quality

COLLABORATORS			
	TITLE : quality		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		August 25, 2021	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

1 Quality **1**
1.1 Product Quality . . . 2
1.1.1 Domains . . . 3
1.1.2 Product Quality Taxonomy . . . 4
1.1.3 Quality Characteristics . . . 6
1.1.3.1 Quality Subcharacteristics I . . . 7
1.1.3.1.1 Quality Tree for Performance Efficiency . . . 9
1.1.3.1.1.1 Scenarios relating to Performance Efficiency . . . 10
1.1.3.1.2 Strategies for Performance Efficiency . . . 12
1.1.3.1.2.1 Memory Controlling Patterns/Tactics . . . 13
1.1.3.1.2.2 Memory Management Patterns/Tactics . . . 15
1.1.3.2 Quality Subcharacteristics II . . . 17
1.1.3.2.1 Strategies for Maintainability . . . 19
1.1.3.2.1.1 Simplicity Terms . . . 21
1.1.3.2.1.2 SOLID Principles . . . 22
1.1.3.2.2 Strategies for Reliability . . . 23
1.1.3.2.2.1 Tactics for Maturity . . . 26
1.1.3.2.3 Strategies for Security . . . 27
1.1.3.2.3.1 Tactics for Partitioning . . . 30
1.1.3.3 Quality Subcharacteristics III . . . 31
1.1.3.3.1 Strategies for Safety . . . 32
1.1.3.3.1.1 Tactics for Safety . . . 33
1.2 Process Quality . . . 34
1.2.1 Standard Process Models . . . 36
1.2.2 V Model . . . 36
1.2.3 Standard Methods . . . 38
1.2.3.1 Security Analysis . . . 39
1.3 Bibliography . . . 41

Chapter 1

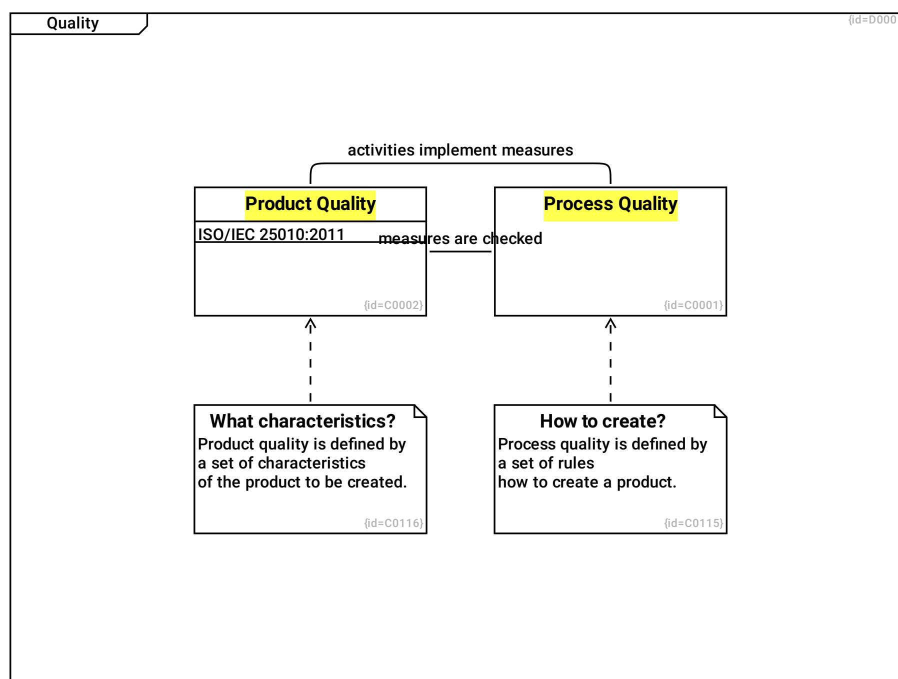
Quality

Quality is a set of characteristics of a product. These characteritics can be concretised to quality attributes (measureable properties of the product).

To achieve quality at development and during operation, one can set up processes to ensure that each development or operation step implements mechanisms to reach a defined level of quality.

In the end, the product has to encompass technical measures to provide expected quality characteristics.

(c) 2019-2020 Andreas Warnke License: Choose either Apache-2.0 or Creative Commons Attribution (BY) Licence



Process Quality C0001

Process quality can be measured by analyzing the reports and other workproducts that have been created during different phases at engineering and during operation.

activities implement measures --> Product Quality R0138

How to create? C0115

Process quality is defined by a set of rules how to create a product.

--> Process Quality R0131

Product Quality C0002

Product quality refers to characteristics that can be measured by analyzing the product that was created and that is in use.

ISO/IEC 25010:2011 F0004

see Section 1.3: Bibliography

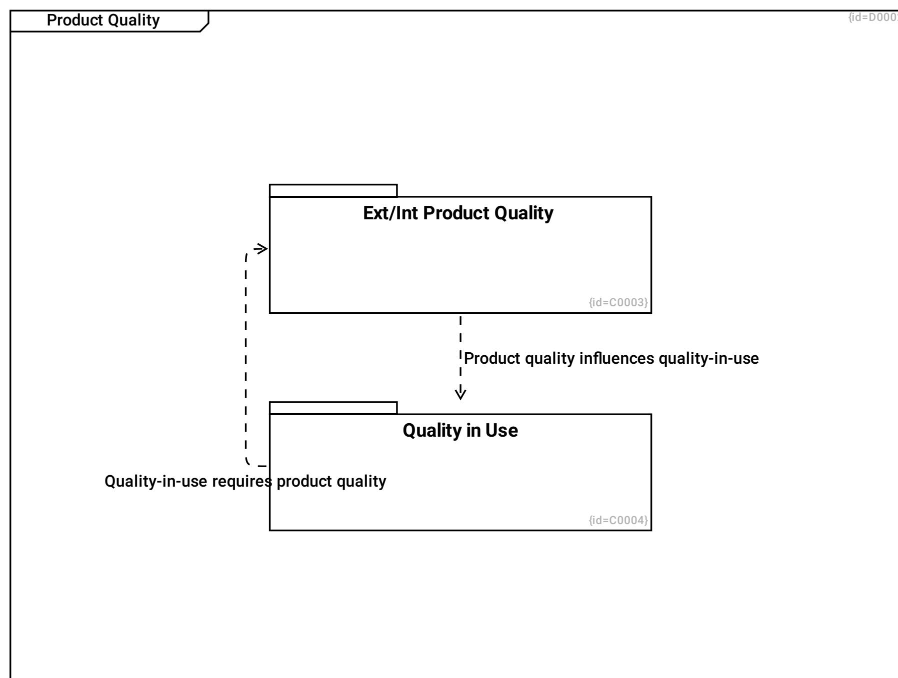
measures are checked --> Process Quality R0137

What characteristics? C0116

Product quality is defined by a set of characteristics of the product to be created.

--> Product Quality R0132

1.1 Product Quality



Ext/Int Product Quality C0003

Product quality are internal and externally visible qualities, such as memory consumption or startup timings.

Product quality influences quality-in-use --> Quality in Use R0139

Quality in Use C0004

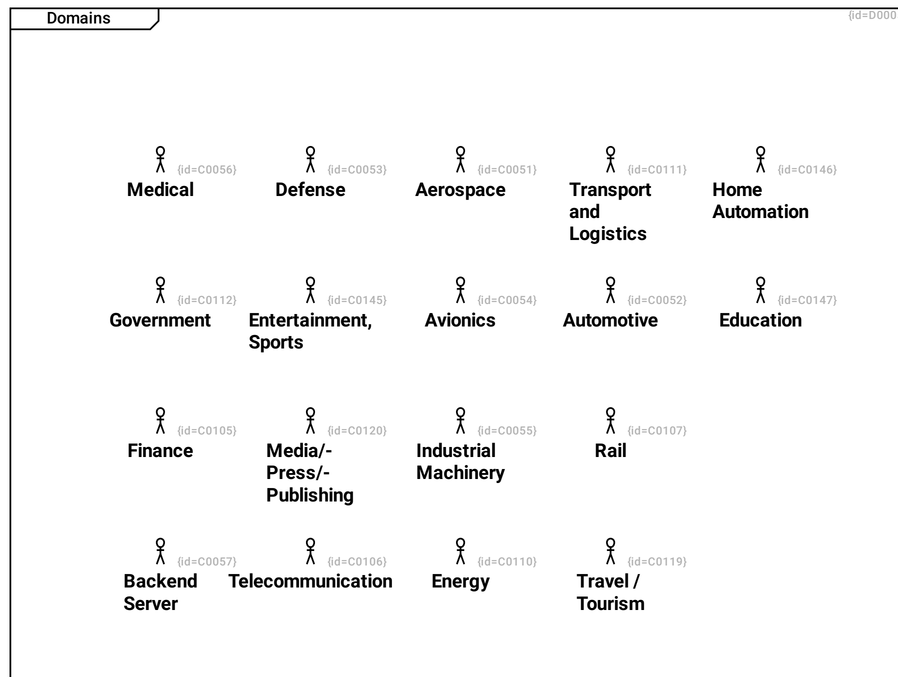
Quality in use can be measured when the product is already in use, e.g. the percentage of satisfied customers can be determined.

Quality-in-use requires product quality --> Ext/Int Product Quality R0140

1.1.1 Domains

The "Quality in Use" depends on the domain. Measures to improve software quality are often domain-specific.

Some domains are more focusing on tests, some on formal proves, some on reaction times till deploying software updates.



Aerospace C0051

Automotive C0052

Defense C0053

Avionics C0054

Industrial Machinery C0055

Medical C0056

Backend Server C0057

Finance C0105

Telecommunication C0106

Rail C0107

Energy C0110

Transport and Logistics C0111

Government C0112

Home Automation C0146

Entertainment, Sports C0145

Education C0147

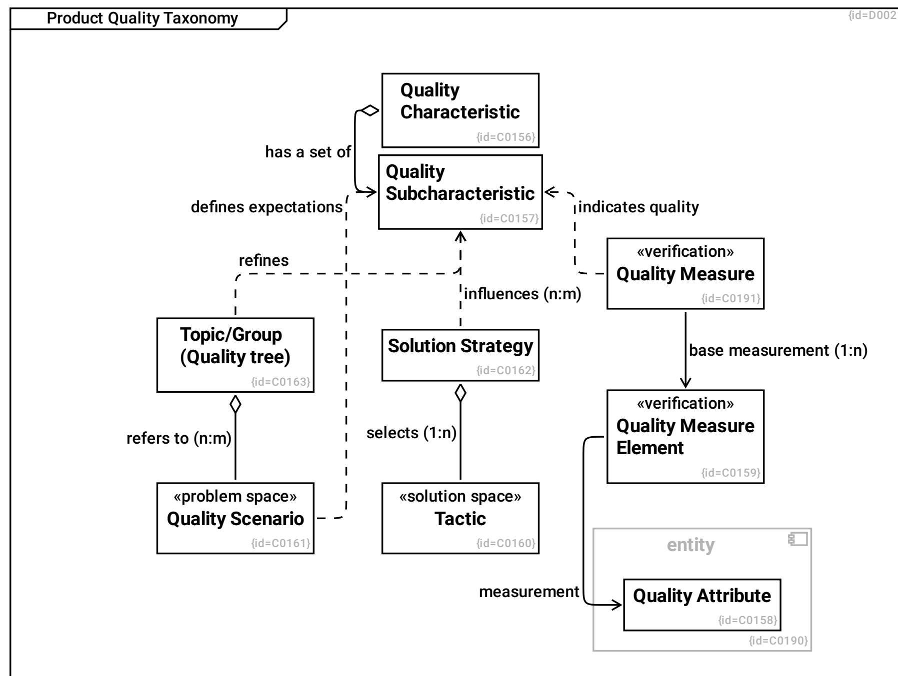
Media/Press/Publishing C0120

Travel / Tourism C0119

1.1.2 Product Quality Taxonomy

This diagram shows some terms in the context of product quality and their relationships. (Domain Model)

This picture is composed of ideas from 1) Software Architecture in Practice (C0165), 2) ISO/IEC 9126 (C0166), 3) arc42 (C0168), see Section 1.3: Bibliography.



