Test Driven Development: Requirements and Test Design

Gregory Solovey

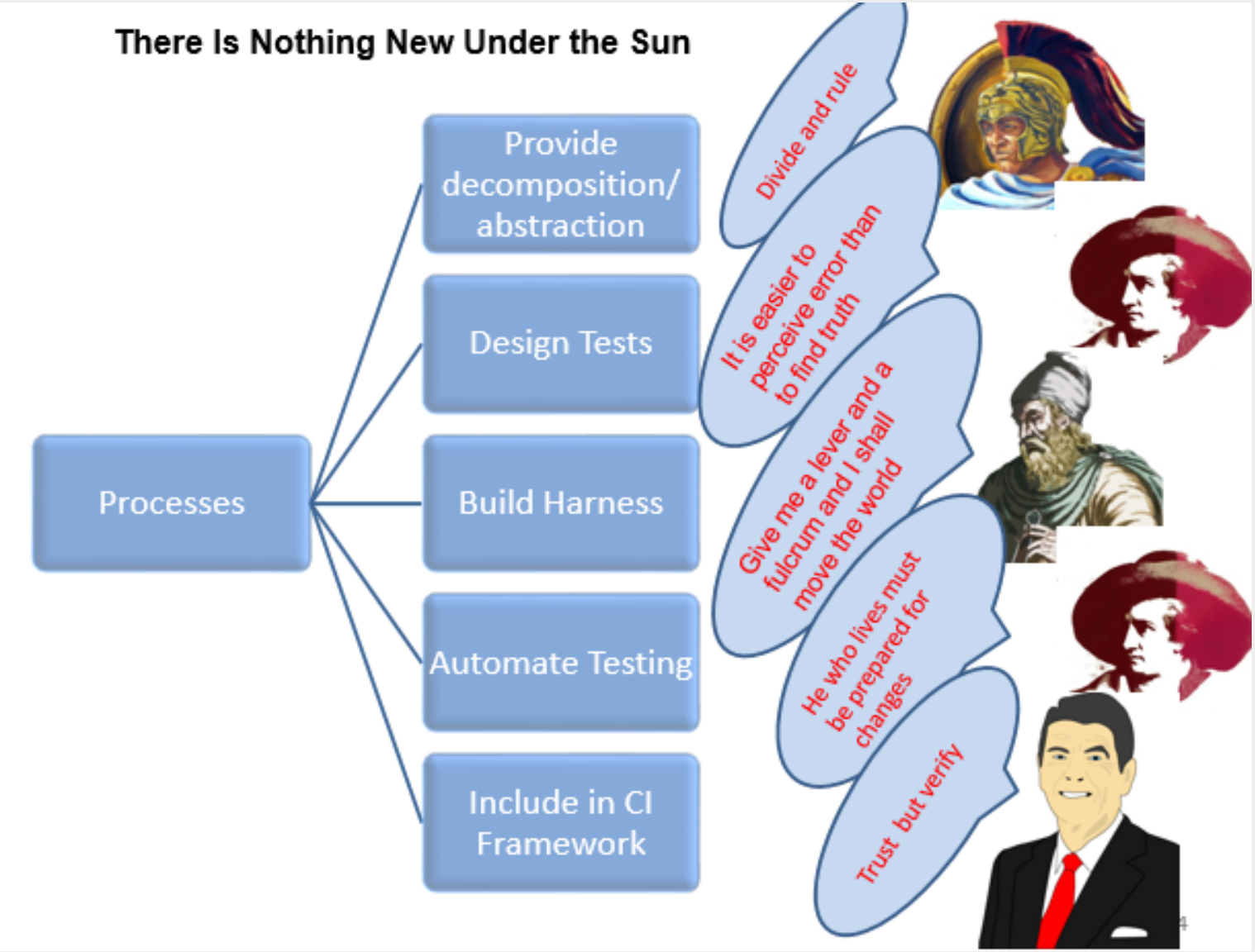
NOKIA

Agenda

- Introduction
  - goal, objectives
- Part 1. Test-Friendly Requirements
  - attributes of ideal tests
  - attributes of ideal requirements
  - insist on ideal requirements
- Part 2. Embed Test-Friendliness
  - examples
  - exercises

Introduction

Test Automation Processes



Goal

Our ultimate goal is to deliver products with zero implementation defects.

Usual attributes of requirements' quality are their completeness and consistency.

This presentation emphasizes the requirements' testability aspects – properties that allow them to be completely and efficiently tested.

Objectives

We can never be sure that the requirements are correct and we can never be sure that their implementation is correct.

But it is absolutely certain that testable requirements save enormous effort and time to verify the code and maintain the tests. Moreover, when continuous delivery is in place but the code quality is poor, the testware quality can only be improved in the presence of requirements' testability.

Requirements' testability makes it possible to:

- design tests in a day
- find all implementation errors
- maintain the test mapping with the requirements

Part 1

Test-Friendly Requirements

Ideal Requirements

A Google search on “requirements testability” will return the following criteria:

Atomic, Complete, Consistent, Controllable, Formalized, Non-Conjugated, Observable, Quantitative, Traceable, Unambiguous, Unitary

The common sense definitions of these criteria are not always obvious guidelines for creating “testable” requirements. We will attempt to present such guidelines (the devil is in the detail).

From Test to Requirements

First, we will define what an “ideal test” is - what are its attributes and constraints.

Next, we will infer the attributes and constraints of requirements which support ideal tests.

Last, we will discuss how to create, verify, and implement ideal requirements.

Ideal Test

1. **Complete**
   - 1.1. contains clearly set pre-conditions, stimuli, and expected results
   - 1.2. covers all requirements
   - 1.3. covers all implementation errors
2. **Effortless to create**
   - 2.1. uses known test design methods
   - 2.2. uses manageable model sizes
3. **Reviewable**
   - 3.1. supports the test object hierarchy, to reduce the complexity
   - 3.2. mirrors the requirements structure
4. **Executable**
   - 4.1. is controllable: can be initiated from external interfaces
   - 4.2. is observable: the execution results are accessible
5. **Maintainable**
   - 5.1. initiates a single testware update for a single requirement change
   - 5.2. is traceable to requirements to support debugging

Complete

1.1. Contains clearly set pre-conditions, stimuli, and expected results

Top down object description

The requirement document should present the forest before zooming-in on the trees (deductive approach). The documents should be presented from higher to lower levels of abstraction, i.e. from business logic to implementation details. In this case the pre-conditions can be inferred from the path and position of the particular requirement in the document hierarchy.

Unambiguous

Each requirement should allow to clearly define the expected results.

Complete

1.2. Covers all requirements

Traceable

Implement tagging for each requirement. The tags can then be included in the tests, to link them to the requirements. This allows to calculate the requirements coverage.

Covers all implementation errors

1.3. Covers all implementation errors

Use known software models

A test is designed to uncover an error. Errors are well defined for formal models (UML-like). If a system component is described informally in business terms (in spoken language), only a subject-matter expert can define the associated tests. However, the completeness of such a test cannot be proved. The completeness can be achieved only for a test built to cover all conventional errors of a formal model. There are a few types of commonly used formal models:

- Structural models: object attributes, configurations, formats, interfaces, messages
- Behavioral models: use cases, algorithms, state machines, sequence diagrams

Complete

1.3. Covers all implementation errors

Use known software models

Once the models are identified, we can identify the implementation error classes.

> All implementation errors fall in the “swap” category, where one of the object’s elements is inadvertently swapped with another element of the same type. Such errors cannot be detected by a syntax analyzer.

For known software models there are known test design methods that guarantee the minimum set of test cases that can detect all known error classes.

Complete

1.3. Covers all implementation errors

- Arithmetic expressions: α + β

  Error Model: α ➔ β, α ➔ constant; '+' ➔ '-'
- Relational expressions: α == β

  Error Model: α ➔ β, α ➔ constant; '==' ➔ '<='
- Logical expressions: α ∧ β

  Error Model: α ➔ β, α ➔ constant; '∧' ➔ '∨'
- Algorithm: a set of functional and conditional blocks and the connections among them.

  Error Model: expression and connection errors
- State Machine: a set of states and a set of events that transfer control between states and generate outputs.

