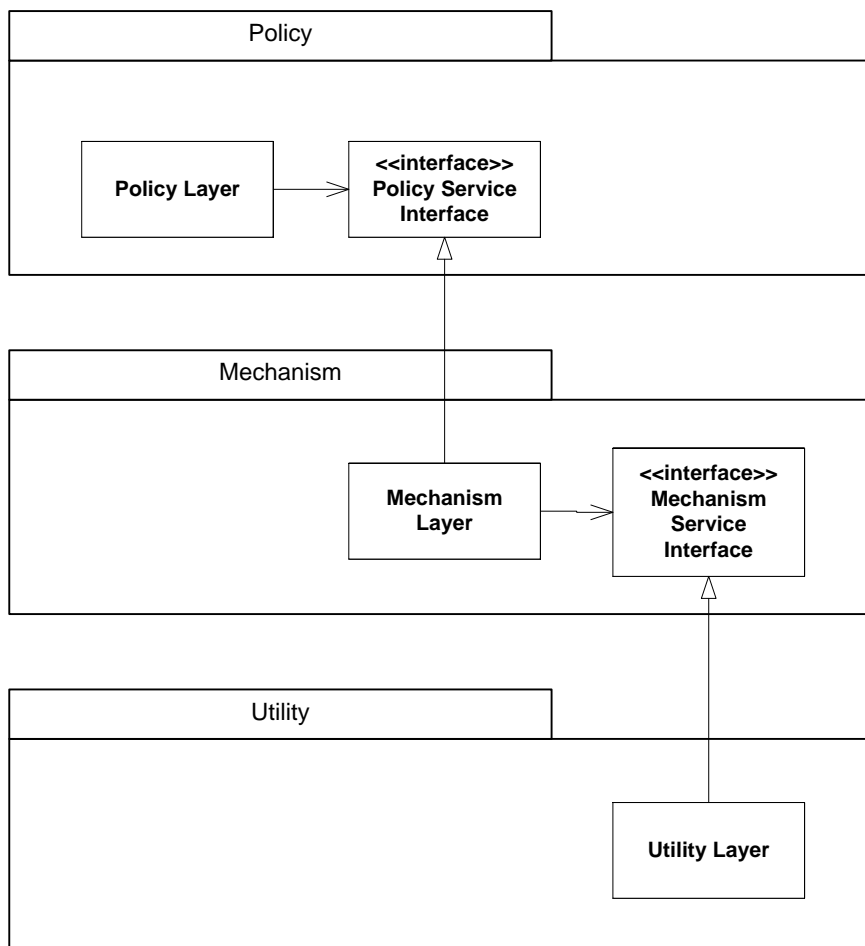


Figure 7: Inverted dependencies between layers

A typical problem is when infrastructural libraries such as Hibernate, that offer transparent services, are mixed into some higher level library. Although the service that the utility library offers is transparent in operation, the dependency towards the utility library makes it very difficult to adapt other implementations of the utility functions in the high level library.

3.2.6 Interface segregation principle

Clients should not be forced to depend on methods they do not use.

Metrics

If type T exposes N methods, and client C uses n of them, then T's interface is n/N specific with respect to C.

Average n/N for all clients of T

3.2.7 Law of Demeter

Objects should only collaborate with their nearest neighbours the less they depend on the interfaces of friends of a friend, the less reasons they might have to change. This means avoiding long navigations and deferring knowledge of interactions with objects that aren't directly related to your nearest neighbours.

Metrics

Average depth of navigation.

3.2.8 Reused abstractions

Abstractions should be extended or implemented by more than one class.

In test-driven development, abstractions are discovered by looking for similarities between classes or interfaces. Designers should distinguish between bona fide abstractions and indirection. A bona fide abstraction incorporates shared elements of two or more types into a single, shared abstraction to which both types conform. When we create arbitrary abstractions (e.g., interfaces for mock object tests), we create an extra maintenance burden with no pay off in term so removal of duplication.

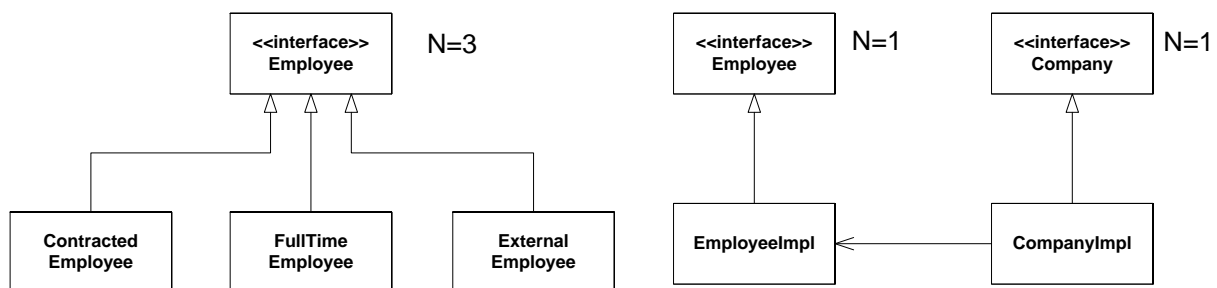


Figure 8: Example of reused abstractions

Metrics

For an abstract class or interface T, which is extended or implemented by N classes or interfaces, such that $N > 1$

3.3 Principles of package and component design

3.3.1 Principles of component cohesion: granularity

3.3.1.1 Reuse/Release Equivalence Principle

The unit of reuse is the unit of release.

3.3.1.2 Common Reuse Principle

The classes in a component are reused together. If you reuse one of the classes in a component, you reuse them all.

3.3.1.3 Common Closure Principle

The classes in a component should be closed together against the same kinds of changes. A change that affects a component affects all the classes in that component and no other components.

3.3.1.4 Package Cohesion Metrics

- Class C depends directly or indirectly on N classes in the same package P
- There are M classes in P
- Common reuse & common closure with respect to C is $N/(M - 1)$
- Package cohesion for P is the average of $N/(M - 1)$ across all classes in P

Except when $M \leq 1$, in which case package cohesion is zero (as opposed to $0 / (1 - 1)$ which would be undefined!)

3.3.2 Principles of component coupling: stability

3.3.2.1 Acyclic Dependencies Principle

Packages must not be indirectly dependent on themselves.

3.3.2.2 Stable-Dependencies Principle

Depend in the direction of stability.

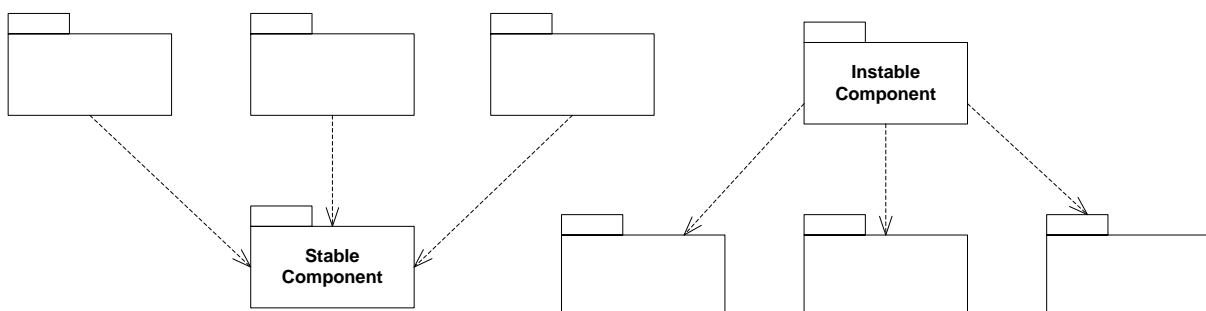


Figure 9: Stable and instable component

Definition of stability:

- C_a (afferent couplings): The number of classes outside this component that depend on classes within this component
- C_e (efferent couplings): The number of classes inside this component that depend on classes outside this component
- $I(Instability): I = \frac{C_e}{C_a + C_e}$

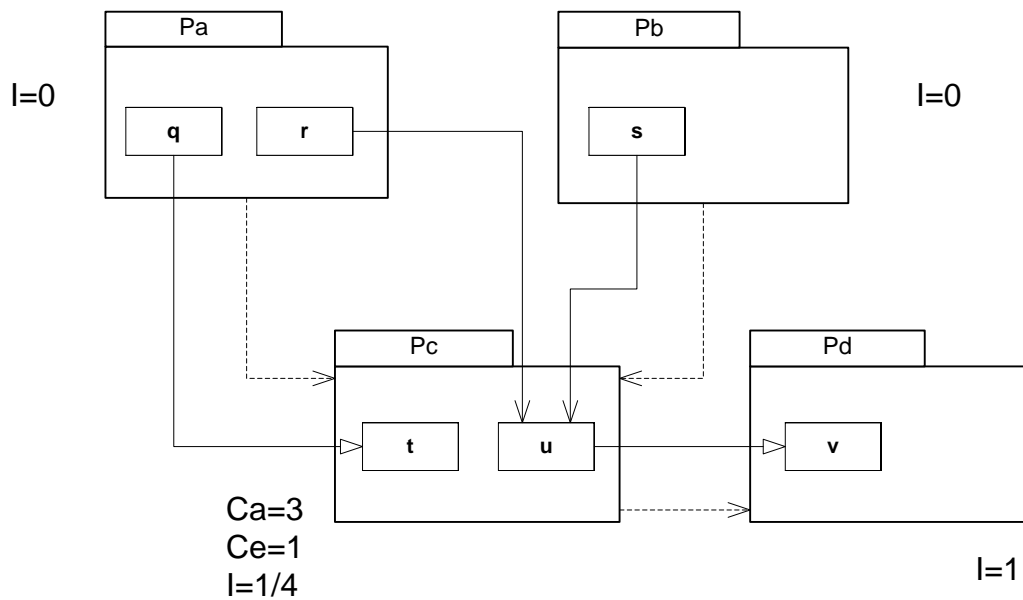


Figure 10: Instability metric calculation example

The I metric of a component should be larger than the I metrics of the components that it depends on. That is, I metrics should decrease in the direction of dependency.

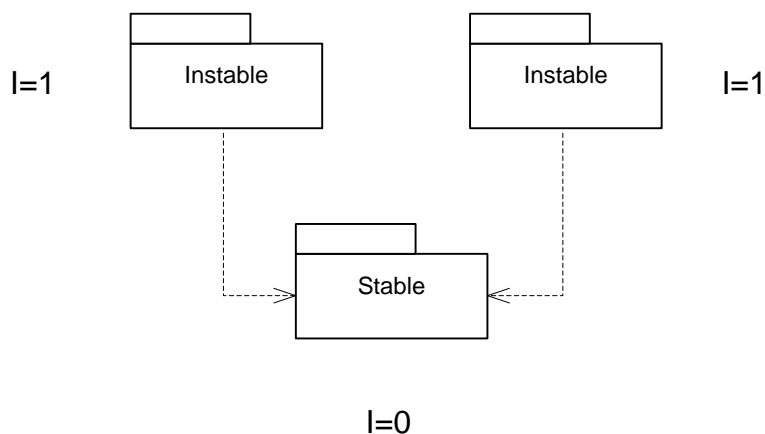


Figure 11: Ideal package configuration

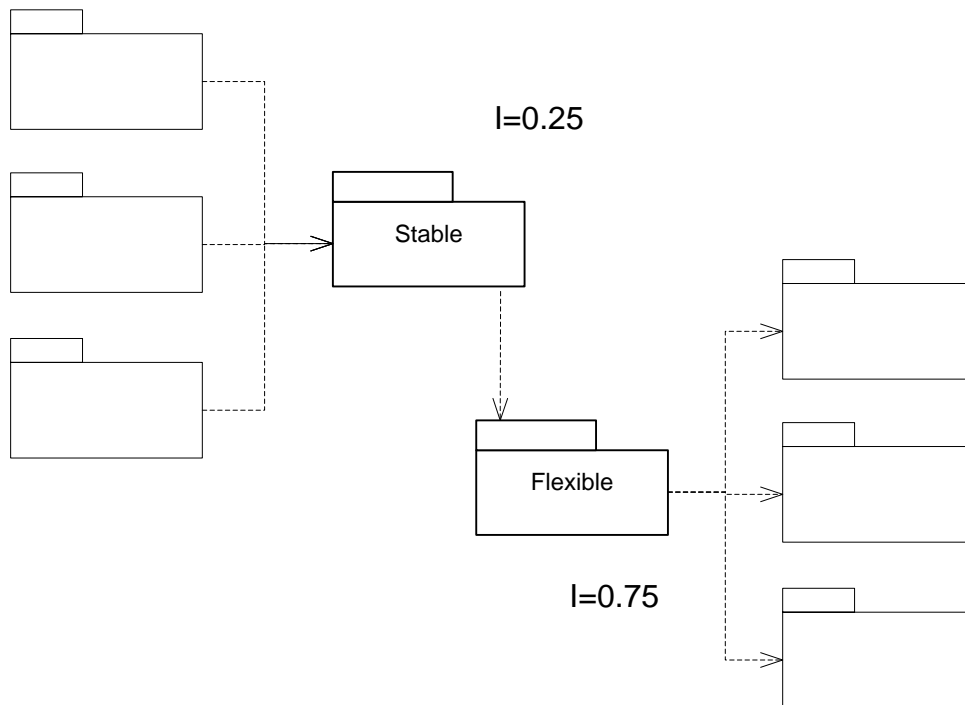


Figure 12: Violation of the stable dependencies principle

3.3.2.3 Stable-abstractions principle

A component should be as abstract as it is stable.

The A metric is a measure of the abstractness of a component. Its value is simply the ratio of abstract classes in a component to the total number of classes in the component, where

- N_c is the number of classes in the component.
- N_a is the number of abstract classes in the component. Remember, an abstract class is a class with at least one abstract method and cannot be instantiated:

$$A(\text{Abstractness}).A = \frac{N_a}{N_c}$$

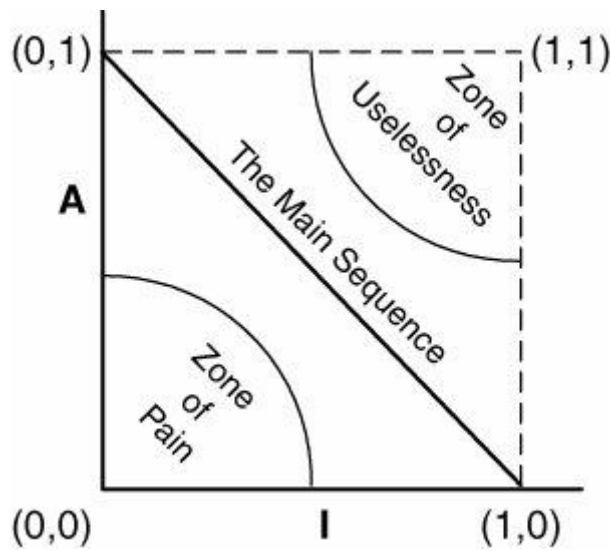


Figure 13: Relation between Instability and Abstractness

Components near (0,0) are highly stable and concrete. These components are difficult to extend because it is not abstract, and very difficult to change because of its stability. Components near (1,1) are fully abstract but has no implementations.