Quality Measure C0191

A quality measure is a metric based on a set of observed values indicating the qualiity of an entity.

indicates quality --> Quality Subcharacteristic R0201

base measurement (1:n) --> Quality Measure Element R0202

Quality Subcharacteristic C0157

A subcharacterstic is a property of a system that expresses how well it fits to a quality expectation.

Topic/Group (Quality tree) C0163

A quality tree starts at the quality characteristics as the main branches and ends at detailed quality requirements as the leaves.

The detailed quality requirements then may refer to quality szenarios where these are of significance.

refers to (n:m) --> Quality Scenario R0162

refines --> Quality Subcharacteristic R0166

entity C0190

--> Quality Attribute R0199

Quality Scenario C0161

defines expectations --> Quality Subcharacteristic R0165

Solution Strategy C0162

selects (1:n) --> Tactic R0163

influences (n:m) --> Quality Subcharacteristic R0164

Quality Attribute C0158

An attribute is an internal property of an entity (e.g. software).

Quality Measure Element C0159

A measurement value. It can be objectively measured but may possibly need interpretation to state if the measured value is sutable to support a subcharacteristic.

measurement --> Quality Attribute R0200

Tactic C0160

A tactic is a method to influence a system to inherit a subcharacteristic.

See: Software Architecture in Practice (C0165) in Section 1.3: Bibliography.

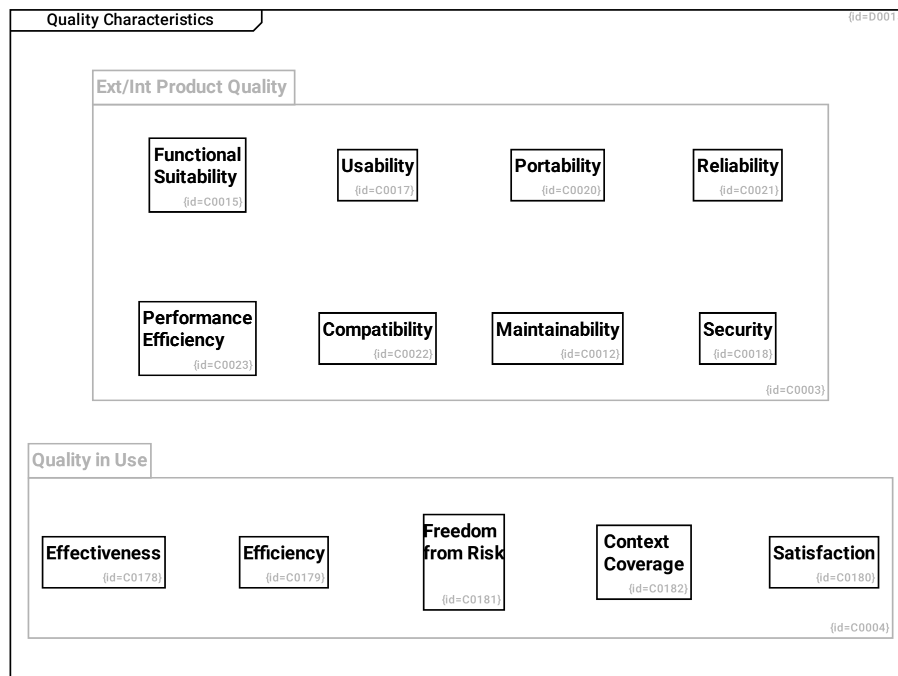
Quality Characteristic C0156

A characteristic is a set of subcharacteristics.

has a set of --> Quality Subcharacteristic R0158

1.1.3 Quality Characteristics

Charactersitics of Ext/Int Product Quality according to ISO/IEC 25010:2011 (C0167), see Section 1.3: Bibliography.



Functional Suitability C0015

Compatibility C0022

Reliability C0021

Maintainability C0012

Security C0018

Efficiency C0179

Context Coverage C0182

Effectiveness C0178

Freedom from Risk C0181

see Section 1.1.3.3.1: Strategies for Safety.

Satisfaction C0180

Portability C0020

Performance Efficiency C0023

Usability C0017

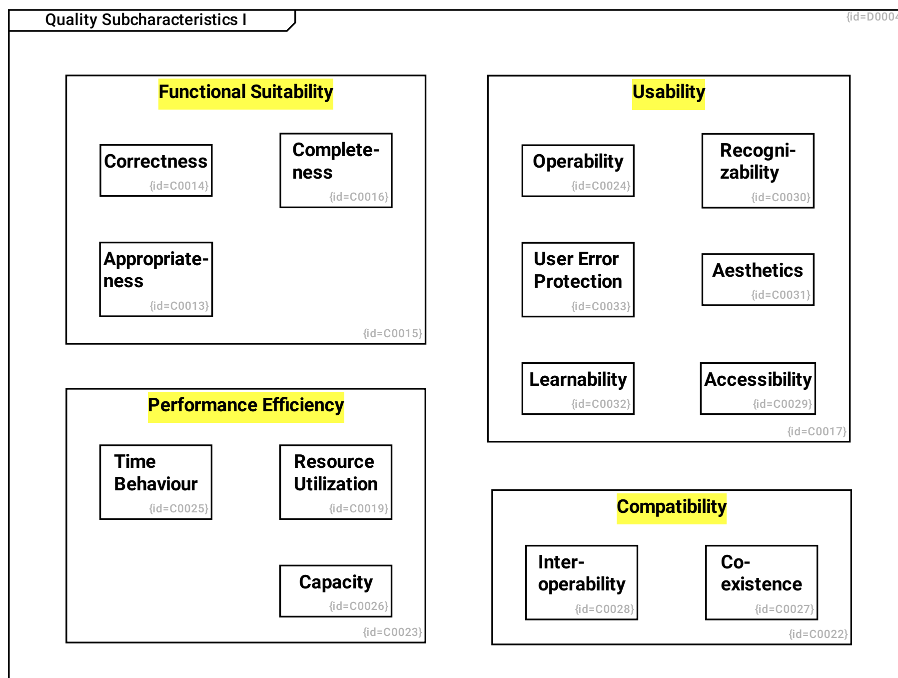
Ext/Int Product Quality C0003

Product quality are internal and externally visible qualities, such as memory consumption or startup timings.

Quality in Use C0004

Quality in use can be measured when the product is already in use, e.g. the percentage of satisfied customers can be determined.

1.1.3.1 Quality Subcharacteristics I



Correctness C0014

Functional Suitability C0015

--> Complete-ness R0056

--> Correctness R0057

--> Appropriate-ness R0058

Resource Utilization C0019

Compatibility C0022

--> Co-existence R0066

--> Inter-operability R0067

Operability C0024

Time Behaviour C0025

Co-existence C0027

Inter-operability C0028

Accessibility C0029

Recogni-zability C0030

Aesthetics C0031

Learnability C0032

Appropriate-ness C0013

Complete-ness C0016

Usability C0017

--> Recogni-zability R0071

--> Learnability R0072

--> Operability R0073

--> User Error Protection R0074

--> Aesthetics R0075

--> Accessibility R0076

Performance Efficiency C0023

--> Time Behaviour R0059

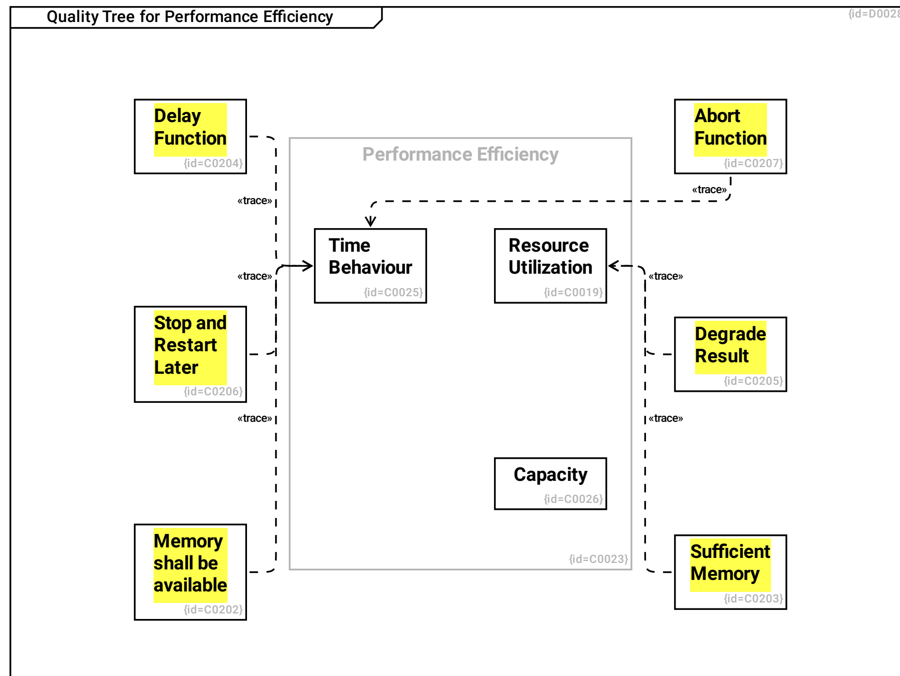
--> Resource Utilization R0060

--> Capacity R0061

Capacity C0026

User Error Protection C0033

1.1.3.1.1 Quality Tree for Performance Efficiency



Time Behaviour C0025

Resource Utilization C0019

Memory shall be available C0202

For dedicated use cases and functions, the software shall access required memory within 1 usec.

--> Time Behaviour R0237

Degrade Result C0205

For dedicated use cases and functions, the software shall calculate a result where accuracy may degrade if memory is sparse.

--> Resource Utilization R0234

Sufficient Memory C0203

For dedicated use cases and functions, ste software shall be able to access sufficient memory.

--> Resource Utilization R0233

Delay Function C0204

For dedicated use cases and functions, the software shall delay processing till enough memory is available.

--> Time Behaviour R0232

Stop and Restart Later C0206

In case of insufficient amount of memory, the software shall stop operating and be started again at a later point in time.

--> Time Behaviour R0236

Abort Function C0207

In case of insufficient memory, the software shall abort the function.