  Error Model: transfer and output function errors

Effortless to Create

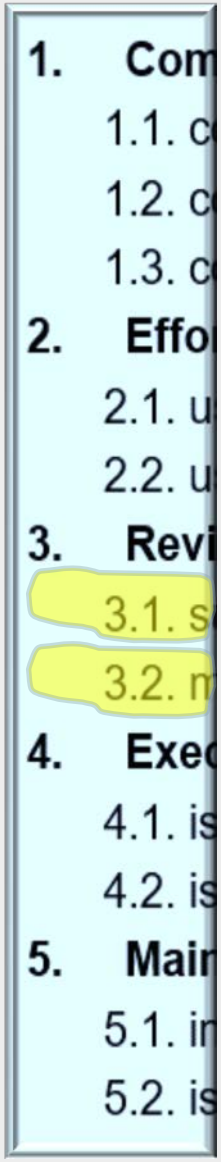
2.1. Uses known test design methods

2.2. Uses manageable model sizes

Limit model size

In the software testing world, in contrast with VLSI, the automation of test generation is very rarely used. Therefore, independent software components have to be reasonably sized for manual test creation. The size of a requirement that a tester can analyze independently might be limited to approximately ten objects: states in a finite automaton, "if" statements in an algorithm, objects and methods in an OO model, message exchanges in a sequence diagram, elements of GUI, etc. The approach is to "divide and conquer". If the subsystem description is much bigger than the “desired” size, then some elements (nodes of state machine, algorithm blocks, message sequence, etc.) have to be combined in separate units and presented as children requirements.

Reviewable



3.1. supports the test object hierarchy, to reduce the complexity

3.2. mirrors the requirements structure

Top-down hierarchy of software models

Limit model size to 10 elements

It should be possible to evaluate the completeness and correctness of the test in the limited time of a test review. To do this, the hierarchy of test cases should to be mapped directly to the requirements, which are presented as a top-down hierarchy of software models.

Moreover, each model should consist of no more than 10 elements, to make it easy to “absorb” and validate the respective test set in real time.

Executable

4.1. is controllable: can be initiated from external interfaces

4.2. is observable: the execution results are accessible

Test harness - input to design document

Each test case must comply with the testability prerequisites, namely controllability and observability. Each test case should be able to be launched from an external interface (accessible by the tester) and the results of its execution should be available for a verdict. However, the nature of the object-to-test (especially for embedded systems) is that sometimes there is no direct access to APIs, messages and internal attributes that are necessary to execute the test.

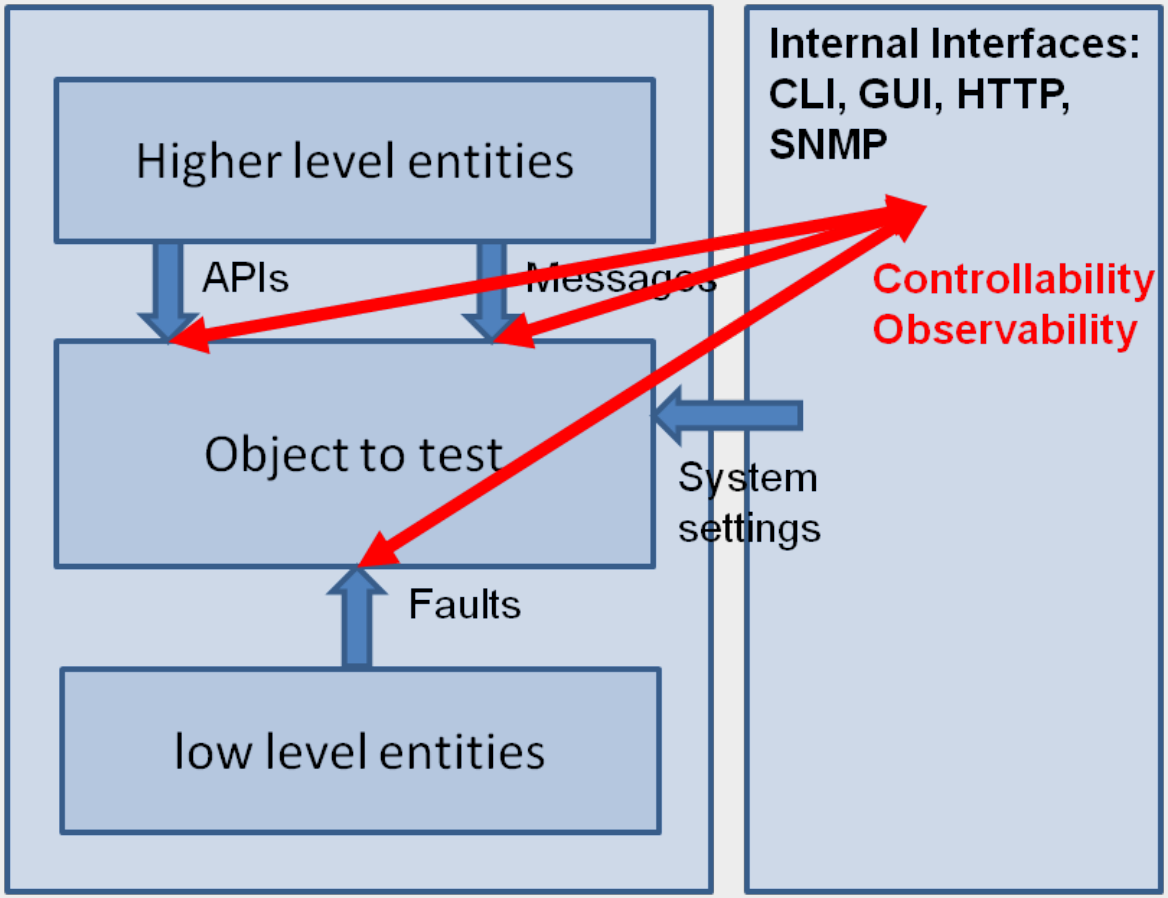
A test harness is a set of "instruments" that provides the ability to access APIs, test functions, database content, or to send messages. A test harness can be implemented as a set of CLI commands, GUI objects, or http requests.

Executable

1. Com
   1.1. c
   1.2. c
   1.3. c
2. Effo
   2.1. u
   2.2. u
3. Revi
   3.1. s
   3.2. m
4. Exec
   4.1. is
   4.2. is
5. Mair
   5.1. in
   5.2. is

4.1. is controllable: can be initiated from external interfaces

4.2. is observable: the execution results are accessible

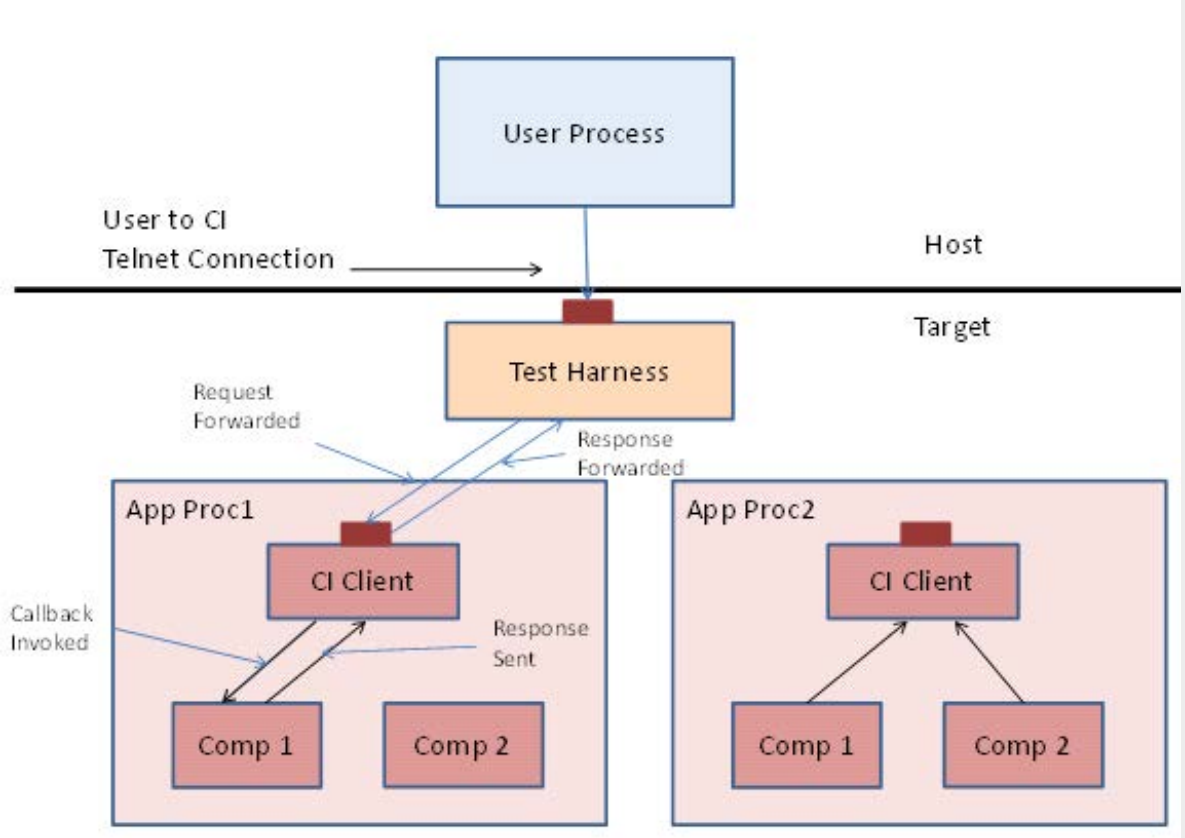


Executable

4.1. is controllable: can be initiated from external interfaces

4.2. is observable: the execution results are accessible

CLI access



Executable

4.1. is controllable: can be initiated from external interfaces

4.2. is observable: the execution results are accessible

HTTP access

Structure a url as a RESTfull API:

http://<object>.com:10080/<appl>/<api_name>/<parameter>?token=<value>

DB access

Define keywords which are translated into SQL queries, from templates stored in libraries. Provide connection to the database and return results to the user interface.