Distance from main sequence:

$$D(\text{Distance}).D = \frac{|A + I - 1|}{\sqrt{2}}$$

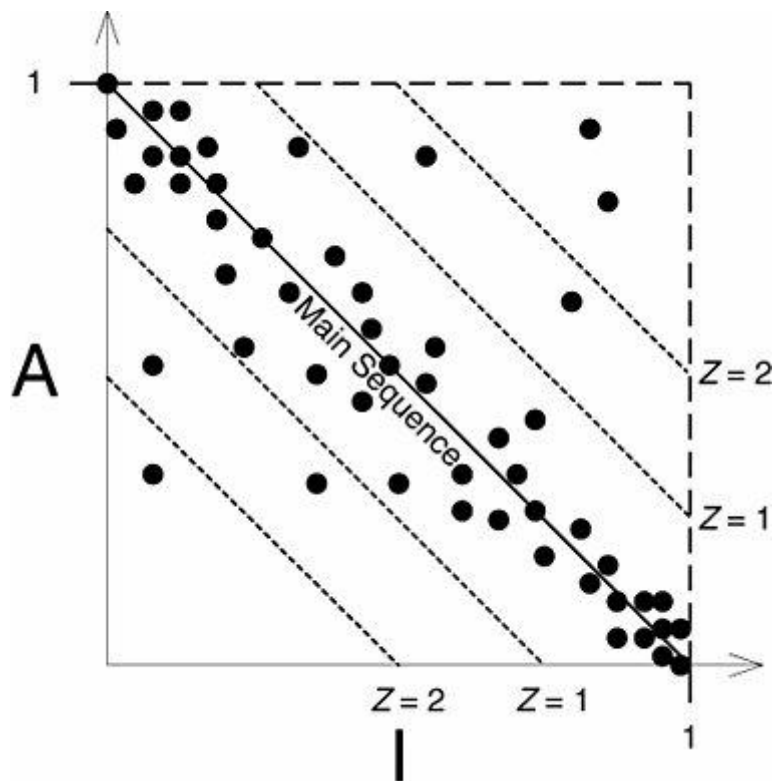


Figure 14: Typical distribution of software packages

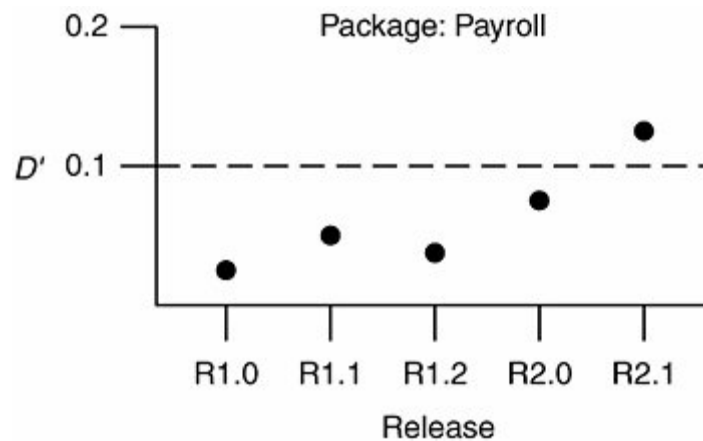


Figure 15: Timeline of a software package distance

4 Code quality

The first barrier against bugs is distributed between all developers. Each developer should follow rules and recommendations (See document 589 “Programming Guidelines”, “Development Rules” 633). Ideally the organization of source files and the structure, style and content of individual files should feel like **if it were created by one highly competent developer**. Also each contributor should follow processes of handling the software code.

Some rules can be checked automatically, others not. Trivial example is the compiler, which checks for programming language syntax – some semantic errors are covered by compiler warnings but the real intent of the code is not.

One goal of WP6 is to automate as much as possible through the lifecycle of the HUMBOLDT project. This is less important while less developers work on the project, but gets more significant as the project goes open. The first step of code check automation is classification of each rule and recommendation if it is theoretically possible to check by software. For example line endings and tabs are trivial to check, misspelling check is hard to implement but not impossible by recent technologies, sensible and easily readable naming of symbols is nearly impossible.

There is another class of recommendations, that can't be linked to concrete single code elements. These are stylistic questions which are below the level of architectural recommendations but more general than single rules. These include distribution of private functions, complexity of code (of course this can be measured by various metrics, but the real intent of the complexity cannot). The main goal of these rules is code readability, i.e. a new developer how fast and easily can find out the inner working of the source code.

Assets for code quality:

- Programming Guidelines (document 589)
- Development Rules (document 633)
- Automated testing framework (see A3.4-D1 State of the art in software architectures)
- Static code analysis tools

- Code coverage tools
- Profiler tools

4.1 Test driven development

Automated test should come from developers together with implementation. Each requirement should be represented by a unit test. The term test driven development refers to the development process where the test are created first, then the code that satisfies the test.

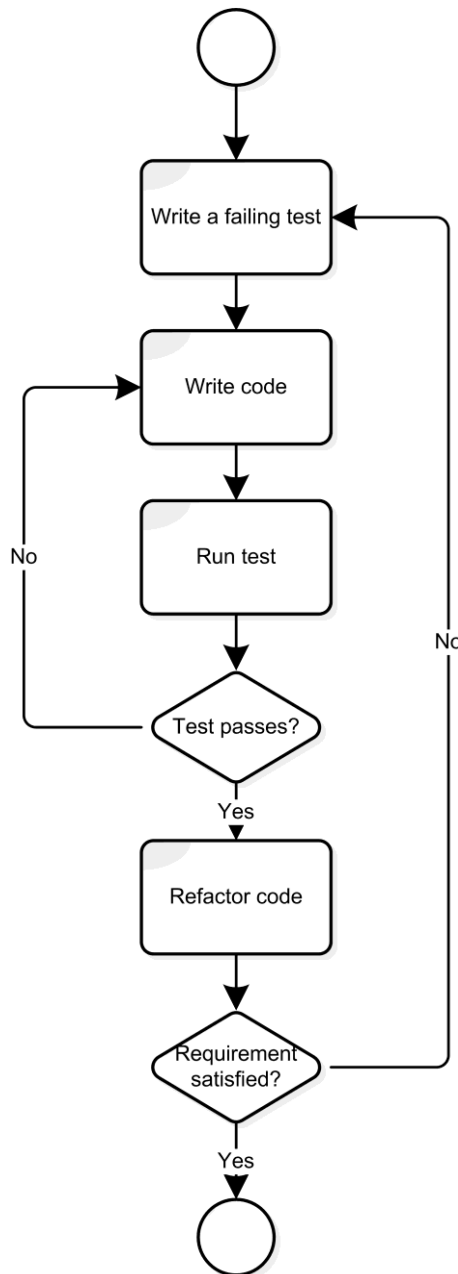


Figure 16: Test driven development process

4.2 Guides and rules from QA perspective

The “Programming Guidelines” and “Development Rules” documents should contain all required rules for quality code development. This is an extract for the most important aspects.

4.2.1 Project Folder Layout

- Standard folder tree for all projects and subprojects. This is independent of the SVN repository tree i.e. this regulates the source code of the project as a standalone entity (written on optical media, uploaded to an FTP server as a tar, etc.). See section xxx.
- The project folder should contain every file and library that is needed for building the product. See build scripts

4.2.2 Compiler Settings

- Settings for the highest warning level.
- Configurations for dev, test, production (debug info, optimization ...).

4.2.3 Build Environment

- Description of the required tools for performing a complete build on each platform.
- The toolset should be minimal. (i.e. MSBuild is included in the .NET redistributable vs. NAnt) Choosing other than default tool should have a good reason.
- No IDEs for building the product.