--> Time Behaviour R0238

Performance Efficiency C0023

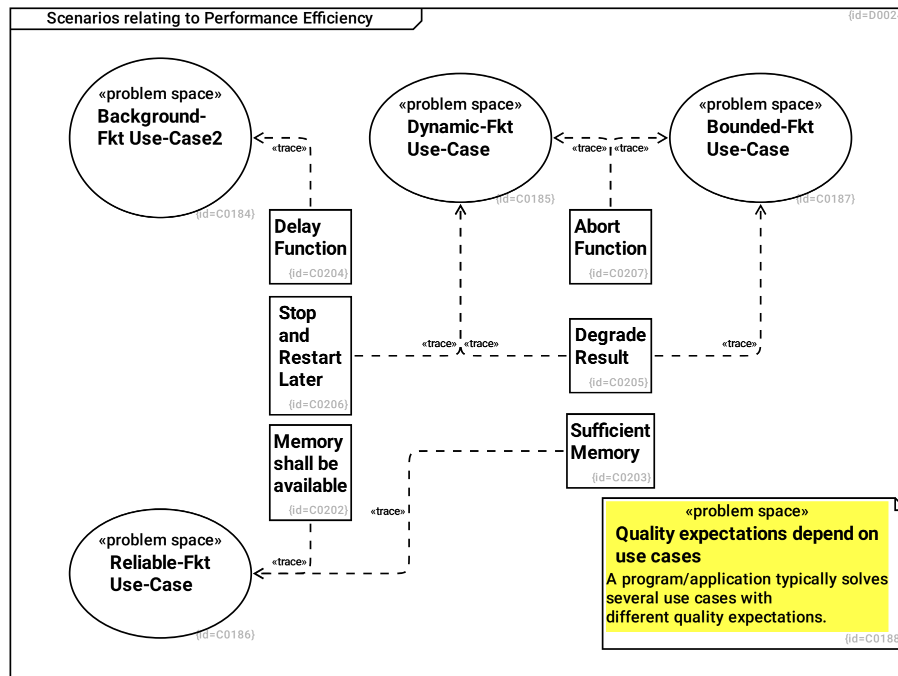
--> Time Behaviour R0059

--> Resource Utilization R0060

--> Capacity R0061

Capacity C0026

1.1.3.1.1.1 Scenarios relating to Performance Efficiency



Memory shall be available C0202

For dedicated use cases and functions, the software shall access required memory within 1 usec.

--> Reliable-Fkt Use-Case R0239

Degrade Result C0205

For dedicated use cases and functions, the software shall calculate a result where accuracy may degrade if memory is sparse.

--> Bounded-Fkt Use-Case R0242

--> Dynamic-Fkt Use-Case R0246

Sufficient Memory C0203

For dedicated use cases and functions, ste software shall be able to access sufficient memory.

--> Reliable-Fkt Use-Case R0240

Delay Function C0204

For dedicated use cases and functions, the software shall delay processing till enough memory is available.

--> Background-Fkt Use-Case2 R0243

Stop and Restart Later C0206

In case of insufficient amount of memory, the software shall stop operating and be started again at a later point in time.

--> Dynamic-Fkt Use-Case R0244

Quality expectations depend on use cases C0188

A program/application typically solves several use cases with different quality expectations.

Background-Fkt Use-Case2 C0184

This use case is a proxy for use cases that can be delayed if resources are sparse,

e.g. polling for software updates or performing data backups.

Dynamic-Fkt Use-Case C0185

This use case is a proxy for use cases that allow to produce degraded or no results if memory gets low.

E.g. A list of search results may be truncated, an undo-history may have limited size, the number of simultaneously open windows may be limited by the available memory.

Reliable-Fkt Use-Case C0186

This use case is a proxy for use cases that shall always work (available and reliable).

Abort Function C0207

In case of insufficient memory, the software shall abort the function.

--> Bounded-Fkt Use-Case R0241

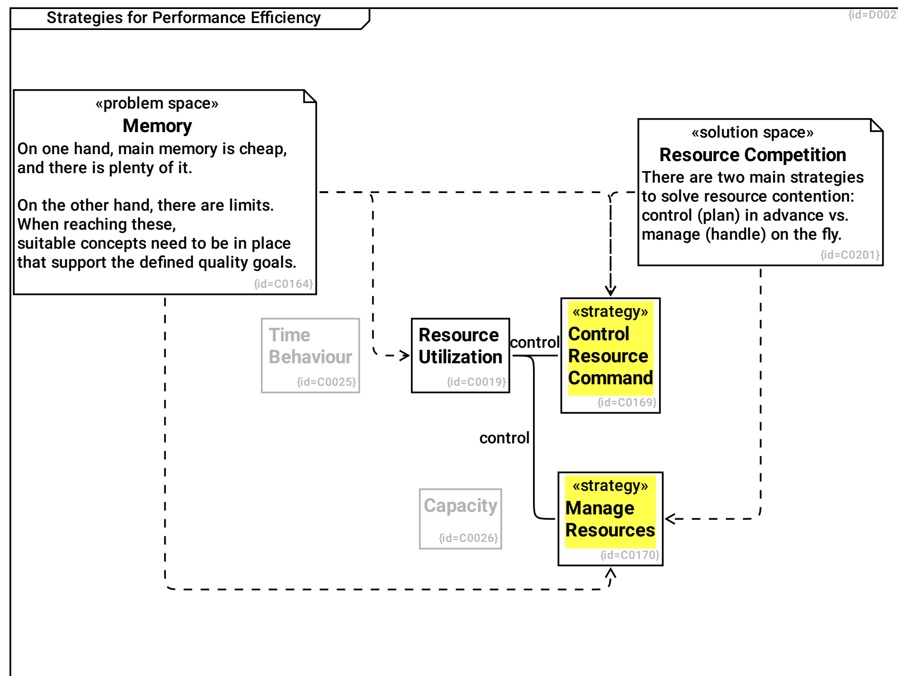
--> Dynamic-Fkt Use-Case R0245

Bounded-Fkt Use-Case C0187

This use case is a proxy for use cases that allow to produce degraded results if memory gets low or an upper boundary of resource-consumption is reached.

This is use-ful e.g. to enhance security when processing input data of unknown size or to prevent stealing resources from higher-priority dynamic functions.

1.1.3.1.2 Strategies for Performance Efficiency



Resource Utilization C0019

Time Behaviour C0025

Memory C0164

On one hand, main memory is cheap, and there is plenty of it.

On the other hand, there are limits. When reaching these, suitable concepts need to be in place that support the defined quality goals.

--> Resource Utilization R0167

--> Control Resource Command R0211

--> Manage Resources R0212

Resource Competition C0201

There are two main strategies to solve resource contention: control (plan) in advance vs. manage (handle) on the fly.

--> Control Resource Command R0230

--> Manage Resources R0231

Manage Resources C0170

For detailed tactics, see Section 1.1.3.1.2.2: Memory Management Patterns/Tactics.

Source: Software Architecture in Practice (C0165) in Section 1.3: Bibliography.

control --> Resource Utilization R0170

Control Resource Command C0169

For detailed tactics, see Section 1.1.3.1.2.1: Memory Controlling Patterns/Tactics.

Source: Software Architecture in Practice (C0165) in Section 1.3: Bibliography.

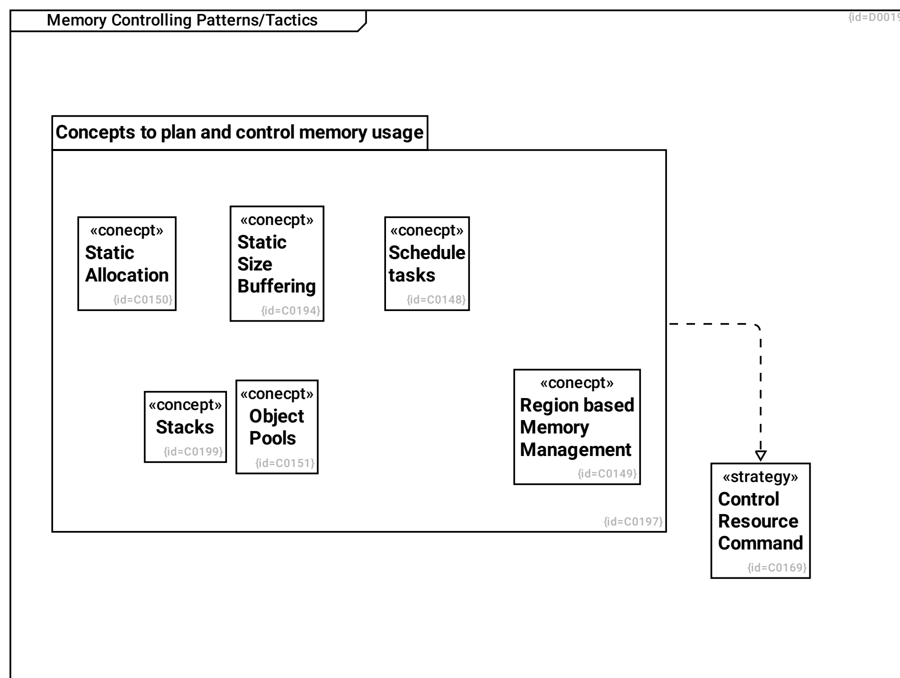
control --> Resource Utilization R0171

Capacity C0026

1.1.3.1.2.1 Memory Controlling Patterns/Tactics

This diagram shows patterns to control memory in a way that guarantees availability.

Concept is that software components have a guarantee for a minimal amount of memory that they need to operate.



Concepts to plan and control memory usage C0197

--> Stacks R0228

--> Static Allocation R0215

--> Static Size Buffering R0216

--> Schedule tasks R0217

--> Object Pools R0218

--> Region based Memory Management R0219

--> Control Resource Command R0227

Static Size Buffering C0194

Windowing Concept Streaming/Buffering

Stacks C0199

A stack deallocates memory only in the reverse order as it was allocated before.

Variant: monotonic_buffer_resource(C++17)

Control Resource Command C0169

For detailed tactics, see Section 1.1.3.1.2.1: Memory Controlling Patterns/Tactics.

Source: Software Architecture in Practice (C0165) in Section 1.3: Bibliography.

Region based Memory Management C0149

A region/arena/stack is valid while performing one operation/thread. It provides single-threaded access only. It is either used stack-like or cleaned up when the operation/thread is finished.

This concept allows to fast allocate and completely release memory arenas. It prevents fragementation within the stack. It supports NUMA architectures. It may be faster than concurrent access to single heap. It allows to reserve appropriate memory arenas for more important tasks

see https://en.wikipedia.org/wiki /Region-based_memory_management

Implementation support by e.g. polymorphic_allocator and memory_resource(C++17/20) rust allocator: https://rust-lang.github.io/rfcs/1398-kinds-of-allocators.html

Schedule tasks C0148

- Schedule tasks that require memory to prevent contention

E.g. only one memory-intensive task may run at a time

Object Pools C0151

Pools of same-type objects are pre-allocated.

This concept guarantees a predefined amount of memory and prevents fragmentation

Examples: synchronized_pool_resource(C++17) unsynchronized_pool_resource(C++17)

Static Allocation C0150

All memory is statically allocated except for stacks. The number of threads is static.