Executable

4.1. is controllable: can be initiated from external interfaces

4.2. is observable: the execution results are accessible

Recommendation for test harness implementation

- Access all object APIs: create, activate, discover, delete, send, etc
- Print the states and values of the object or group of objects in TLV, Json, xml format
- Provide routing to a particular harness command, using UNIX-like syntax (“cd”, “ls”, “help”, “show”)
- Report errors using a predefined syntax
- Redirect system messages to the external interfaces
- Do not perform any checks other than syntax
- Return acknowledgement and completion messages for all harness commands

Maintainable

5.1. initiates a single testware update for a single requirement change

5.2. is traceable to requirements to support debugging

Traceable

Using tagged requirements will speed-up debugging and defect creation.

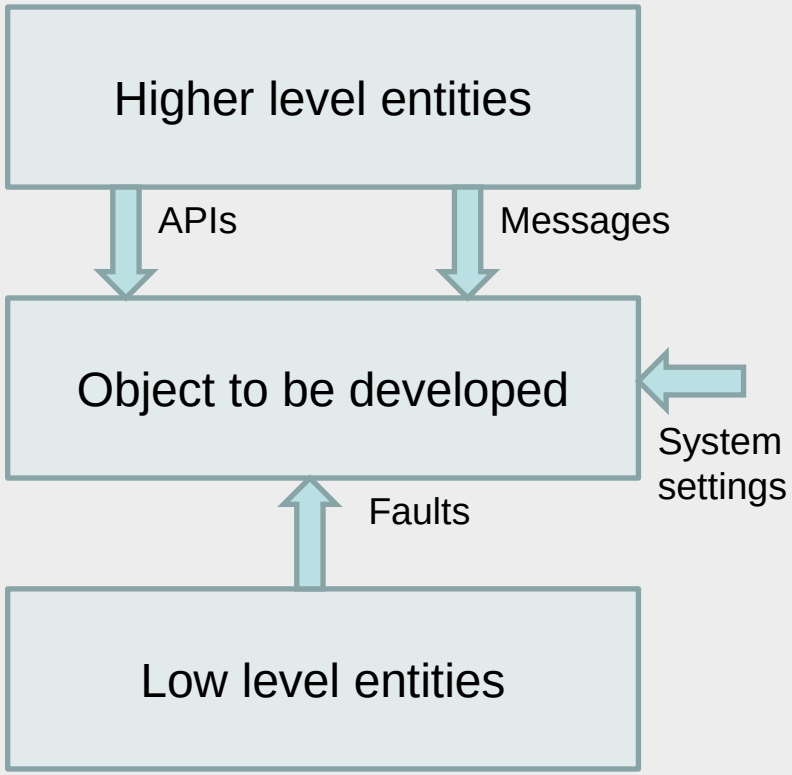
Avoid model duplication

The models and their elements used by other models (for example messages or algorithms) should be uniquely defined. All models that need to reuse a particular element should reference it through the requirement tag. This ensures that only one set of test cases will be created. Any changes in the model will trigger the respective testware changes in a single place.

Summary - Ideal Requirements

- Present an object as a top-down hierarchy of software models as deep as necessary, where an element of the model can be elaborated by another model.
- Each model is to be presented in a clear and unambiguous manner, using UML-like models, such as use case, algorithm, state machine, sequence diagram, syntax, condition, etc.
- Include each model into a requirement as an mandatory section. Optional sections may include informal descriptions of the functionality, pre-conditions, triggers, post-conditions.
- Identify each requirement in the document with a unique tag.
- Use references to existing models instead of duplicating them.
- Each model should consist of no more than 10 elements, which is practical for manual test generation.
- Only the requirements' model sections will be developed as code and tested.