4.2.4 Build Scripts

- The whole project should have a single build script for every subproject. This master script should be development platform independent and should build the deployment kit for the whole project including JAR files, WAR files, MSI packages install scripts, etc.
- The whole project should have a master deployment script, which can operate without user interaction via command line parameters or configuration files or any other method.
- Each subproject should have a single command line build script for each defined build platform (Windows, Linux). The naming and layout for the build script should be the same for every development platform (Java, .NET)

4.3 Static code analysis tools

4.3.1 Checkstyle

Current Version	4.4
License	
Platform	Java
Integration	Eclipse, IntelliJ IDEA, Maven, Hudson

Input	Source
Technology	

The prime focus of Checkstyle used to be to check code layout issues. It analyses the source code.

4.3.2 Findbugs

Current Version	1.3.6
License	LGPL
Platform	Java
Integration	Eclipse, Maven, Hudson
Input	Bytecode
Technology	Syntax, Dataflow

Findbugs analyses the Java bytecode for possible bugs.

4.3.3 PDM

Current Version	4.2.4
License	BSD-style
Platform	Java
Integration	Eclipse, Maven, Hudson
Input	Source
Technology	Syntax

PDM is aiming to find possible bugs in Java source code. It looks for potential problems like:

- Possible bugs - empty try/catch/finally/switch statements
- Dead code - unused local variables, parameters and private methods
- Suboptimal code - wasteful String/StringBuffer usage
- Overcomplicated expressions - unnecessary if statements, for loops that could be while loops
- Duplicate code - copied/pasted code means copied/pasted bugs

4.3.4 JLint

Current Version	3.1
License	GPL

Platform	Java
Integration	
Input	Bytecode
Technology	Syntax, dataflow

4.4 Terms and definitions

4.4.1 Traceability Analysis – on code

Tracing the source code components to the corresponding design specification, and design specification to source code components is needed. Analysis of identified relationships for correctness, consistency and completeness is required. Criteria are the follows:

- Correctness: Validate the relationship between source code components and design elements,
- Consistency: Verify that the relationship between source code components and design element are specified to a consistent level of detail,
- Completeness:
 - Verify that all source code components are traceable from design elements,
 - Verify that all design elements are traceable to the source code components.

4.4.2 Source code and documentation evaluation

This step means the evaluation of source code components for correctness, consistency, completeness, accuracy, readability and testability. Criteria are the follows:

- Correctness:
 - Verify and Validate that the source code component satisfies the software design,
 - Verify that the source code components comply with applicable standards, references, regulations, policies and business rules,
 - Validate the source code component sequences of states and state changes,
 - Validate that the flow of data and control satisfy functionality and performance requirements,
 - Validate data usage and format,
 - Assess the appropriateness of coding methods and standards,
- Consistency:
 - Verify that all terms and code concepts are documented consistently,
 - Verify that there is internal consistency between the source code components,

- Validate external consistency with the software design and requirements,
- **Completeness:**
 - Verify that the following elements are in the source code, within the assumptions and constraints of the system:
 - Functionality (e.g. algorithms, state/mode definitions, input/output validation, exception handling, reporting and logging),
 - Process definition and scheduling,
 - Hardware, software and user interface descriptions,
 - Performance criteria (e.g. timing, sizing, speed, capacity, accuracy, precision, safety, and security),
 - Critical configuration data,
 - System, device, and software control (e.g. initialisation, transaction and state monitoring, self-testing),
 - Verify that the source code documentation satisfies specified configuration management procedures,
- **Accuracy:**
 - Validate the logic, computation, and interface precision (e.g. truncation and rounding) in the system environment,
- **Readability:**
 - Verify that the documentation is legible, understandable, and unambiguous to the intended audience,
 - Verify that the documentation defines all acronyms, mnemonics, abbreviations, terms and symbols, and design language if any.
- **Testability:**
 - Verify that there are objective acceptance criteria for validating each source code component,
 - Verify that each source code component is testable against objective acceptance.

4.4.3 Interface analysis - code

In interface analysis (in implementation phase) verification and validation of that, the software source code interfaces with hardware, user, operator and other systems are correct, consistent, complete, accurate and testable is needed. Criteria are the follows:

- **Correctness:** Validate the external and internal software interface code in the context of system requirements,
- **Consistency:** Verify that the interface code is consistent between source code components and to external interfaces (e.g. hardware, user, operator and other software),

- **Completeness:** Verify that each interface is described and includes data format and performance criteria (e.g. timing, bandwidth, accuracy, safety and security),
- **Accuracy:** Verify that each interface provides information with the required accuracy,
- **Testability:** Verify that there are objective acceptance criteria for validating the interface code.

4.4.4 System test case analysis

In system test case analysis the verification of system test cases is needed, that they comply with HUMBOLDT defined test document purpose, format, and content. Validation of system test cases is required, that they satisfy the criteria in System test plan.

4.4.5 Software final qualification test case analysis

In software final qualification test case analysis the verification of software final qualification test cases is needed, that they comply with HUMBOLDT defined test document purpose, format, and content. Validation of software final qualification test cases is required, that they satisfy the criteria in software final qualification test plan.

4.4.6 Software integration test case analysis

In software integration test case analysis the verification of software integration test cases is needed, that they comply with HUMBOLDT defined test document purpose, format, and content. Validation of software integration test cases is required, that they satisfy the criteria in software integration test plan.

4.4.7 Acceptance test case analysis

In acceptance test case analysis the verification of acceptance test cases is needed, that they comply with HUMBOLDT defined test document purpose, format, and content. Validation of acceptance test cases is required, that they satisfy the criteria in acceptance test plan.

4.4.8 Software integration test procedure analysis

In software integration test procedure analysis the verification of software integration test procedures is needed, that they comply with HUMBOLDT defined test document purpose, format, and content. Validation of software integration test procedures is required, that they satisfy the criteria in software integration test plan.

4.4.9 Software integration test results analysis

Usage of developers' integration test results is required to verify that the software components are integrated correctly. Verification of test results is needed, that they trace to the test criteria established by the test traceability in the test planning documents. Documentation of discrepancies between actual and expected results is required.

4.4.10 Component test case analysis

In component test case analysis the verification of component test cases is needed, that they comply with HUMBOLDT defined test document purpose, format, and content. Validation of component test cases is required, that they satisfy the criteria in component test plan.

4.4.11 System test procedure analysis

In system test procedure analysis the verification of system test procedures is needed, that they comply with HUMBOLDT defined test document purpose, format, and content. Validation of system test procedures is required, that they satisfy the criteria in system test plan.

4.4.12 Software final qualification test procedure analysis

In software final qualification test procedure analysis the verification of software final qualification test procedures is needed, that they comply with HUMBOLDT defined test document purpose, format, and content. Validation of software final qualification test procedures is required, that they satisfy the criteria in software final qualification test plan.

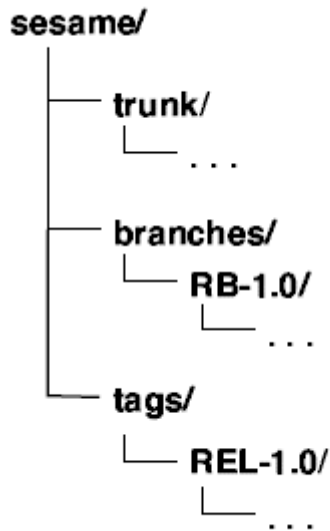
5 Integration quality

One of the most error-prone phases of software production is the integration of the work of separate groups and individuals. The two infrastructural elements of code integration are the Version Control System and the Continuous Integration System.

5.1 Version control system (SVN)

The “Programming Guidelines” and “Development Rules” documents should contain all required rules for SVN usage. This is an extract for the most important aspects.

- Never check in temporary build files, build output files, personal tool settings, local folder settings or any file or folder, which is not required to build the product.
- Never check in files that are in conflict state.
- Strategy for handling required personal settings (for example database access strings, ...)
- There are no other folders on the root of the repository other the trunk, branches, tags.