Example implementations:

- classic AUTOSAR, MISRA-C, MISRA-C++,

- Ada-Spark,

- NASA Power-of-Ten

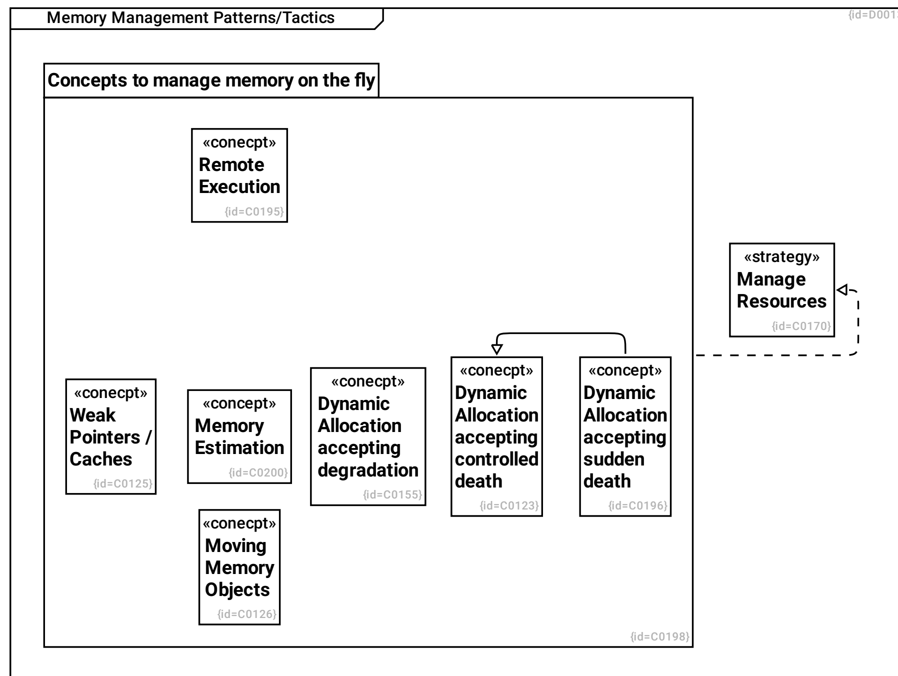
- crystal_facet_uml (except gtk3+sqlite3)

- GPSd deamon (see the_architecture_of _open_source_applications__volume_ii.pdf)

1.1.3.1.2.2 Memory Management Patterns/Tactics

This diagram shows patterns that manage memory in a way that whoever needs memory gets unused memory if available.

Concept is, that every software component only allocates memory that it really needs; if no memory is left, something will fail anyhow.



Concepts to manage memory on the fly C0198

--> Memory Estimation R0229

--> Weak Pointers / Caches R0220

--> Dynamic Allocation accepting degradation R0221

--> Remote Execution R0222

--> Moving Memory Objects R0223

--> Dynamic Allocation accepting controlled death R0224

--> Dynamic Allocation accepting sudden death R0225

--> Manage Resources R0226

Remote Execution C0195

Required memory may be provided on a different hardware, e.g. Cloud

Memory Estimation C0200

Estimate required memory, expected fragmentation overhead and additional buffer to prevent out-of-memory situations.

Weak Pointers / Caches C0125

- There are managed data structures (hashmaps, caches) that hold data as long as much memory is available and that drop data when memory becomes sparse

Example implementations: Java Weak Hashmap, sqlite, OS file system caches

Challenge: The cache needs to know when memory is needed elsewhere.

Manage Resources C0170

For detailed tactics, see Section 1.1.3.1.2.2: Memory Management Patterns/Tactics.

Source: Software Architecture in Practice (C0165) in Section 1.3: Bibliography.

Dynamic Allocation accepting sudden death C0196

- Linux overcommit

- Compress memory pages

- End any process if page-fault by r/w operation to overcommitted memory

Example Implementations: Android App Management (see low memory killer daemon: lmkd) Linux OOM-Killer

--> Dynamic Allocation accepting controlled death R0214

Moving Memory Objects C0126

Some instance manages memory (e.g. OS) while another instance uses it (e.g. App).

This mamagement may reduce/fix fragmentation issues.

Examples:

- The java virtual machine can move objects while a java application runs

- The classic MacOS7-9 could move memory while the application holds a handle (pointer to a pointer)

- Todays processors convert virtual to physical addresses on the fly (reallocations via e.g. sbrk, mmap)

Dynamic Allocation accepting controlled death C0123

- Accept memory fragmentation (physical pages as well as virtual addresses)

- Swap memory pages to flash

- End a process if no memory available

Dynamic Allocation accepting degradation C0155

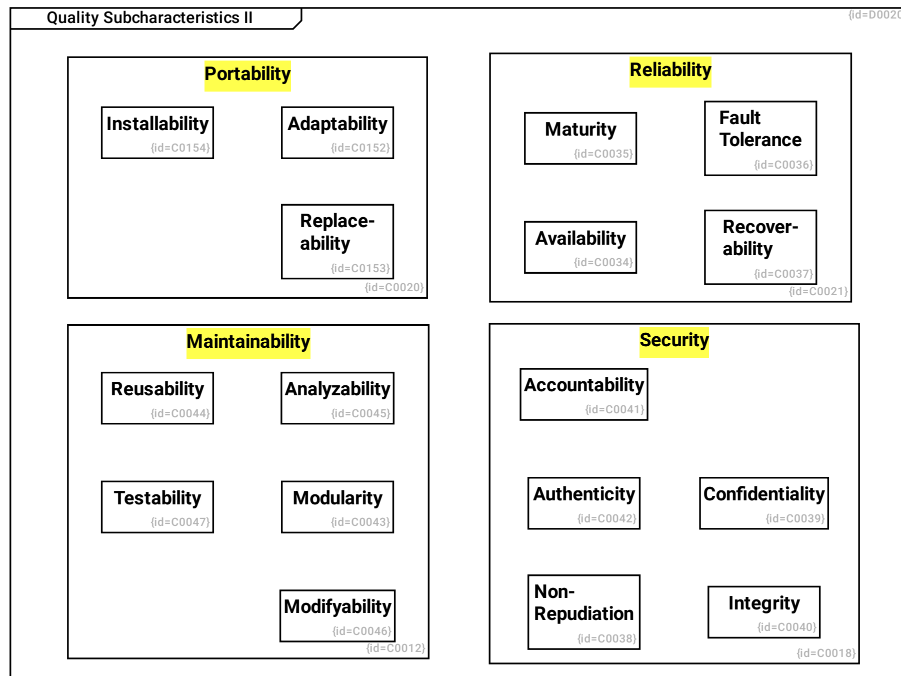
Continue working, accept that some requested functionality

- cannot be performed at current point in time

- with expected precision

- within expected timeframe

1.1.3.2 Quality Subcharacteristics II



Reliability C0021

--> Maturity R0062

--> Availability R0063

--> Fault Tolerance R0064

--> Recover-ability R0065

Fault Tolerance C0036

Maturity C0035

Recover-ability C0037

Security C0018

--> Authenticity R0082

--> Non-Repudiation R0083

--> Accountability R0084

--> Integrity R0085

--> Confidentiality R0086

Confidentiality C0039

Integrity C0040

Authenticity C0042

Non-Repudiation C0038

Maintainability C0012

--> Testability R0077

--> Modifyability R0078

--> Analyzability R0079

--> Reusability R0080

--> Modularity R0081

Analyzability C0045

Reusability C0044

Testability C0047

Modularity C0043

Availability C0034

Portability C0020

--> Adaptability R0155

--> Replace-ability R0156

--> Installability R0157

Adaptability C0152

Replace-ability C0153

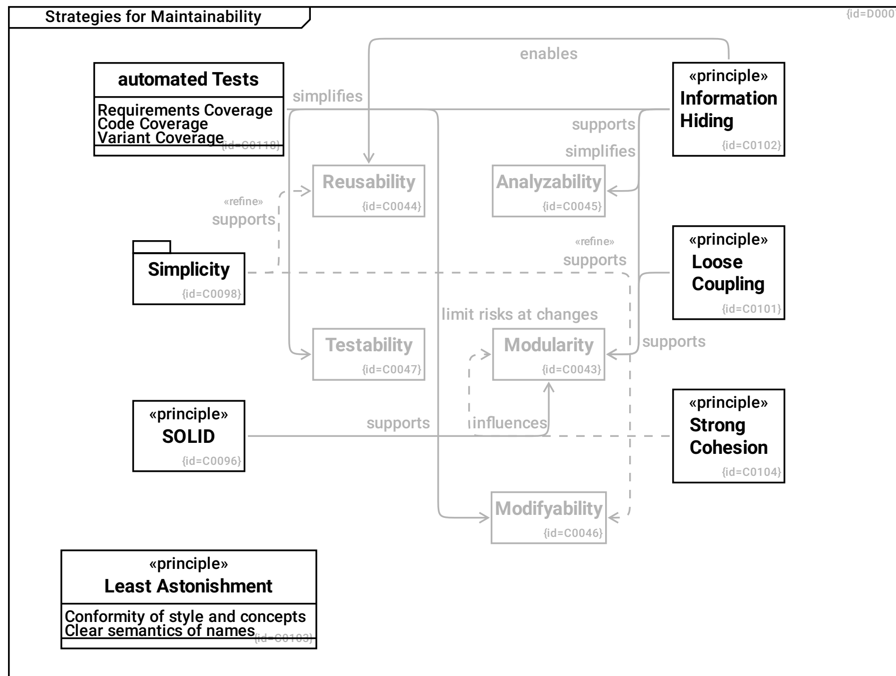
Installability C0154

Accountability C0041

Modifyability C0046

1.1.3.2.1 Strategies for Maintainability

This diagram shows examples - not aiming for completeness.