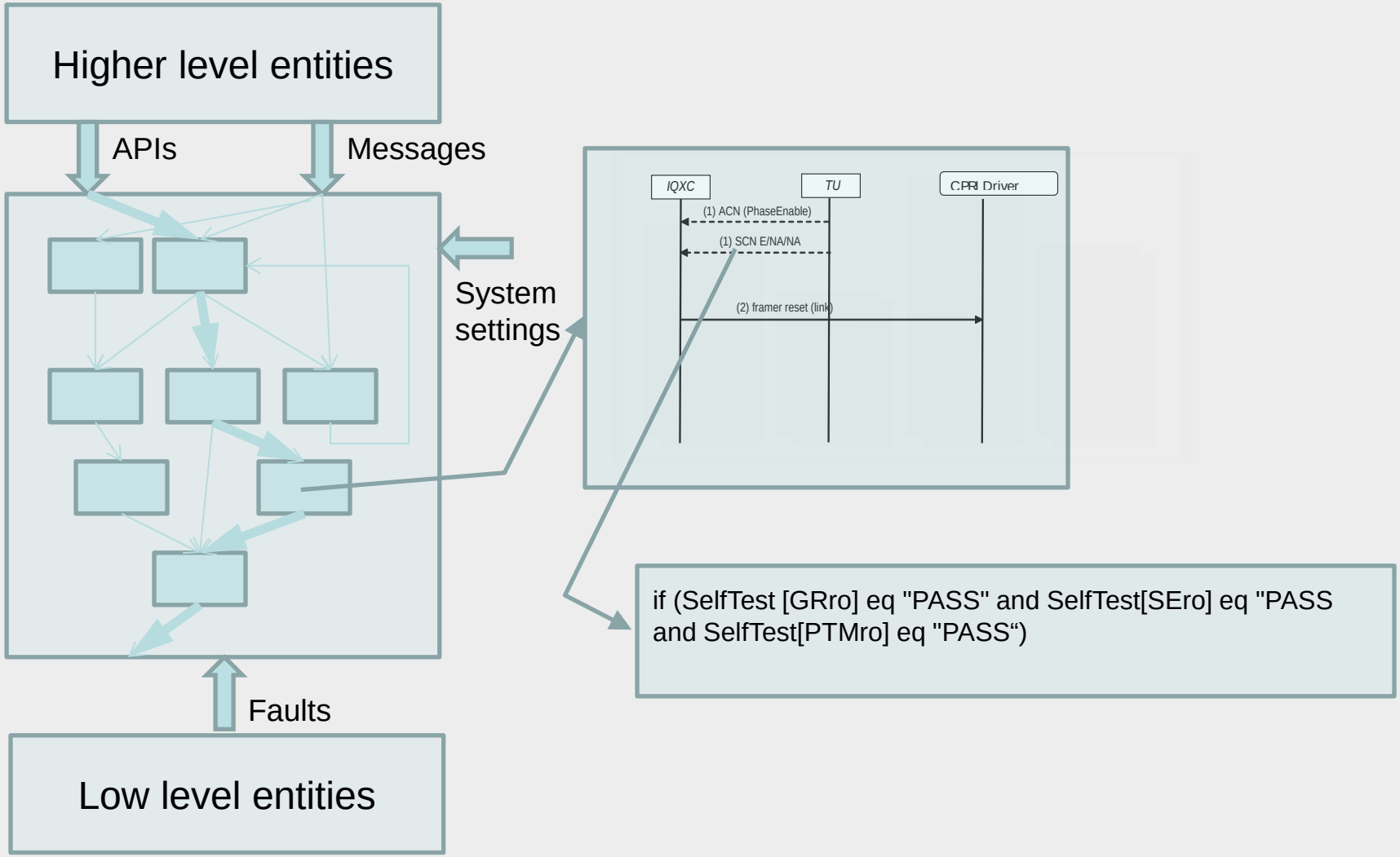
Object



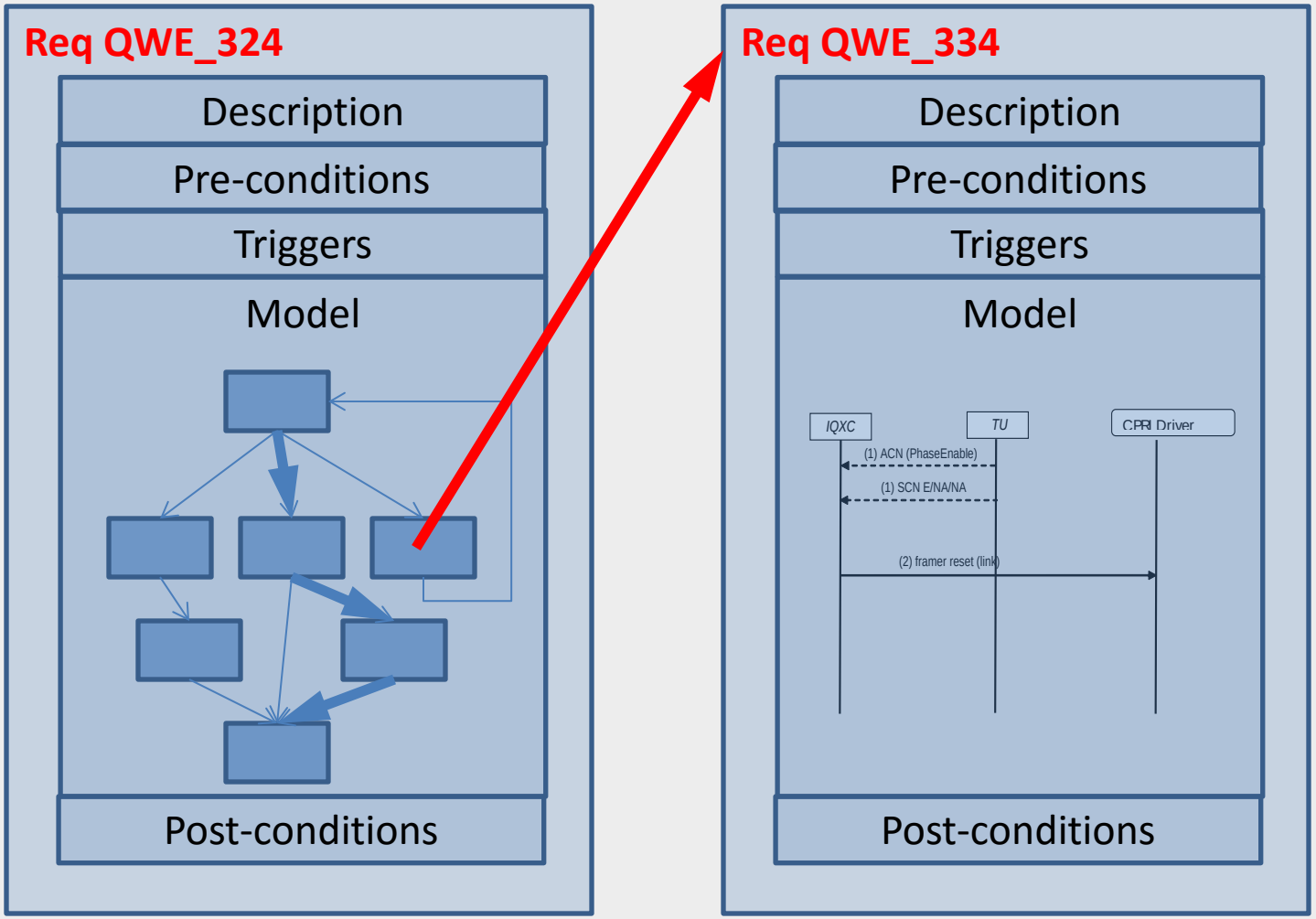
Modern software is architected as layered (often embedded) systems. Lower layer modules provide services to the higher layer ones.

Similarly, a requirement document should first present the scenarios of the higher layers, and then with each layer, increase the level of detail by specifying the structure, functionality and implementation details.

Model Hierarchy



Requirement Hierarchy



Document Formats

Should we use general document formats (Word, Acrobat, PowerPoint) or use specialized tools (No Magic MagicDraw or Rational Rose Modeler)?

When using general format documents, the test-friendly requirements guidelines have to be reinforced during the review.

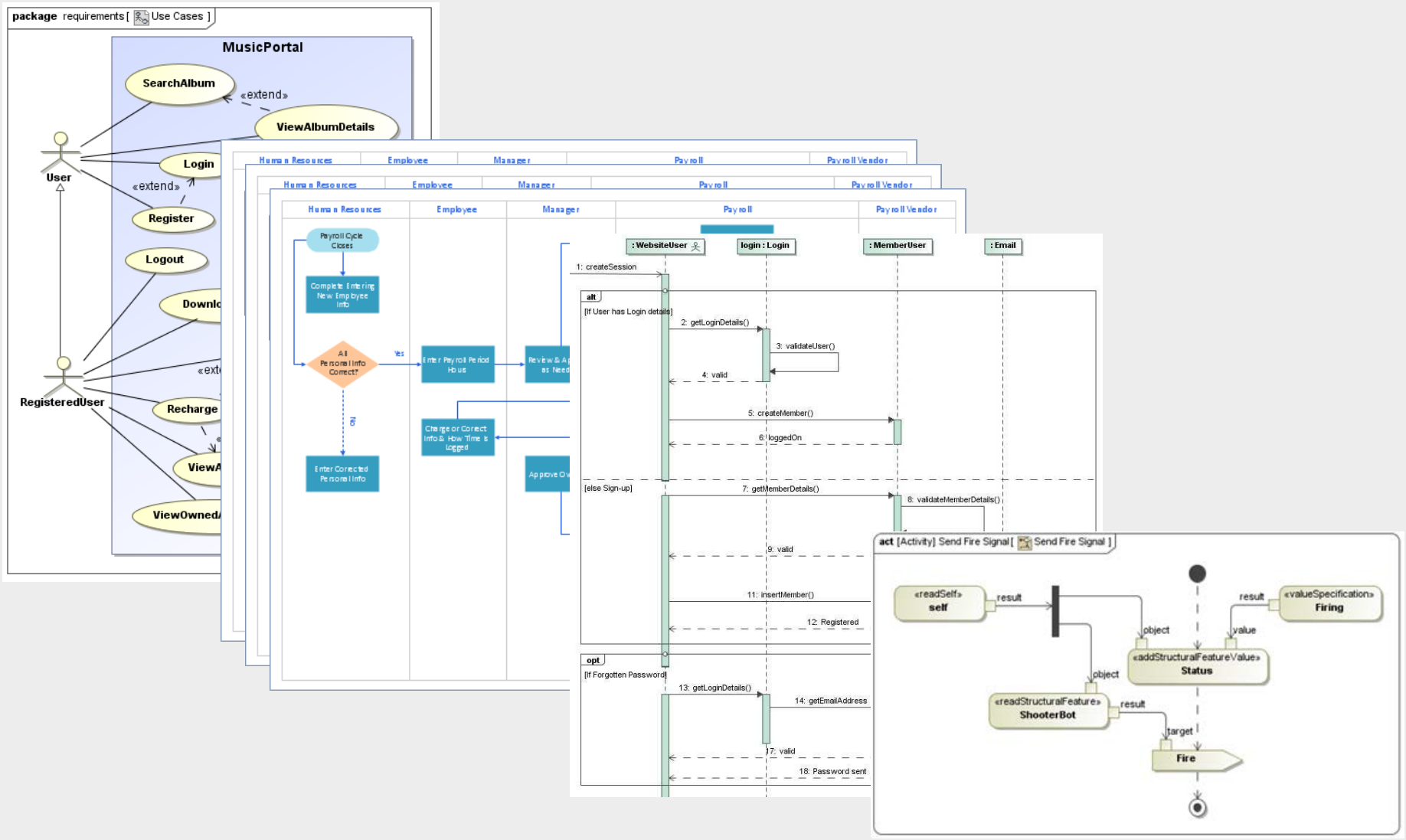
Even modeling tools do not support the complete set of guidelines.

Model Hierarchy – Nokia

Let’s take a look at an approach using MagicDraw.

- 1st level: Use case diagram: describes the main functionality based on customer requirements
- 2nd level: Activity diagram: describes the functionality as algorithms and/or state machines
- 3rd level: Lower level activity diagram: describes elements of the 2nd level activities in more detail
- 4th level: Sequence diagram: describes the interactions between components of the 3rd level during the execution of an activity
- 5th level: Individual activity diagram: describes the behavior of 3rd and 4th levels’ components .

Model Hierarchy – Graphical View



Document Tools

Modeling tools are better than general-purpose document tools. However, they do not support the testability guidelines out of the box.