5.1.1 Trunk

- **What the Trunk contains is the software product!** If the project stops for any reason the trunk contains what we achieved. If the product is released (for a milestone, beta, production ...) the trunk is simply checked out (see release tags) and a full build is performed.
- Never break the build on the trunk! All projects and subprojects, on every platform, should build and should pass all automated tests.
- Never commit if the trunk is broken.
- Never commit without reason, don't "massage" the code. For example if the product is in a mature and tested state, don't even clean up whitespaces. Save that for the next release.

5.1.1.1 States of the trunk

- Definition of the states of the trunk during release cycles (instable, stable, closed).
- Additional rules for stable and closed states.

5.1.2 Tags and Branches

- Each branch and tag must contain the whole project, including all libraries, setup and auxiliary files that are needed for a full build (branching is cheap in SVN).

5.1.2.1 Naming Convention for Tags and Branches

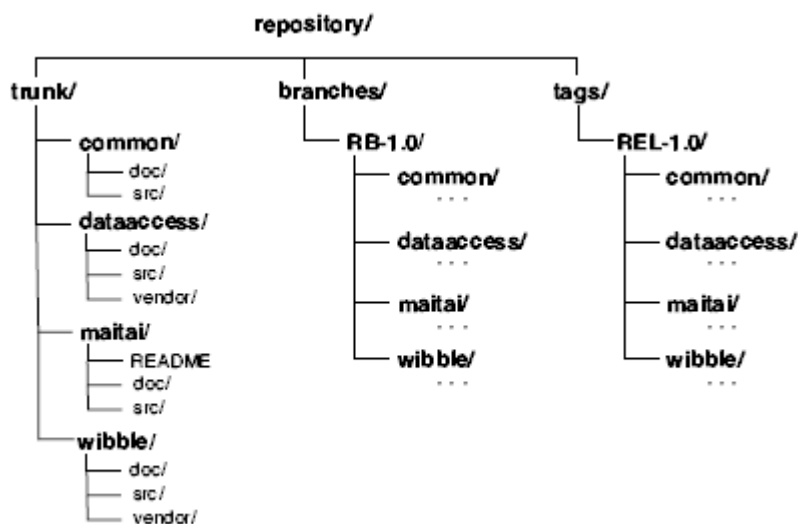
Example:

Thing to Name	Name Style	Examples
Release branch	RB- <i>rel</i>	RB-1.0 RB-1.0.1a
Releases	REL- <i>rel</i>	REL-1.0 REL-1.0.1a
Bug fix branches	BUG- <i>track</i>	BUG-3035 BUG-10871
Pre-bug fix	PRE- <i>track</i>	PRE-3035 PRE-10871
Post-bug fix	POST- <i>track</i>	POST-3035 POST-10871
Developer experiments	TRY- <i>initials-desc</i>	TRY-MGM-cache-pages TRY-MR-neo-persistence

5.1.2.2 Release Tags

- Each release (milestone, beta, production) must be tagged.
- The tag must contain the whole project.
- The build for the release should be performed on the code clean checked out from the release tag.
- Tags are read only. Never commit to a tag. (Setup SVN access rules to make it impossible.)

5.1.2.3 Release Branches



- If a release have to be patched the release tag should be branched.
- Only create a release branch if it is necessary.
- Release branches start as stable. All rules to the stable branch apply to a release branch.
- Integrate form release branch to trunk if feasible.

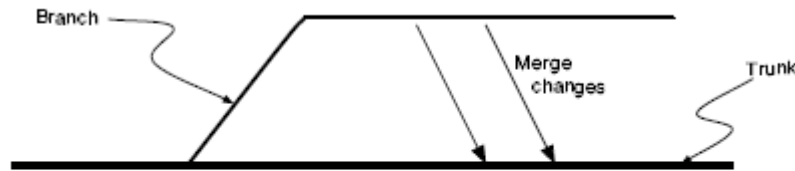


Figure 9.3: Release Branch Merges to the Trunk

5.1.2.4 Private branches

- Private branches are for storing and exchanging immature code between developers, i.e. a private branch must not follow the rules for the trunk. (The rules and recommendations for quality coding should be followed regardless.)
- Anyone can create a private branch.
- Larger multi developer efforts should be done on a private branch. The integration to trunk should be done by the leader of the group.
- Private branches should not live long.
- Private branches should follow the defined naming conventions.
- Private branches can be used implicitly as private tags.
- All developers should commit their work to a private branch before leaving the project for a longer period of time. (for example vacation)

5.2 Continuous Integration

As lead of WP6, FÖMI operates the Continuous Integration Server for HUMBOLDT (<http://humboldt.fomi.hu>). Hudson was chosen for its flexibility, configurability and simplicity.

5.2.1 Concept of Continuous Integration

Definition by Martin Flower inventor of the concept: *Continuous Integration is a software development practice where members of a team integrate their work frequently; usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible. Many teams find that this approach leads to significantly reduced integration problems and allows a team to develop cohesive software more rapidly.*

A Continuous Integration Server monitors the source control server (in this case Subversion) for changes. If change occurs checks out or updates the source code and performs full or incremental build. Also automated tests are executed. Parses the result of the build and tests and publishes the results in various ways

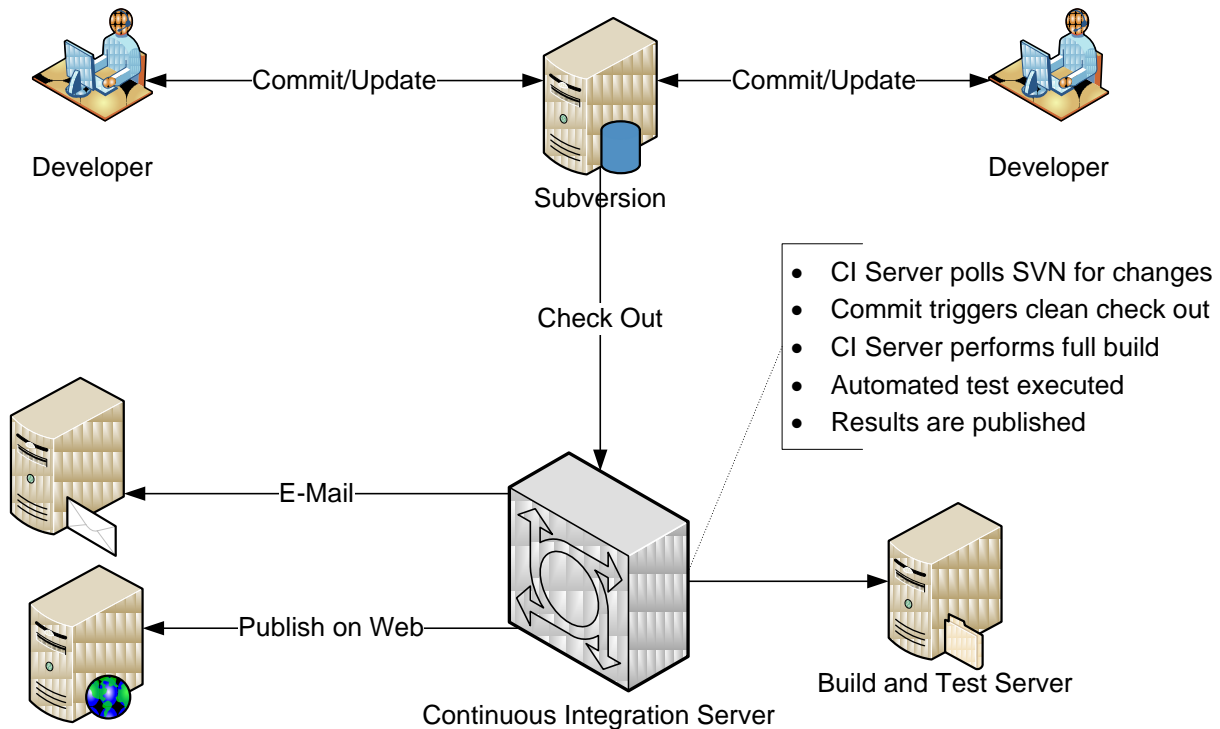


Figure 17: Overview of continuous integration

The CI server connects to the development infrastructure through the following interfaces:

- Access to Source Code Management server. This includes interpretation of tags or branches, possibly monitoring multiple branches of the same project simultaneously.
- Knowledge of the build process, to be able to execute it and parse its results.
- Execution of automated tests and the ability to parse its results. Unit tests might be included in the build script but most CI servers are capable of carrying out tests without it.
- The CI server must have access to the build and if tested, the deployment environment. Depending on the CI server architecture this might be a separate server or the CI server itself must have the necessary infrastructure.