Testability C0047

Analyzability C0045

Reusability C0044

Modularity C0043

Loose Coupling C0101
split an entity that consists of multiple loosely coupled parts

supports --> Modularity R0114

Information Hiding C0102
A sofware component shall hide its implementation details and make information accessible only via defined interfaces

enables --> Reusability R0115

supports --> Modularity R0116

simplifies --> Testability R0117

simplifies --> Analyzability R0118

Strong Cohesion C0104

influences --> Modularity R0119

automated Tests C0118

Requirements Coverage F0069

Code Coverage F0068

Variant Coverage F0080

HW variants, Feature Variants, Compiler/OS Variants

limit risks at changes --> Modifyability R0135

Modifyability C0046

SOLID C0096

supports --> Modularity R0111

Simplicity C0098

supports --> Modifyability R0109

supports --> Reusability R0110

Least Astonishment C0103

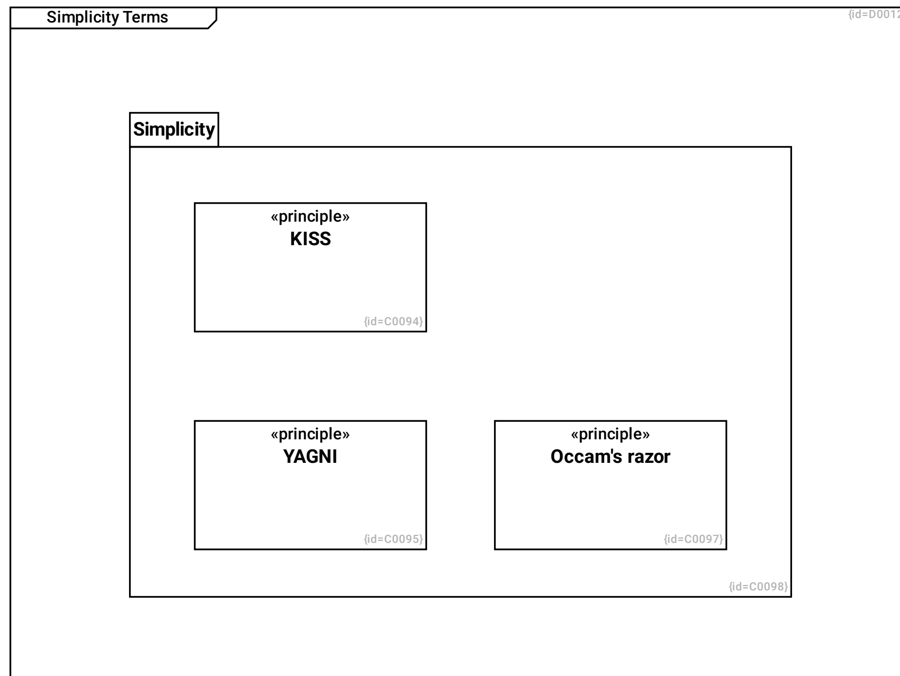
A reader shall not be surprised when looking at the design.

Conformity of style and concepts F0066

Clear semantics of names F0067

Module names, Interface names, Message names, Port names: The name shall state what the data/function represents. The name shall be short and as concrete as possible.

1.1.3.2.1.1 Simplicity Terms



KISS C0094
Keep it simple and stupid

Occam's razor C0097
Among competing hypotheses, the one with the fewest assumptions should be selected

YAGNI C0095
You aren't gonna need it

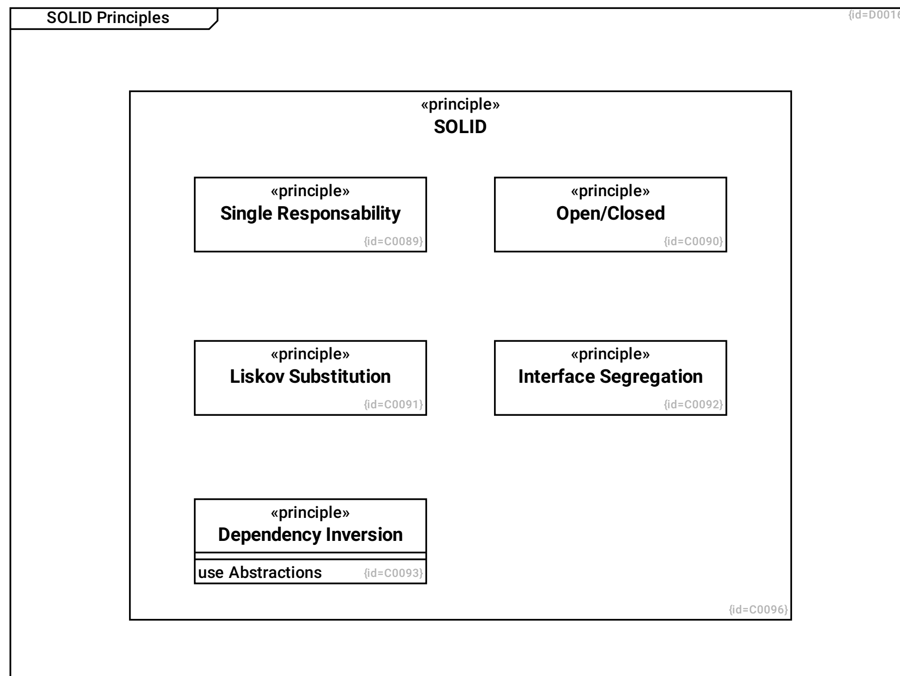
Simplicity C0098

--> KISS R0106

--> YAGNI R0107

--> Occam's razor R0108

1.1.3.2.1.2 SOLID Principles



Single Responsability C0089

A software component shall be responsible for one topic only

Open/Closed C0090

Open for extension, closed for modification

Interface Segregation C0092

Avoid general purpose interfaces, design multiple interfaces specific to the needs of different users/clients

Dependency Inversion C0093

A software component shall depend on abstractions, not on concrete implementations

use Abstractions F0046

SOLID C0096

--> Interface Segregation R0101

--> Liskov Substitution R0102

--> Dependency Inversion R0103

--> Open/Closed R0104

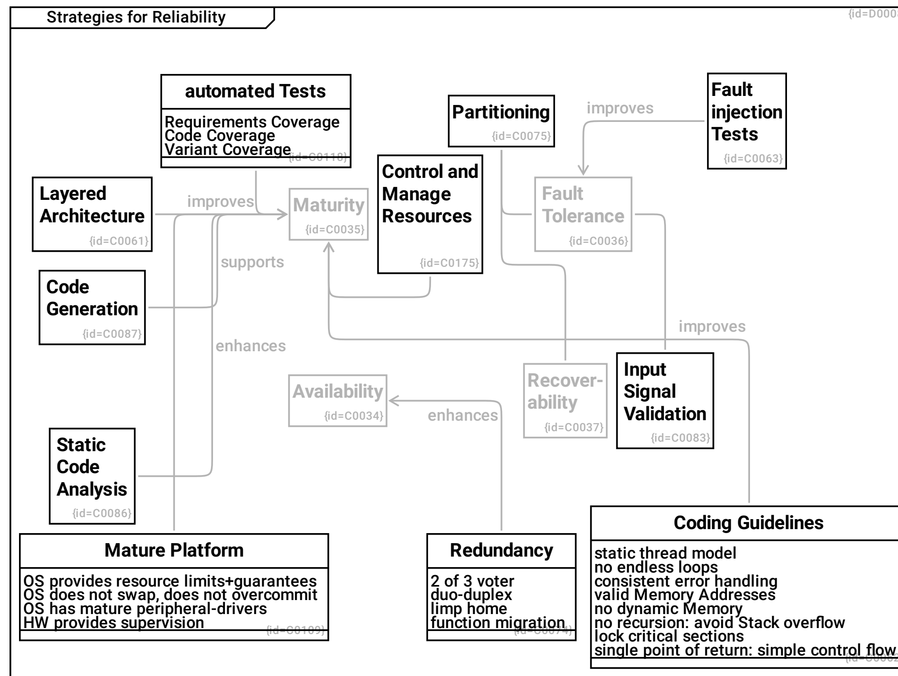
--> Single Responsability R0105

Liskov Substitution C0091

An implementation of an interface can be replaced by another implementation of the same interface. In object oriented design, types can be replaced by subtypes.

1.1.3.2.2 Strategies for Reliability

This diagram shows examples - not aiming for completeness.



Recover-ability C0037

Fault Tolerance C0036

Maturity C0035

Layered Architecture C0061

improves --> Maturity R0039

Coding Guidelines C0062

Coding guidelines define how to get reproducible behavior of software. Managing system resources is a key factor.

static thread model F0010

Execution threads shall not be started/stopped dynamically

no endless loops F0008

Every loop shall have a counter to ensures that after a predefined maximum value the loop is definitely quit

consistent error handling F0009

Inconsistencies in error handling make bugs in error handling more likely

valid Memory Addresses F0007

Only valid memory addresses may be read/written.

- Java solves this by prohibiting pointers,

- In C, check pointers and array indices before usage,

- In C++, use std::shared_ptr and std::vector,

- Kotlin even distinguishes between nullable references and non-null references.

no dynamic Memory F0006

When the program is running,

- it must not fail due to - memory fragmentation (virtual addresses/physical pages) - out of memory situations

- it shall have a defined timing (which new/malloc cannot provide)

no recursion: avoid Stack overflow F0005

lock critical sections F0024

Always lock critical sections.

single point of return: simple control flow F0023

Simple control flow is key to understandable code

improves --> Maturity R0040

Fault injection Tests C0063

improves --> Fault Tolerance R0041

Redundancy C0074

2 of 3 voter F0025

duo-duplex F0026

duo-duplex is a concept that consists of 4 independant parts: Two functions (F1 and F2) and two monitors (M1 and M2).

- M1 supervises F1. In case F1 fails, M1 ensures that F1 has no effect.

- M2 supervises F2. In case F2 fails, M2 ensures that F2 has no effect.

As long as (F1/M1) produces output, (F2/M2) is ignored.

limp home F0027

function migration F0028

enhances --> Availability R0055

Static Code Analysis C0086

enhances --> Maturity R0099

Code Generation C0087

An understandable model and a small code generator allow to generate mature software.

supports --> Maturity R0100

Mature Platform C0109

OS provides resource limits+guarantees F0061

OS does not swap, does not overcommit F0062

OS has mature peripheral-drivers F0063

HW provides supervision F0070

E.g.

- timer trigger OS-interrupts

- access violations trigger OS-interrupts

--> Maturity R0124

Input Signal Validation C0083

Precondition is a specification of valid input signals.

This can be implemented by different means, e.g.

- assert() statements as declared in assert.h

- if(COND) {...} else {...} statements where the else-path logs the error

For Security: Check input signals already at the gate, not later.

--> Fault Tolerance R0128

Partitioning C0075

synonym for Compartments

see Section 1.1.3.2.3.1: Tactics for Partitioning.

--> Fault Tolerance R0129

--> Recover-ability R0130

Control and Manage Resources C0175

see Section 1.1.3.1.1.1: Scenarios relating to Performance Efficiency.