“ … So we created a hairy billiards ball, now let's think how to use it…”

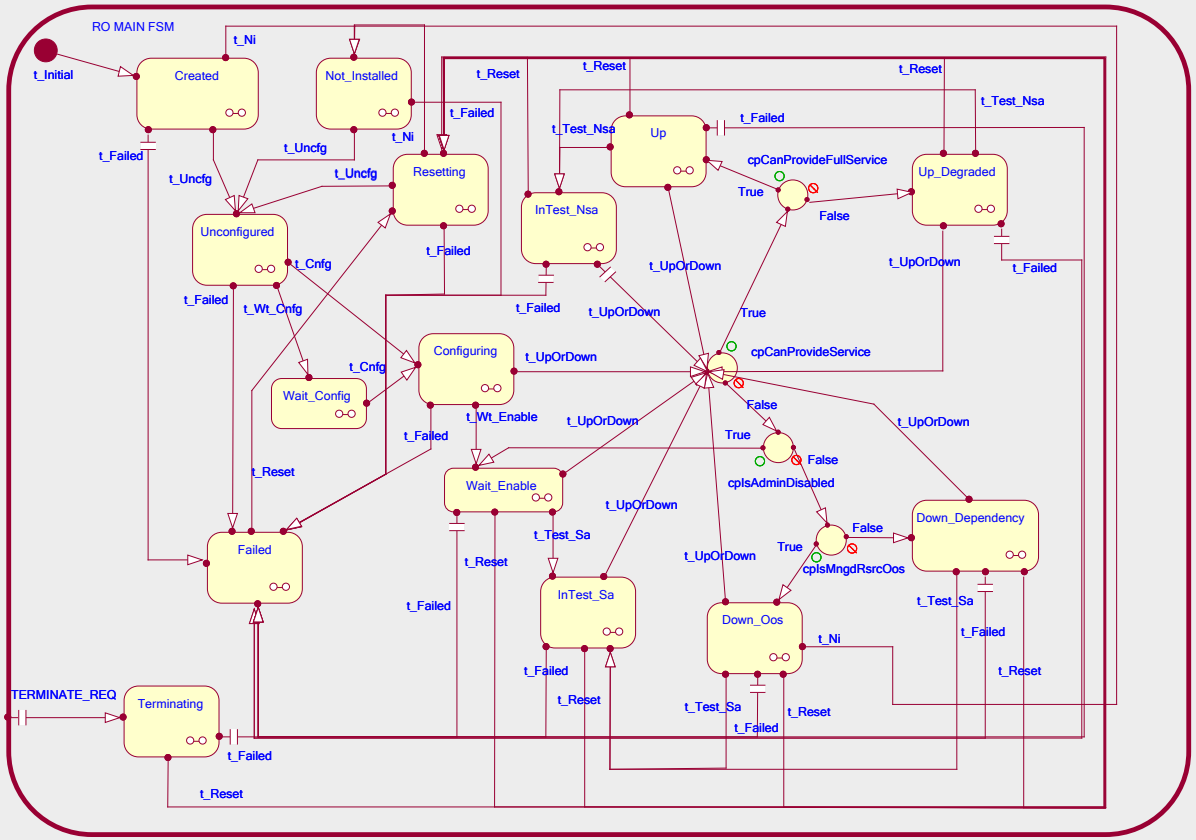
A Tool is not a Panacea

Out of the box MagicDraw still allows to:

- define models outside a top-down hierarchy
- present a model as plain English text, for example, in the functional box of an activity diagram, the comments section, or the requirements table, etc.
- have a model without a unique tag
- use multiple copies of unique models
- have models with unmanageable numbers of elements

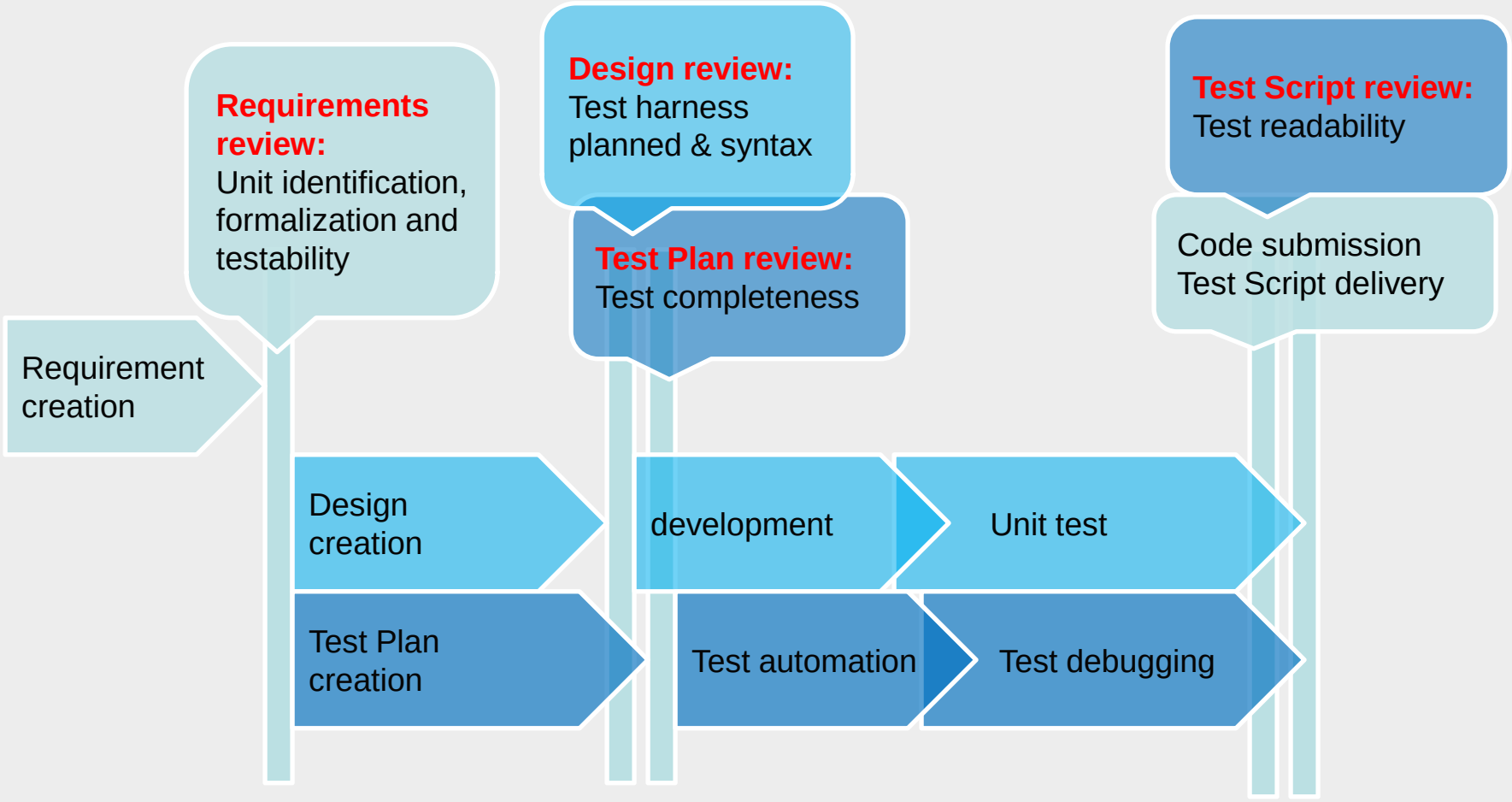
Of course, it is possible to create verification rules in MagicDraw to enforce the guidelines. However, the point is that, without additional customization, none of the tools explicitly support Test Friendly Requirements guidelines.

Unmanageable Model Size



This “Resource Object” state diagram has 15 states, 4 decision blocks, 75 transitions with conditions. Testing will require more than a thousand test cases, and the review of their completeness is not feasible.

Review Processes



Requirements Review Process

During the requirements review testers should verify that the document complies with the test-friendly guidelines:

- the document is presented in a top-down hierarchical structure.
- the document is split into tagged requirements.
- the model section of the requirement description is presented as a known software model.
- models have acceptable sizes.
- each requirement has five sections: definitions, pre-conditions, triggers, model, and post-conditions.
- all object attributes, messages, and APIs are accessible from external interfaces. If not, request should be made to developers to implement.