5.2.2 CI System Setup

The CI server is operated by FÖMI. It is available on <http://humboldt.fomi.hu>. It is a standard setup of Hudson (<http://hudson-ci.org/>). It is running on a virtual Ubuntu server with Tomcat and Apache.

The build server has complete build environment for the HUMBOLDT components. It checks out code from the HUMBOLDT SVN repository and uses the HUMBOLDT Artifactory for as POM repository third party libraries.

5.3 Code Analysis Server

As the next step of improving the code quality server infrastructure, WP6 operates an instance of the open source Sonar code quality platform (<http://sonarsource.org>). Sonar is a platform for integrated monitoring and tracking of:

- code coverage
- standard metrics
- static code analysis results

5.3.1 Sonar Overview

Sonar analysis is performed in the last phase of builds.

```
[INFO] -----  
[INFO] Building HUMBOLDT Context Service Component  
[INFO]   task-segment: [sonar:sonar] (aggregator-style)  
[INFO] -----  
[INFO] [sonar:sonar {execution: default-cli}]  
[INFO] Sonar host: http://humboldt.fomi.hu/sonar  
[INFO] Sonar version: 1.12  
[INFO] [sonar-core:internal {execution: default-internal}]  
[INFO] Database dialect class org.sonar.api.database.dialect.MySql  
[INFO] ----- Analyse HUMBOLDT Context Service Component...  
[INFO] Selected quality profile : HUMBOLDT profile, language=java  
[INFO] Configure maven plugins...  
[INFO] Executing sensor class org.sonar.plugins.core.sensors.JavaSourceImporter  
[INFO] Executing sensor class org.sonar.plugins.core.sensors.AsynchronousMeasuresSensor  
[INFO] Executing sensor class org.sonar.plugins.squid.SquidSensor  
[INFO] Executing sensor class org.sonar.plugins.surefire.SurefireSensor  
[INFO] parsing /var/lib/hudson/jobs/HUMBOLDT Context Service Component  
(trunk)/workspace/trunk/target/surefire-reports  
[INFO] Executing sensor class org.sonar.plugins.cpd.CpdSensor  
[INFO] Executing sensor class org.sonar.plugins.core.sensors.ProfileSensor  
[INFO] Executing sensor class org.sonar.plugins.core.sensors.ProjectLinksSensor  
[INFO] Executing sensor class org.sonar.plugins.core.sensors.VersionEventsSensor  
[INFO] Execute decorators...  
[INFO] ANALYSIS SUCCESSFUL, you can browse http://humboldt.fomi.hu/sonar  
[INFO] Optimizing database...  
[INFO] Database purged in 600 ms.  
[INFO] -----  
[INFO] BUILD SUCCESSFUL  
[INFO] -----  
[INFO] Total time: 29 seconds  
[INFO] Finished at: Mon Sep 27 00:17:40 CEST 2010  
[INFO] Final Memory: 12M/22M  
[INFO] -----  
Finished: SUCCESS
```

The static code analysis and the collection of code coverage data is integrated into the Maven build process and the results are pushed to the Sonar database.

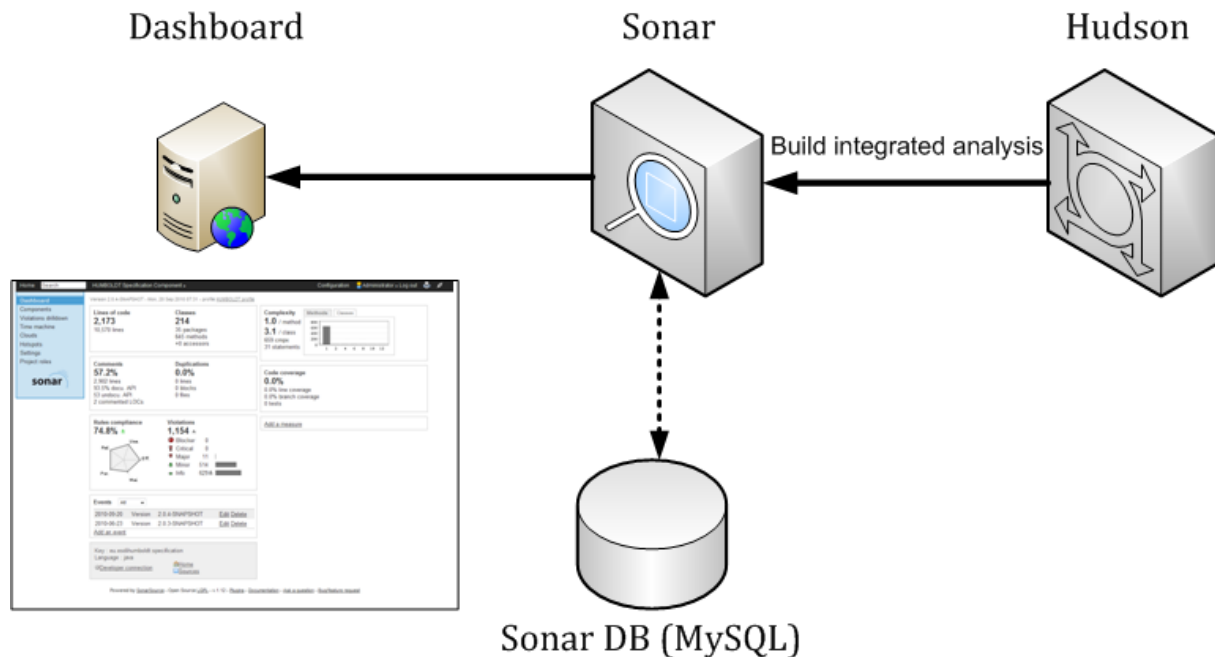


Figure 18: Overview of Sonar operation

Sonar then provides comprehensive analysis of results via its web interface. The web interface allows:

- hierarchical browsing of components, classes and methods
- charting trends of metrics, rules compliance and code coverage
- browsing source code overlaid with coverage and rule violations

5.3.1.1 Code Coverage

Measuring code coverage is the process of detecting elements of source code that are executed during a certain drive of test code. In our case the execution of automated test cases drives the process of code coverage measurement. The measured elements of code can be:

- lines
- branches
- statements
- conditions

The measured coverage can be expressed by the percentage of code element covered against all elements.

There are no exact figures to be defined as acceptance criteria for coverage results. Code coverage rather indicates the strategy of development. Low coverage indicates a development process that is not driven by exact definitions of requirements. On the other hand, a very high coverage might still result software that does not satisfies the needs of the end user.

Trends of coverage percentage should stay steady as the code grows during development indicating that new code introduced to the product also comes with its tests.