--> Maturity R0173

automated Tests C0118

Requirements Coverage F0069

Code Coverage F0068

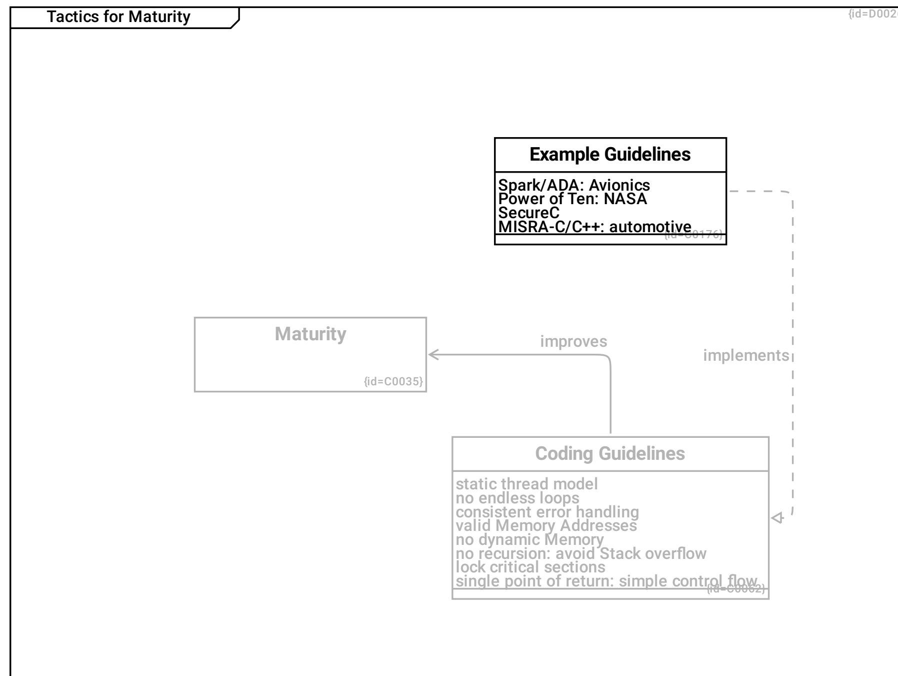
Variant Coverage F0080

HW variants, Feature Variants, Compiler/OS Variants

--> Maturity R0136

Availability C0034

1.1.3.2.2.1 Tactics for Maturity



Maturity C0035

Coding Guidelines C0062

Coding guidelines define how to get reproducible behavior of software. Managing system resources is a key factor.

static thread model F0010

Execution threads shall not be started/stopped dynamically

no endless loops F0008

Every loop shall have a counter to ensures that after a predefined maximum value the loop is definitely quit

consistent error handling F0009

Inconsistencies in error handling make bugs in error handling more likely

valid Memory Addresses F0007

Only valid memory addresses may be read/written.

- Java solves this by prohibiting pointers,

- In C, check pointers and array indices before usage,

- In C++, use std::shared_ptr and std::vector,

- Kotlin even distinguishes between nullable references and non-null references.

no dynamic Memory F0006

When the program is running,

- it must not fail due to - memory fragmentation (virtual addresses/physical pages) - out of memory situations

- it shall have a defined timing (which new/malloc cannot provide)

no recursion: avoid Stack overflow F0005

lock critical sections F0024

Always lock critical sections.

single point of return: simple control flow F0023

Simple control flow is key to understandable code

improves --> Maturity R0040

Example Guidelines C0176

Spark/ADA: Avionics F0076

see C0173#name

Power of Ten: NASA F0077

SecureC F0078

MISRA-C/C++: automotive F0079

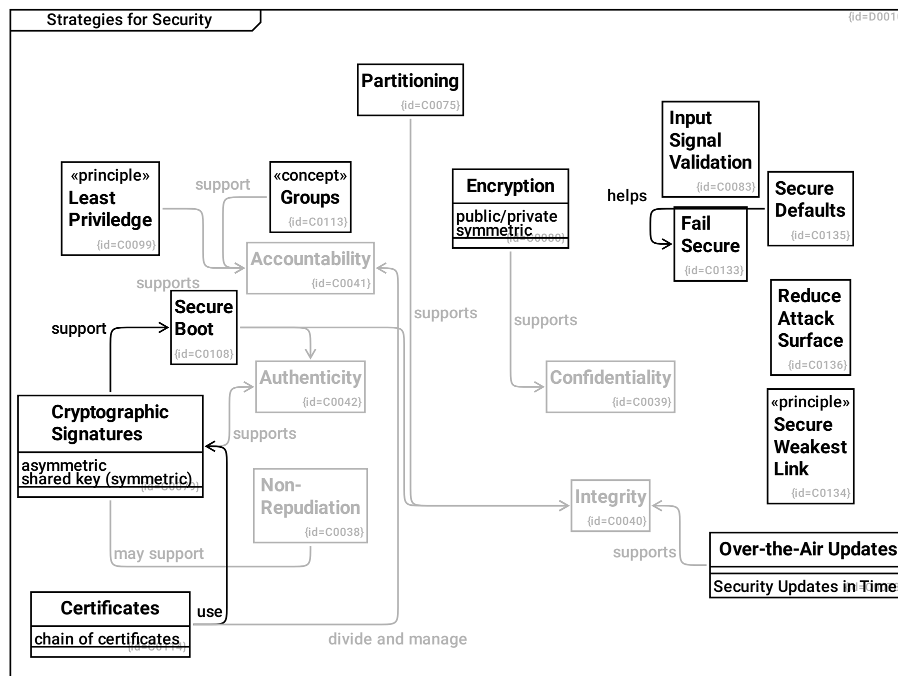
implements --> Coding Guidelines R0174

1.1.3.2.3 Strategies for Security

Functional safety and security are different goals but have common mechanisms to support these.

The diagram is not meant to be complete, it just shows some technical mechanisms support quality goals.

Especially for safety and security, selecting the right set of the appropriate mechanisms is crucial.



Confidentiality C0039

Integrity C0040

Authenticity C0042

Partitioning C0075

synonym for Compartments

see Section 1.1.3.2.3.1: Tactics for Partitioning.

supports --> Integrity R0089

Over-the-Air Updates C0078

Security Updates in Time F0035

supports --> Integrity R0090

Cryptographic Signatures C0079

asymmetric F0038

shared key (symmetric) F0039

supports --> Authenticity R0091

may support --> Non-Repudiation R0120

support --> Secure Boot R0123

Encryption C0080

public/private F0037

symmetric F0036

supports --> Confidentiality R0092

Least Priviledge C0099

Entities shall have only the access rights they need for their purpose

supports --> Accountability R0112

Non-Repudiation C0038

Secure Boot C0108

--> Integrity R0121

--> Authenticity R0122

Groups C0113

Grouping Clients/Actors/Users and grouping Services helps in administration of access rights

support --> Accountability R0125

Certificates C0114

chain of certificates F0064

use --> Cryptographic Signatures R0126

divide and manage --> Accountability R0127

Input Signal Validation C0083

Precondition is a specification of valid input signals.
This can be implemented by different means, e.g.
- assert() statements as declared in assert.h
- if(COND) {...} else {...} statements where the else-path logs the error

For Security: Check input signals already at the gate, not later.

Reduce Attack Surface C0136

Remove unused functions

Secure Defaults C0135

The default settings/configuratoin of a system shall be tuned for security.

helps --> Fail Secure R0146

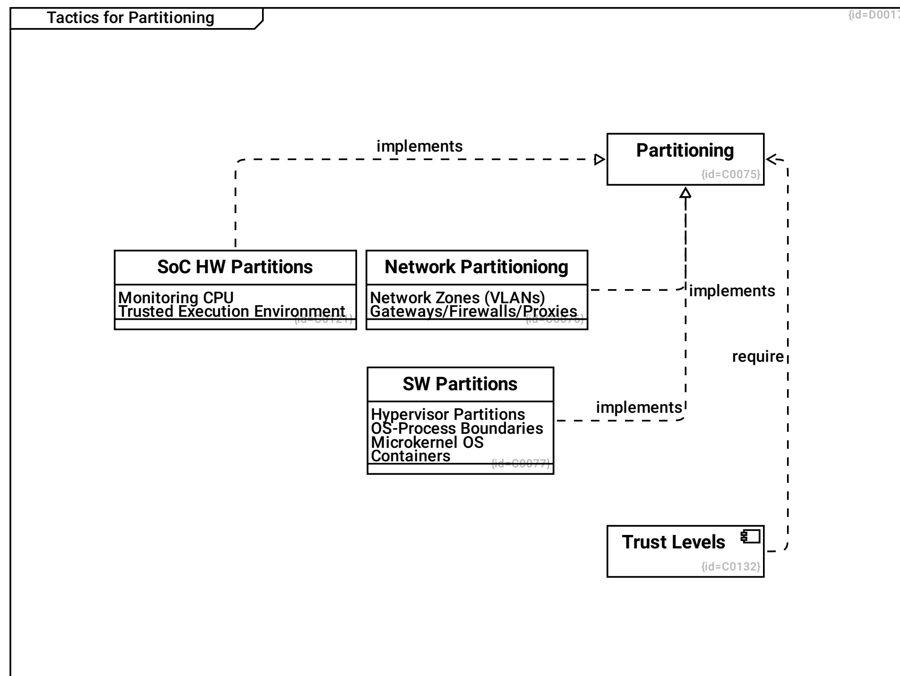
Secure Weakest Link C0134

Fail Secure C0133

Do not expose data or system details when a fault occurs.

Accountability C0041

1.1.3.2.3.1 Tactics for Partitioning



Partitioning C0075

synonym for Compartments

see Section 1.1.3.2.3.1: Tactics for Partitioning.

Network Partitioniong C0076

Network Zones (VLANs) F0029

Gateways/Firewalls/Proxies F0030

implements --> Partitioning R0087

SW Partitions C0077

Hypervisor Partitions F0031

OS-Process Boundaries F0032

Microkernel OS F0033

for Device Driver Partitioning

Containers F0034

Linux-Containers, QNX-Partitions, Address-space separation

implements --> Partitioning R0088

SoC HW Partitions C0121

The HW provides e.g.

- a trusted execution environment for cryptographic computations

- a separated monitoring CPU to supervise the functional CPUs

Monitoring CPU F0071

Trusted Execution Environment F0072

implements --> Partitioning R0141

Trust Levels C0132

Create a Multi-Layer security, where elements of one layer mistrust these of other layers. If one layer is compromized, the next still holds up.

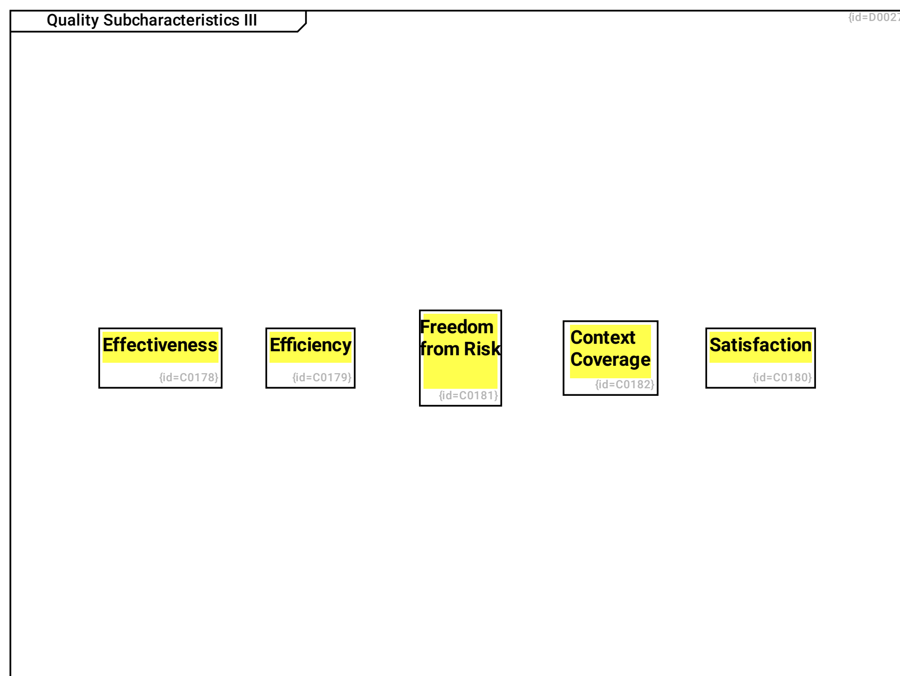
Examples:

1) Internet - DMZ - Intranet

2) OS-Process (user), root-acess, OS(kernel-space), Hypervisor, Trusted Execution Environment of SoC

require --> Partitioning R0144

1.1.3.3 Quality Subcharacteristics III



Efficiency C0179

Context Coverage C0182

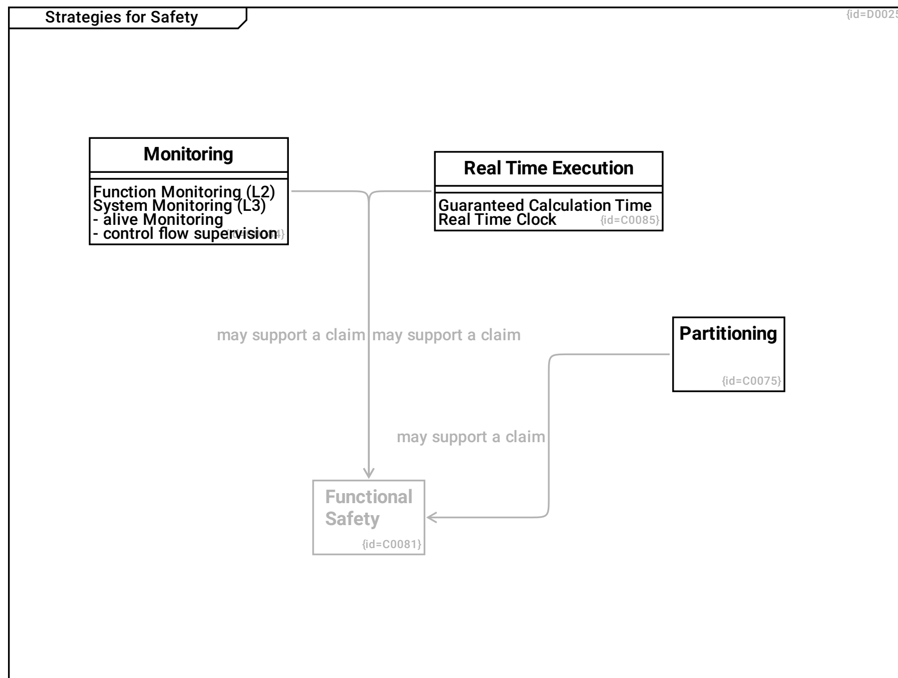
Effectiveness C0178

Freedom from Risk C0181

see Section 1.1.3.3.1: Strategies for Safety.

Satisfaction C0180

1.1.3.3.1 Strategies for Safety



Partitioning C0075

synonym for Compartments

see Section 1.1.3.2.3.1: Tactics for Partitioning.

may support a claim --> Functional Safety R0096

Functional Safety C0081

Monitoring C0084

Function Monitoring (L2) F0040

System Monitoring (L3) F0041

- alive Monitoring F0059

- control flow supervision F0060

may support a claim --> Functional Safety R0093

Real Time Execution C0085

Guaranteed Calculation Time F0044

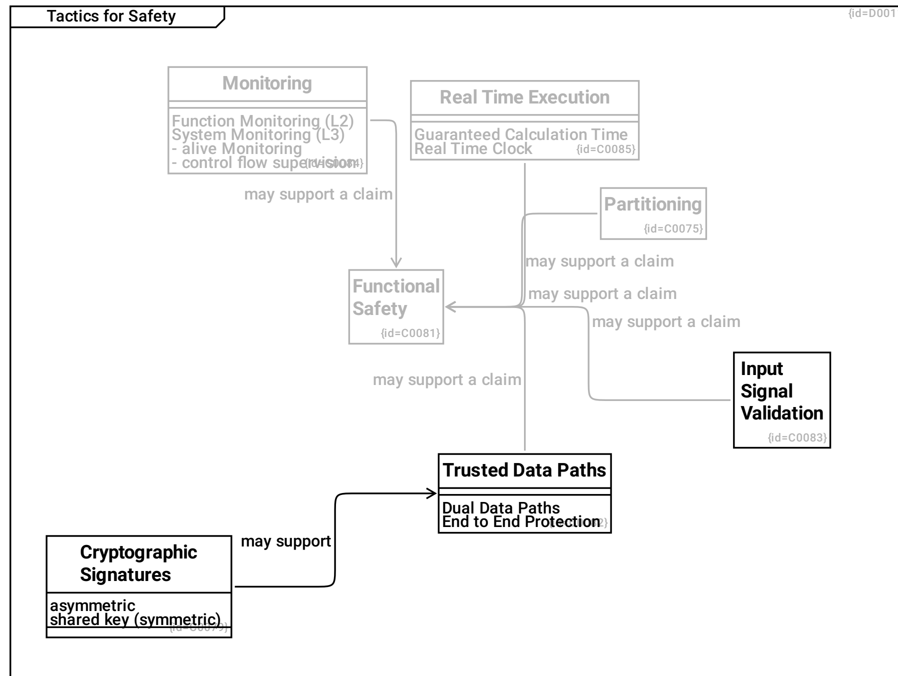
Real Time Clock F0045

may support a claim --> Functional Safety R0098

1.1.3.3.1.1 Tactics for Safety

This diagram shows examples for measures - not aiming for completeness.

Especially for safety and security, selecting the right set of the appropriate mechanisms is crucial.



Functional Safety C0081

Monitoring C0084

Function Monitoring (L2) F0040

System Monitoring (L3) F0041

- alive Monitoring F0059

- control flow supervision F0060

may support a claim --> Functional Safety R0093

Input Signal Validation C0083

Precondition is a specification of valid input signals.

This can be implemented by different means, e.g.

- assert() statements as declared in assert.h

- if(COND) {...} else {...} statements where the else-path logs the error

For Security: Check input signals already at the gate, not later.

may support a claim --> Functional Safety R0095

Real Time Execution C0085

Guaranteed Calculation Time F0044

Real Time Clock F0045

may support a claim --> Functional Safety R0098

Trusted Data Paths C0082

Dual Data Paths F0042

End to End Protection F0043

may support a claim --> Functional Safety R0094

Cryptographic Signatures C0079

asymmetric F0038

shared key (symmetric) F0039

may support --> Trusted Data Paths R0097

Partitioning C0075

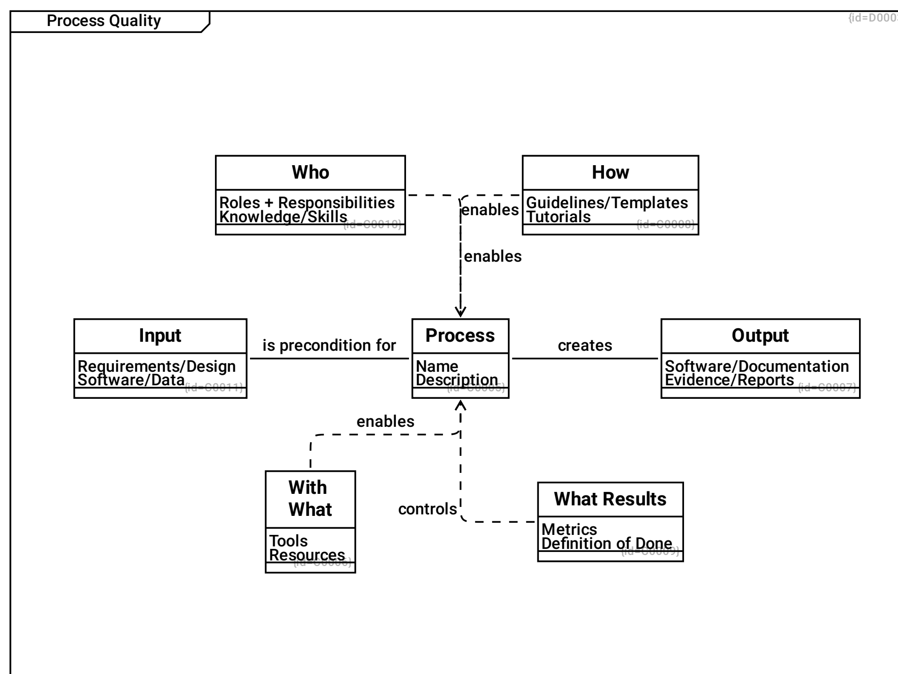
synonym for Compartments

see Section 1.1.3.2.3.1: Tactics for Partitioning.

may support a claim --> Functional Safety R0096

1.2 Process Quality

The turtle diagram shows the elements of a process.



Process C0005

Name F0011

Description F0012

creates --> Output R0002

With What C0006

Tools F0050

Resources F0051

enables --> Process R0003

How C0008
Guidelines, Checklists, Templates

Guidelines/Templates F0052

Tutorials F0065

enables --> Process R0005

What Results C0009

Metrics F0053

Definition of Done F0054

controls --> Process R0006

Who C0010
Roles, Skills, Knowledge, Trainings

Roles + Responsibilities F0048

Knowledge/Skills F0049

enables --> Process R0004

Input C0011

Requirements/Design F0057

Software/Data F0058

is precondition for --> Process R0001

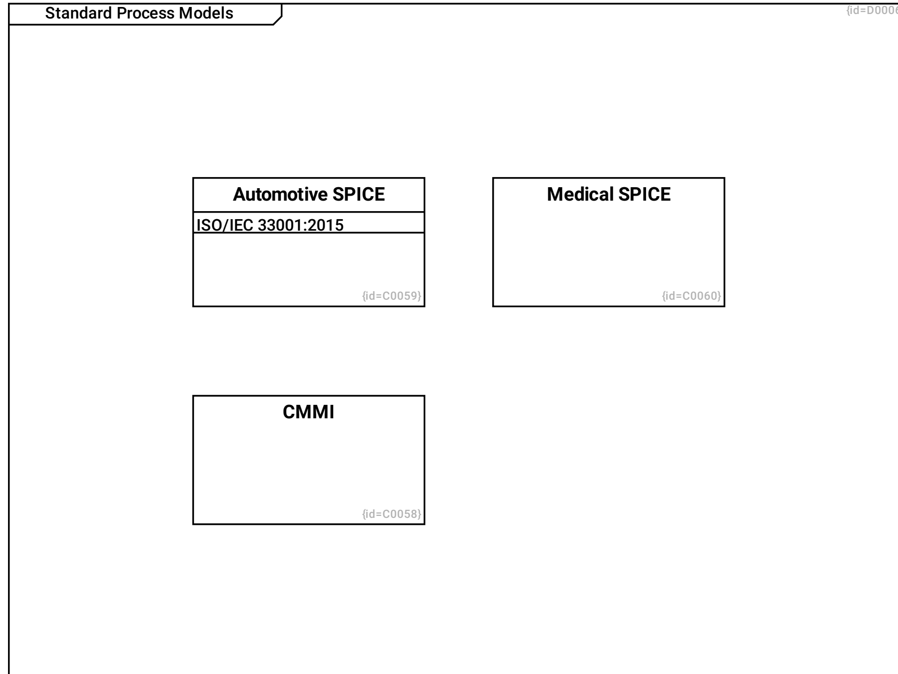
Output C0007
Process output, Evidence on performed process