Design Review Process

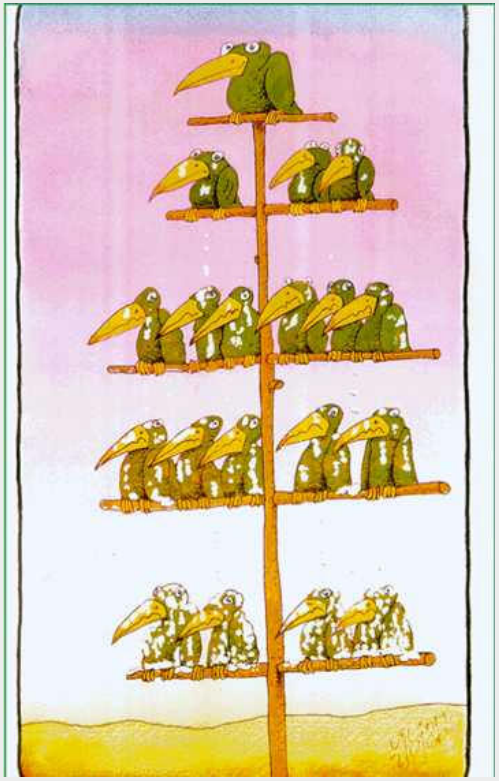
Developers should implement a test harness prior to testing. This requires that additional time be allotted in the project plan. Based on our observation, the overhead is approximately 5%. Developers always dedicate time to incorporate hooks, simulators, code traces in order to execute their unit test. The same debugging code can be reused as a base for the test harness.

Note that the test harness has to be defined during the requirements review and it has to be available at code submission time.

Test Plan Review Process

Reviewers should make sure that tests:

- mirror the structure of the object-to-test (hierarchy of models). This simplifies the test maintenance and reduces the time to find errors.
- cover all implementation errors. This is done by checking that known test design methods are used.



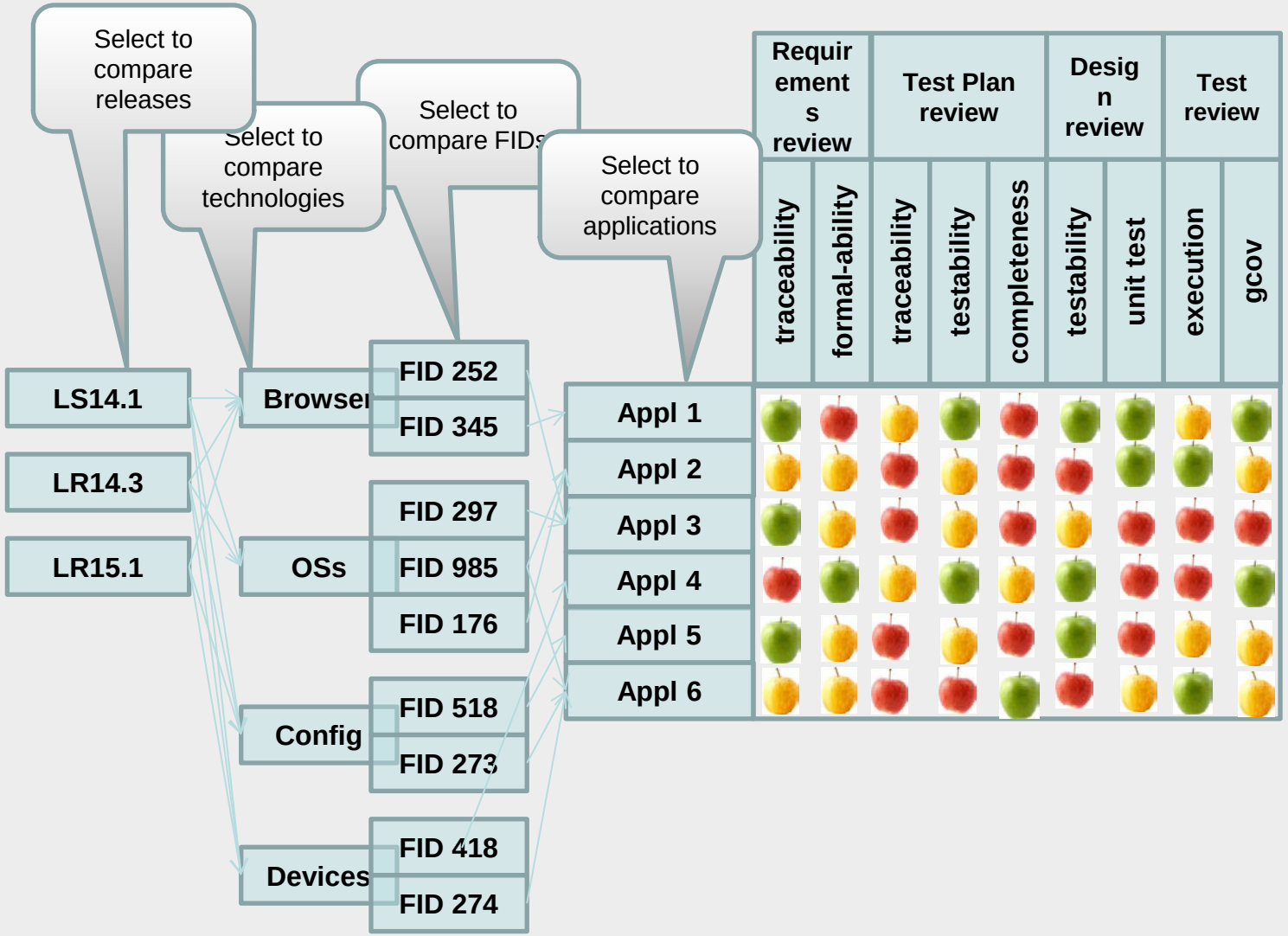
Test Set (architectural-level sub-system, for example) consisting of
Use Cases (a scenario of the sub-system), where each use case consists of
Test Cases (verification points of a scenario), and each test case consists of
Test Actions (a single communication act with the object-to-test).

Quality Monitoring

A system is as good as its weakest link. It is important that organizations monitor the test quality by employing dashboards that present the following quality metrics (at the release/project/feature level):

- Requirements' test-friendly format usage
- Requirements' coverage by tests
- Test harness implementation by development groups
- Implementation error coverage by tests
- Test coverage by automation
- Code coverage by automated tests

Quality Dashboard



Part 2

Embed Test-Friendliness

Not So Friendly Requirement

<Start Requirement ABC_456>

Theoretically, an architect, developer, and tester need to prove requirements completeness and consistency, but practically, each have different objectives during the requirements review process. An architect's main goal is to check the completeness of requirements, their consistency, and how they reflect the customer expectations. The developer tries to make sure that the requirements provide enough data to verify that all attributes, messages from external interfaces, and commands are acceptable from external make notes if request access to pre-conditions and triggers which should then be implemented by the developers. The tester checks which should be testable. Verify that the model section of the requirement description is presented as one of the traditional software models: State machine, ladder diagram, algorithm, Syntax, instruction set, etc.

<Finish Requirement ABC_456>

Plain English

The attribute ***BIST_RESULT (Self Test Result)*** represents the aggregated initialization results (PASS/ FAIL) from all of the drivers managed by the TimingUnit.

---

on the ES interrupt, the X_Monitor performs a “double reading” of the TOD (Time Of the Day) to recover the GPS TOD for the next ES. It means the X_Monitor reads the framer register of HSIQ driver with framerid#0 on this ES interrupt and repeats the reading on next ES interrupt to confirm the accuracy of the HSIQ TOD. If confirmed, GpsTime TOD on last ES is initialized and Linux timestamp is memorized.
On each next following ES interrupt, GpsTime TOD on last ES is incremented by two and Linux timestamp is memorized again. To maintain synchronisation between GpsTime TOD and HSIQ TOD, every hour, on an ES interrupt, the X_Monitor repeats the “double reading” of the HSIQ TOD. If HSIQ TOD is consistent, and different from GpsTime TOD on last ES, the GpsTime TOD is reinitialized to the new value. In case of failure of HSIQ links, HSIQ TOD is no more available but GpsTime TOD on last ES is still available, it is incremented by two on each ES interrupt. When HSIQ TOD is available again, a synchronisation to HSIQ TOD is performed.

The requirement is vague: What are the drivers? Do all drivers count? How to simulate “fail” driver self test verdict?
The description is in plain English. None of the architects/ developers/ testers were able to explain this description without looking at the source code.

Models Mix

The attribute ***CLOCK_DELAY*** specifies the delay in the link between the NODE input, and its clock source. It allows the NODE to compensate for this delay and achieve a more accurate phase alignment when <CLOCK> is the synchronization source. Once get the message and the current sync source is <CLOCK>, the NODE shall send SYN_DELAY_COMPENSATION to drvSyn for the delay compensation. Once there is clock switch, NODE shall stop the drvSyn to stop the delay compensation.

The fault ***FAULT_GPS_RECEIVER_FAIL*** is raised by the TimingUnit if GPS is in the sync source list and the TimingUnit receives this fault from the GPS indicating that there is a communication failure to the GPS receiver. This is of major severity and causes TimingUnit to transition to “Enabled / Degraded” state. Additional info field should be filled with the content of the additional info field from GPS fault notification. The fault can also be generated by TimingUnit directly, when it detects that current protocol is mismatching with provisioned priority list.