5.3.1.2 Rules Compliance

Sonar integrates the most common static code analysis tools into a common platform. These include:

- Checkstyle
- PMD
- Findbugs

Static code analysis is a white box method of testing. These tools evaluate either the source code or the byte code without executing it, looking for common patterns of problems or errors. These are all semantic errors since the compiler will not allow any syntax error. Still a very large number of issues are commonly repeated during software development. Each rule by its nature is categorized into one of the following categories:

- Efficiency
- Maintainability
- Portability
- Reliability
- Usability

Rules are also categorized by severity into the following categories:

- Blocker
- Critical
- Major
- Minor
- Info

The collection of applied rules together with the each rule's severity attribute is called quality profile.

5.3.1.3 General Metrics

General metrics are mainly statistical counts of various code elements and some derived metrics calculated against results of code coverage and rule compliance. These include:

- Complexity
 - Complexity /class
 - Complexity /method
 - Complexity
 - Uncovered complexity
- Documentation
 - Commented LOCs

- Comments (%)
 - Comment lines
 - Public documented API (%)
 - Public undocumented API
- Duplication
 - Duplicated blocks
 - Duplicated files
 - Duplicated lines
 - Duplicated lines (%)
- Rule categories
 - Efficiency
 - Maintainability
 - Portability
 - Reliability
 - Usability
- Rules
 - Blocker violations
 - Critical violations
 - Info violations
 - Major violations
 - Minor violations
 - Rules compliance
 - Violations
 - Weighted violations
- Size
 - Accessors
 - Classes
 - Lines
 - Methods
 - Lines of code
 - Files

- Statements
- Packages
- Public API
- Tests
 - Branch coverage
 - Conditions to cover
 - Coverage
 - Unit tests duration
 - Line coverage
 - Lines to cover
 - Skipped unit tests
 - Unit test errors
 - Unit test failures
 - Unit tests
- Unit test success (%)
- Uncovered conditions
- Uncovered lines

5.3.2 Quality Profiles

The HUMBOLDT project has its own quality profile since the coding standard and development rules are unique to each development project so is to the HUMBOLT project. The following table defines the current quality profile apply the HUMBOLDT components:

title	category	plugin	priority
Anon Inner Length	Maintainability	checkstyle	MAJOR
Avoid Array Loops	Efficiency	pmd	MAJOR
Avoid Assert As Identifier	Portability	pmd	MAJOR
Avoid Calling Finalize	Usability	pmd	MAJOR
Avoid Catching NPE	Reliability	pmd	MAJOR
Avoid Catching Throwable	Reliability	pmd	CRITICAL
Avoid Decimal Literals In Big Decimal Constructor	Reliability	pmd	MAJOR
Avoid Duplicate Literals	Maintainability	pmd	MAJOR
Avoid Enum As Identifier	Portability	pmd	MAJOR
Avoid Instanceof Checks In Catch Clause	Maintainability	pmd	MINOR
Avoid Print Stack Trace	Usability	pmd	MAJOR
Avoid Rethrowing Exception	Maintainability	pmd	MAJOR
Avoid Throwing Null Pointer Exception	Usability	pmd	MAJOR

Avoid Throwing Raw Exception Types	Maintainability	pmd	MAJOR
Big Integer Instantiation	Efficiency	pmd	MAJOR
Boolean Expression Complexity	Maintainability	checkstyle	MAJOR
Boolean Instantiation	Efficiency	pmd	MAJOR
Broken Null Check	Reliability	pmd	CRITICAL
Class Cast Exception With To Array	Reliability	pmd	MAJOR
Clone Throws Clone Not Supported Exception	Reliability	pmd	MAJOR
Close Resource	Reliability	pmd	MAJOR
Collapsible If Statements	Maintainability	pmd	MINOR
Compare Objects With Equals	Reliability	pmd	MAJOR
Constant Name	Usability	checkstyle	MINOR
Constructor Calls Overridable Method	Reliability	pmd	MAJOR
Cyclomatic Complexity	Maintainability	checkstyle	MAJOR
Default Comes Last	Usability	checkstyle	MAJOR
Design For Extension	Reliability	checkstyle	MINOR
Dont Import Java Lang	Maintainability	pmd	MINOR
Dont Import Sun	Portability	pmd	MINOR
Double Checked Locking	Reliability	checkstyle	MAJOR
Empty Finalizer	Maintainability	pmd	MAJOR
Empty Finally Block	Maintainability	pmd	CRITICAL
Empty If Stmt	Maintainability	pmd	CRITICAL
Empty Statement	Maintainability	checkstyle	MINOR
Empty Static Initializer	Maintainability	pmd	MAJOR
Empty Switch Statements	Maintainability	pmd	MAJOR
Empty Synchronized Block	Maintainability	pmd	CRITICAL
Empty Try Block	Maintainability	pmd	MAJOR
Empty While Stmt	Maintainability	pmd	CRITICAL
Equals Hash Code	Reliability	checkstyle	CRITICAL
Equals Null	Reliability	pmd	CRITICAL
Exception As Flow Control	Usability	pmd	MAJOR
Final Class	Usability	checkstyle	MAJOR
Final Field Could Be Static	Efficiency	pmd	MINOR
Finalize Does Not Call Super Finalize	Reliability	pmd	MAJOR
Finalize Overloaded	Reliability	pmd	MAJOR
For Loops Must Use Braces	Usability	pmd	MAJOR
Hidden Field	Usability	checkstyle	MAJOR
Hide Utility Class Constructor	Efficiency	checkstyle	MAJOR
Idempotent Operations	Efficiency	pmd	MAJOR
If Else Stmts Must Use Braces	Usability	pmd	MAJOR
If Stmts Must Use Braces	Usability	pmd	MAJOR
Illegal Throws	Maintainability	checkstyle	MAJOR
Inefficient String Buffering	Efficiency	pmd	MAJOR
Inner Assignment	Usability	checkstyle	MAJOR
Instantiation To Get Class	Usability	pmd	MAJOR

Integer Instantiation	Portability	pmd	MAJOR
Local Final Variable Name	Usability	checkstyle	MAJOR
Local Variable Name	Usability	checkstyle	MAJOR
Loose coupling	Maintainability	pmd	MAJOR
Magic Number	Reliability	checkstyle	MINOR
Member Name	Usability	checkstyle	MAJOR
Method Name	Usability	checkstyle	MAJOR
Missing Static Method In Non Instantiatable Class	Maintainability	pmd	MAJOR
Modifier Order	Usability	checkstyle	MINOR
Naming - Avoid dollar signs	Usability	pmd	MINOR
Naming - Class naming conventions	Usability	pmd	MAJOR
Naming - Method with same name as enclosing class	Usability	pmd	MAJOR
Naming - Suspicious Hashcode method name	Usability	pmd	MAJOR
Naming - Suspicious constant field name	Usability	pmd	MAJOR
Naming - Suspicious equals method name	Usability	pmd	CRITICAL
Ncss Method Count	Maintainability	pmd	MAJOR
Ncss Type Count	Maintainability	pmd	MAJOR
Package Name	Usability	checkstyle	MAJOR
Parameter Assignment	Usability	checkstyle	MAJOR
Parameter Name	Usability	checkstyle	INFO
Preserve Stack Trace	Maintainability	pmd	MAJOR
Redundant Modifier	Maintainability	checkstyle	MINOR
Redundant Throws	Maintainability	checkstyle	MINOR
Replace Enumeration With Iterator	Portability	pmd	MAJOR
Replace Hashtable With Map	Portability	pmd	MAJOR
Replace Vector With List	Portability	pmd	MAJOR
Security - Array is stored directly	Reliability	pmd	CRITICAL
Signature Declare Throws Exception	Maintainability	pmd	MAJOR
Simplify Boolean Expression	Maintainability	checkstyle	MAJOR
Simplify Boolean Return	Maintainability	checkstyle	MAJOR
Simplify Conditional	Maintainability	pmd	MAJOR
Singular Field	Maintainability	pmd	MINOR
Static Variable Name	Usability	checkstyle	MAJOR
String Buffer Instantiation With Char	Reliability	pmd	MAJOR
String Instantiation	Efficiency	pmd	MAJOR
String Literal Equality	Reliability	checkstyle	MAJOR
String To String	Maintainability	pmd	MAJOR
System Println	Usability	pmd	MAJOR
Unconditional If Statement	Maintainability	pmd	CRITICAL
Unnecessary Case Change	Efficiency	pmd	MINOR
Unnecessary Local Before Return	Efficiency	pmd	MAJOR
Unused Imports	Maintainability	checkstyle	INFO
Unused Modifier	Maintainability	pmd	INFO
Unused Null Check In Equals	Maintainability	pmd	MAJOR