Software/Documentation F0055

Evidence/Reports F0056

1.2.1 Standard Process Models

Process Models that focus on Software Development



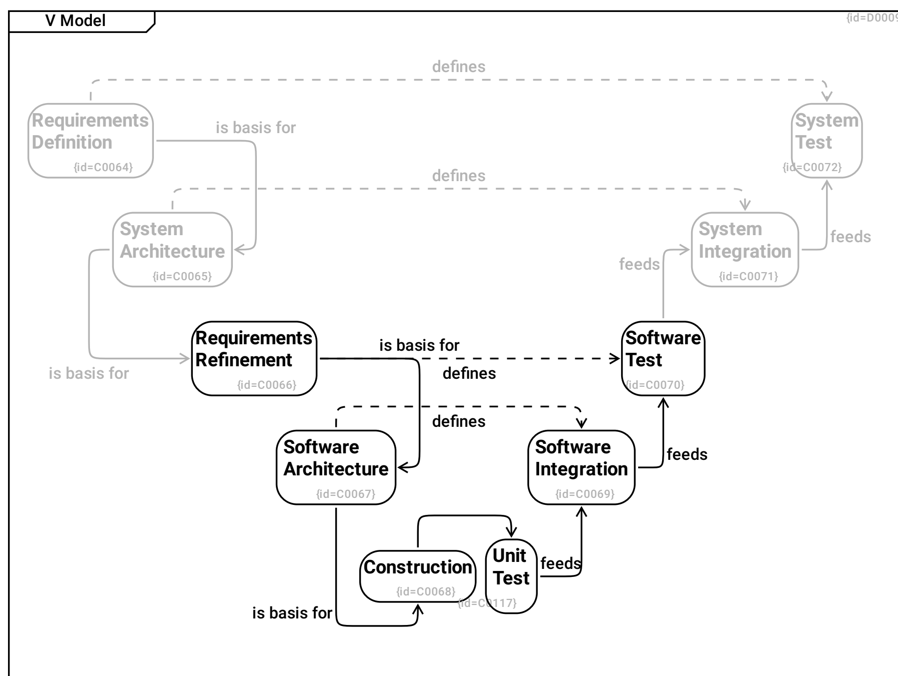
CMMI C0058

Automotive SPICE C0059

ISO/IEC 33001:2015 F0003

Medical SPICE C0060

1.2.2 V Model



Requirements Definition C0064

System Requirements Definition consists of
- elicitating requirements
- baselining requirements
- analyzing requirements

is basis for --> System Architecture R0042

defines --> System Test R0050

System Architecture C0065

is basis for --> Requirements Refinement R0043

defines --> System Integration R0051

Construction C0068

This process group consists of
- detailed design
- creating source code (implementation)
- integrating 3rd party software

--> Unit Test R0133

Software Integration C0069

Software integration consists of
- integration (building, linking, packaging, ...)
- integration test checks compliance to software architecture

feeds --> Software Test R0047

Software Test C0070

feeds --> System Integration R0048

System Integration C0071

System integration consists of
- installing software and configuration to devices
- managing versions (upgrade/downgrade/cross-variant-grade)
- integration test checks compliance to system architecture

feeds --> System Test R0049

System Test C0072

Requirements Refinement C0066

Software Requirements Refinement
- refines system requirements so that only the parts that shall be implemented in software are addressed
- states requirements in a form that these are testable by software/qualification tests
- distinguishes functional requirements
- quality/non-functional requirements
- non-requirements (what must not happen)
- process requirements

is basis for --> Software Architecture R0044

defines --> Software Test R0052

Software Architecture C0067

Template: arc42, see Section 1.3: Bibliography.
Method to check quality: ATAM, see C0165.

defines --> Software Integration R0053

The Software Architecture defines the modules, interfaces and relations needed to integrate and test the system.

is basis for --> Construction R0045

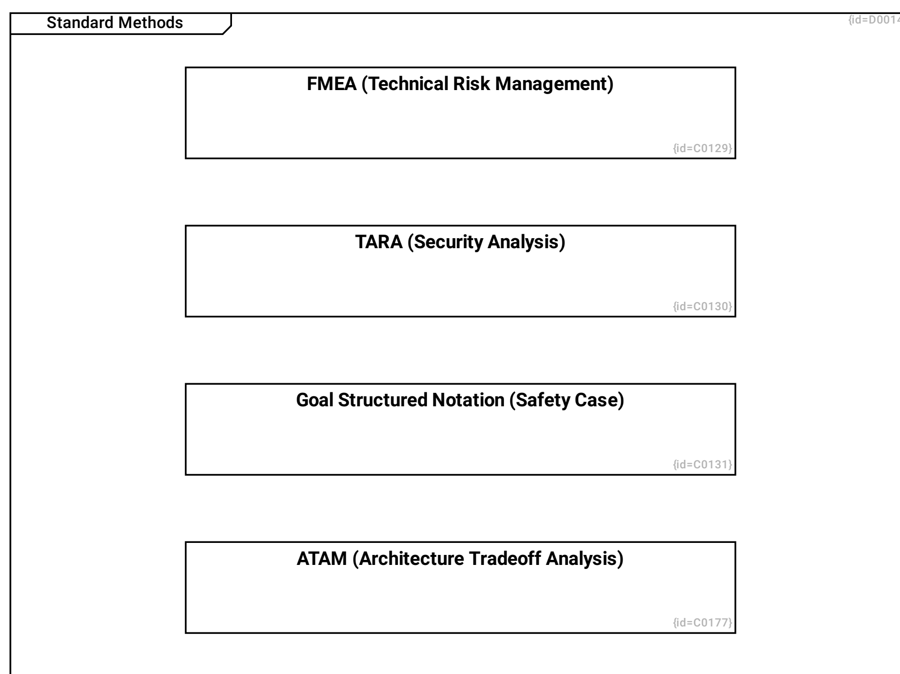
The Software Architecture defines the modules, interfaces and relations needed to create the system parts.

Unit Test C0117

feeds --> Software Integration R0134

1.2.3 Standard Methods

This diagram lists methods to ensure quality.



FMEA (Technical Risk Management) C0129

Failure Mode and Effects Analysis

see https://en.wikipedia.org/wiki/Failure_mode_and_effects_analysis

TARA (Security Analysis) C0130

Threat and Risk Analysis

Goal Structured Notation (Safety Case) C0131

Goal Structured Notation

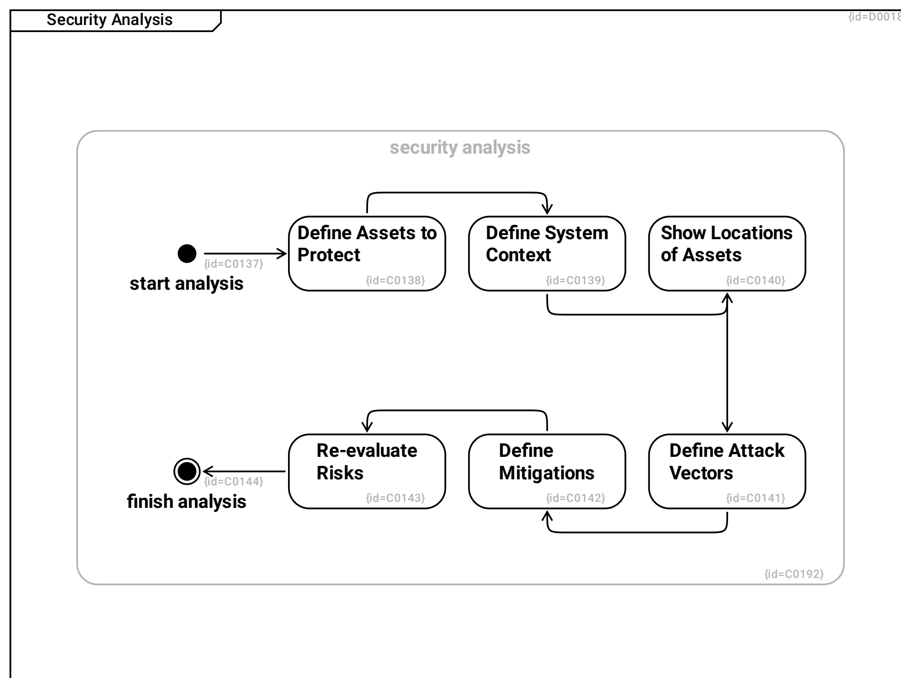
ATAM (Architecture Tradeoff Analysis) C0177

Architecture Tradeoff Analysis

see C0165 at Section 1.3: Bibliography.

1.2.3.1 Security Analysis

see Internet Security (C0172) in Section 1.3: Bibliography.



security analysis C0192

--> start analysis R0203

--> finish analysis R0204

--> Define Assets to Protect R0205

--> Re-evaluate Risks R0206

--> Define System Context R0207

--> Define Mitigations R0208

--> Show Locations of Assets R0209

--> Define Attack Vectors R0210

Define Assets to Protect C0138

This step
- defines which assets to protect
- determines risks for the case these are spied, modified, etc.

--> Define System Context R0151

Define System Context C0139

--> Show Locations of Assets R0152

Show Locations of Assets C0140

Show the components where assets are located and network lines where the assets pass along.

--> Define Attack Vectors R0149

Define Mitigations C0142

--> Re-evaluate Risks R0153

Define Attack Vectors C0141

--> Define Mitigations R0150

finish analysis C0144

Re-evaluate Risks C0143

--> finish analysis R0154

start analysis C0137

--> Define Assets to Protect R0147

1.3 Bibliography

Bibliography {id=D0023}
Engineering a Safer World {id=C0174}
Internet Security {id=C0172}
High Integrity Software {id=C0173}
Software Architecture in Practice {id=C0165}
Software Estimation {id=C0171}
Software Quality Engineering {id=C0189}
ISO/IEC 9126 {id=C0166}
ISO/IEC 25010:2011 {id=C0167}
arc42 {id=C0168}

Engineering a Safer World C0174

Engineering a Safer World - Systems Thinking Applied to Safety by Nancy G. Leveson MIT Press, 2011. ISBN: 9780262016629

Internet Security C0172

Internet-Security aus Software-Sicht by Kriha/Schmitz Springer, 2008. ISBN 978-3-540-22223-1

High Integrity Software C0173

High Integrity Software - The Spark Approach to Saftey and Security by John Barnes Addison Wesley / Pearson, 2003. ISBN 978-0-321-13616-0

Software Architecture in Practice C0165

Software Architecture in Practice (3rd Edition) by Len Bass, Paul Clements, and Rick Kazman. Addison Wesley / Pearson, 2013. ISBN 978-0-321-81573-6

Software Estimation C0171

Software Estimation - Demystifying the Black Art by Steve McConnell, Microsoft Press, 2006. ISBN:0735605351

Software Quality Engineering C0189

Software Quality Engineering - A practitioners approach by Witold Suryn Wiley, 2014 ISBN 978-1-118-59249-6

ISO/IEC 9126 C0166

ISO/IEC 9126 Software engineering — Product quality

ISO/IEC 25010:2011 C0167

ISO/IEC 25010:2011

arc42 C0168

https://arc42.org