Unused Private Field	Maintainability	pmd	MAJOR
Unused formal parameter	Maintainability	pmd	MAJOR
Unused local variable	Maintainability	pmd	MAJOR
Unused private method	Maintainability	pmd	MAJOR
Use Array List Instead Of Vector	Efficiency	pmd	MAJOR
Use Arrays As List	Efficiency	pmd	MAJOR
Use Correct Exception Logging	Maintainability	pmd	MAJOR
Use Index Of Char	Efficiency	pmd	MAJOR
Use String Buffer Length	Efficiency	pmd	MINOR
Useless Operation On Immutable	Reliability	pmd	CRITICAL
Useless Overriding Method	Maintainability	pmd	MAJOR
Useless String Value Of	Efficiency	pmd	MINOR
Visibility Modifier	Maintainability	checkstyle	MAJOR
While Loops Must Use Braces	Usability	pmd	MAJOR

5.3.3 Server Implementation

A Sonar code analysis server is implemented in the same virtual machine as the Hudson server, in a minimal server edition of Ubuntu Linux 9.04. Sonar runs standalone in its integrated Winstone servlet container and has public access through AJP connection via an Apache web server. As a backend storage Sonar requires a relational database server. It is served by a default setup of MySQL 5.1. As with the Continuous Integration Server, all components of the infrastructure are open source components.

6 Issue Tracking

During QA processes the following types of issues need to be managed:

- requirements
- bugs
- change requests

Currently three tools are used for these purposes with some overlap:

- Volere (<http://humboldt.etra.es/>) for collecting requirements
- Polarion for managing internal bugs and some requirements
- Redmine (<http://www.esdi-community.eu>)

It is desirable to be able to connect requirements, issues and source code.

6.1 Requirements

The detail and level of a requirement can be multiple levels:

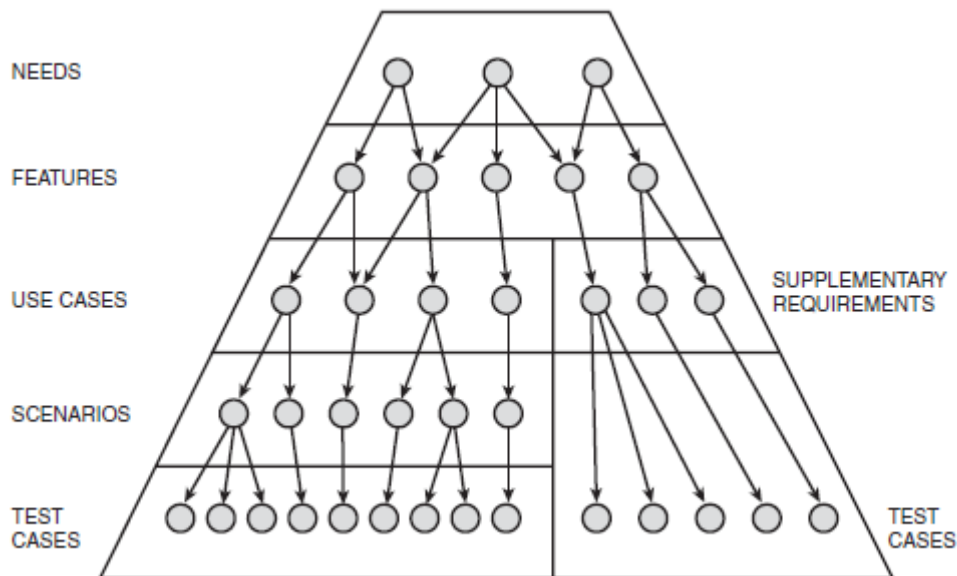


Figure 19: The requirement pyramid

Note that on this figure the term “scenario” is different from the HUMBOLDT term. Going down each level is more specific. To be able to test against a requirement it needs to be atomic. It is very important that the requirement management tool should support the hierarchy of requirements.

It would be beneficial to use one tool for all kinds of issue tracking, including requirement management and bug tracking.

Although Polarion has the largest feature set by far, it is criticised for its complexity. Also it is commercial software licensed to the HUMBOLDT project. As the project ends its licensing is questionable. Polarion currently has some requirements and bugs. Also requirements are hierarchically arranged.

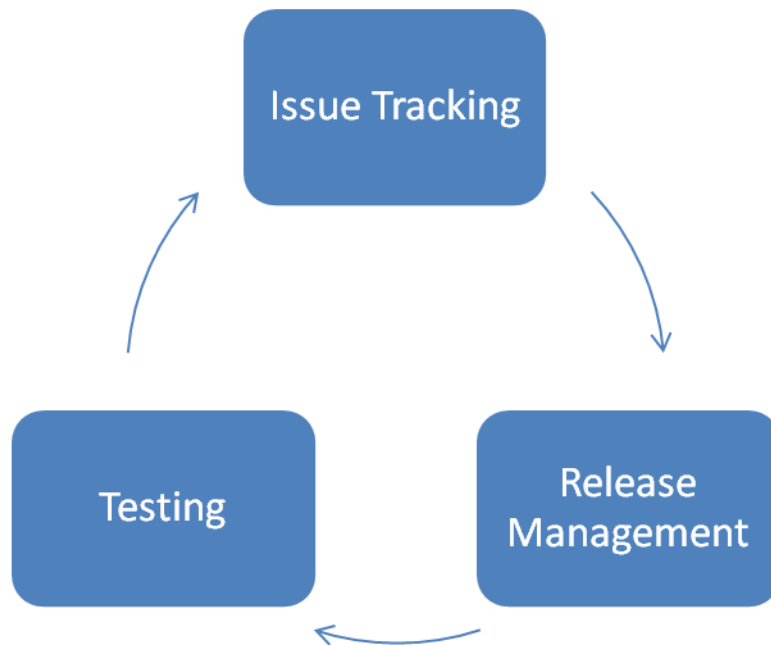
Volere’s primary goal was collecting requirements. It is too simple for managing requirements through its whole lifecycle. Volere contains most of the requirements defined for HUMBOLDT. Also most of these requirements need further breakdown, as they are not atomic.

Redmine is candidate for being selected. It has some bugs, also has a workspace (issue tracking, wiki, news, releases) for the components. Redmine has limited capabilities compared to Polarion, but is has a very intuitive and easy to use user interface and sufficient features required for requirement management. Some concepts are not directly supported (such as hierarchical arrangement of issues) but easily achieved with issue description policies.

7 Release Management

For the QA process the following connections with the development process need to be defined:

- *Connection between issue tracking, general QA and releasing a software product.*



- *Description of processes and strategy for defining feature sets of releases.*
- *Prioritization of bugs and requirements*
- *Setting release dates. Periodical? Feature driven? Other?*
- *Definition of release types (milestone, beta, production, ...).*

Versioning

- Processes for setting a version number for a release (1.8 or 2.0 ?)
- Evaluation of strategies for harmonizing independent software component versioning. Should all components go by the same most significant version number?
 - Framework, Components, Platforms
 - Alpha, Beta, RC, RTM

Release master

- *Definition of persons or a board for making decisions on the above questions.*
- Prioritization of bugs and requirements
- Establish release dates