The object description (an attribute, a fault) and the model (scenario/algorithm), where this object is used have to be separated. Without the full context, it is challenging to understand how the action has to be taken, what are the values to set and how to verify them. The tester should not have to piece together an algorithm from pieces found in different object descriptions.

Vague Descriptions

“...Its responsibilities include, but are not limited to...”
“...performs timely switch...”
“…achieve a more accurate phase alignment…”
“…most likely in 20-40 seconds ranges...”

The description is vague.

Attempt to Implement

Arthur Schopenhauer
“Alle Wahrheit durchläuft drei Stufen. Zuerst wird sie lächerlich gemacht oder verzerrt. Dann wird sie bekämpft. Und schließlich wird sie als selbstverständlich angenommen.”

Stage 1 - *ridiculed*

“Testers’ prerogative is to learn the business and technology in order to verify their implementation, not to dictate to us how to change the documentation culture”.

Stage 2 - *violently opposed*

“We (system engineers and architects) are the experts, and we know that too much formalization and formatting will dramatically slow down the requirements’ delivery”.

Stage 3 - *accepted as being self-evident*

“This is well known stuff and we already monitor traceability, conduct reviews and use UML models where possible”.

Implementation

The following are required to implement complete test:

1. **Support** of other non-test groups: architecture, development, release management and quality

2. **Authority** to request support from other groups

Instructions

For the purposes of this exercise we will use simplified instructions:

- Describe an object as a top-down hierarchy of software models, as deep as necessary, where an element of the model can be elaborated by another model.
- Each model is to be presented in a clear and unambiguous manner, using UML-like models, such as use case, algorithm, state machine, sequence diagram, syntax, condition, etc.
- Each model should consist of no more than 10 elements

We add one more rule to clarify the meaning of “as deep as necessary”:

- The process of model elaboration should stop at the level where programming language data structures are specified (arrays, hashes, pointers, DB schemas, etc).

Examples

Triangle Example

A function that reads three numbers that are the lengths of triangle edges and returns the triangle type.

Iteration 1: informal requirements

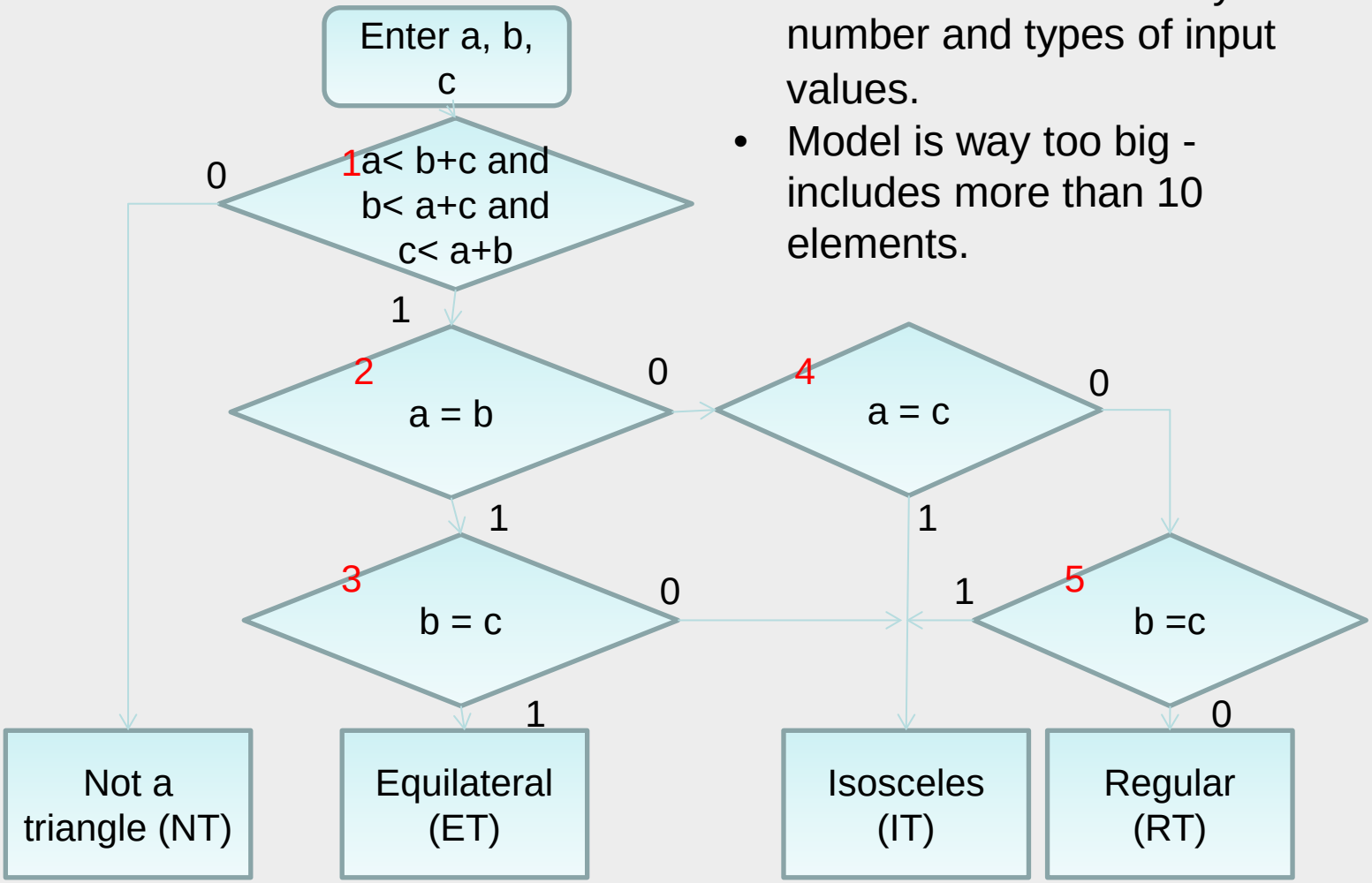
> A script reads three numbers that are the lengths of a triangle's edges and returns the triangle type.
>
> No initial data boundaries and possible outputs are specified

Iteration 2: informal requirements

> A script reads three numbers whose value is less than 30, representing the lengths of a triangle's edges, and returns one of four messages: "These numbers cannot form a triangle (not a triangle) - NT"; "equilateral triangle - ET"; "isosceles triangle - IT"; "regular triangle -RT".
>
> No algorithm is specified

Triangle Example

Iteration 3: formal requirements



- Model is incomplete – there are no blocks that verify the number and types of input values.
- Model is way too big - includes more than 10 elements.

Triangle Example

Iteration 4: test-friendly requirements

Req #	Model
Req_1	Model: algorithm of black boxes If ((side's boundary conditions incorrect::req_2) OR (sides cannot form a triangle"::req_3)) { print NT; } elsif (a triangle is equilateral ::req_4) { print ET; } elsif (a triangle is isosceles ::req_5) { print IT; } else { print RT; }
Req_2	Model: syntax of application attributes • There are three mandatory attributes of the application • Each attribute type is a digit • Each attribute is a positive number
Req_3	Model: a condition on the values of three sides of the triangle ((a< b+c) and (b< a+c) and (c< a+b))
Req_4	Model: a condition on the values of three sides of the triangle ((a=b) and (b=c))
Req_5	Model: a condition on the values of three sides of the triangle ((a=b) or (b=c) or (a=c))
Req_6	Model: syntax of output messages • There are four output messages • Format of these messages is two symbols: (NT; ET; IT; RT)

Triangle Example

TC #	description	a	b	c	result
1-14	Enter less than 3 parameters , enter symbols, enter negative and 0 values for each parameter				NT
15-17	a<b+c: 3 cases for predicate “<“, and each variable needs to be changed	6 6 11	2 3 3	3 3 9	NT NT RT
18-20	b<a+c	2 3 3	6 6 11	3 3 9	NT NT RT
21-23	c<a+b	2 3 3	3 3 9	6 6 11	NT NT RT
24-26	2: a=b: three cases for predicate “=“	11 11 3	3 11 11	9 9 9	RT IT RT
27-29	3: b=c (a=b)	3 11 11	3 11 11	2 11 3	IT ET IT
30-32	4: a=c (a<>b)	3 11 11	1 9 9	2 11 3	RT IT RT
33-35	5: b=c (a<>b and a<>c)	1 9 9	3 11 11	2 11 3	RT IT RT

A Simple Calculator in Google Search

Iteration 1:

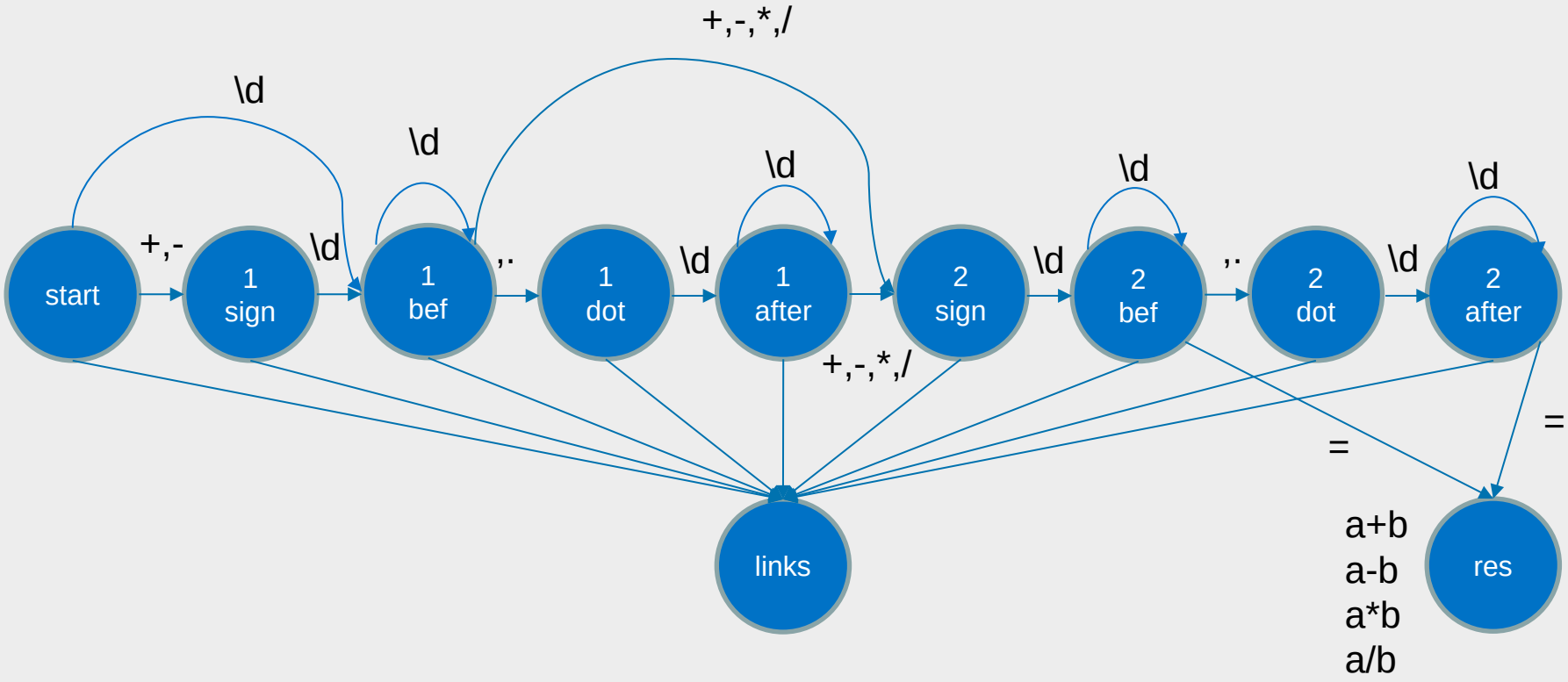
Enter two numbers (a,b) separated by an operator (+,-,*,/) and then enter “=”

Iteration 2:

Enter input string: [+-]?\d+([,.]?\d*)[+-*/]?\d+([,.]?\d*)=

Observe result as \d+(.?\d*)

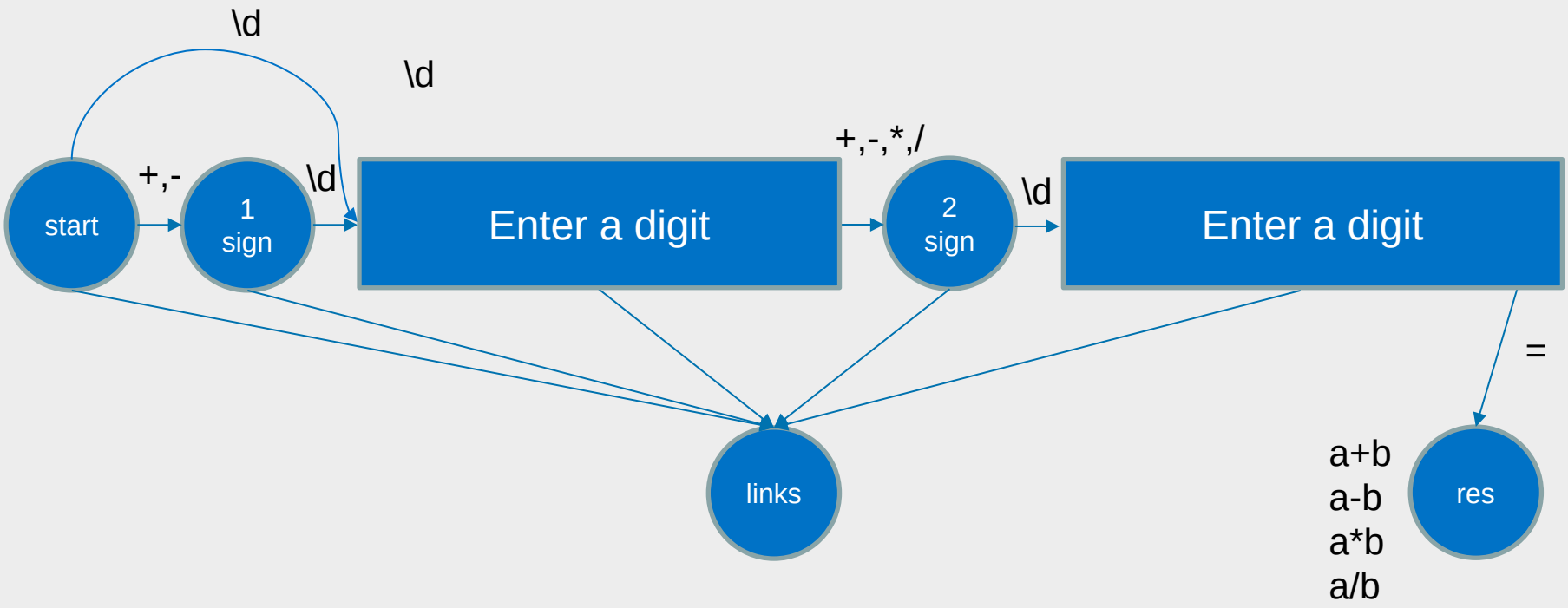
Iteration 3: State machine



A Simple Calculator in Google Search

Iteration 4:

Hierarchy of state machines



Ten Most Frequent Words

A function for efficiently displaying the ten most frequent words in a file

Iteration 1: Hierarchy of algorithms

1. select all words from the file to a set of words:
   Words are “\w+ (‘word’ and ‘_’ characters)”, separated by “\s+ (spaces, tabs, \n, \r)”
2. order all words (sort the set)
3. count instances of each word in the ordered set
   (count the identical words in the sequence, save the number in the respective hash)
4. find the ten biggest numbers (sort numbers in the hash)
5. select the first ten words in the ordered hash

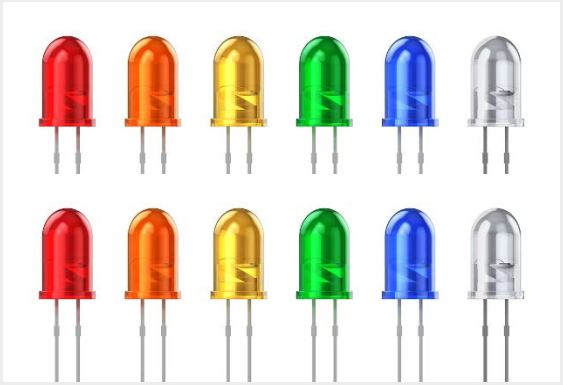
Iteration 2:

LED

The panes contain a LED row for each T1/E1 port. When a dial feature card is up and an alarm is received on the associated T1/E1 port the respective LED should be lighten. T1/E1 Alarm can be associated with the one of the following: “loss of alignment”, or “loss of multi-frame” at the local or remote node. In addition, the same LED indicated receiving of ‘Alarm Indication Signal” from a board XYZ, or “RED Alarm” from a board ABC

Iteration 1: Logical Condition

- DFC - a dial feature card is up
- LOA – receiving of a “loss of alignment” alarm
- LMF – receiving of a “loss of multi-frame” alarm
- AIS – receiving of a “loss of alignment” alarm
- REA - receiving of a RED Alarm
- B-XYZ – the board is XYZ
- B-ABC – the board is ABC



For each T1/E1 the respective LED goes up if:
(DFC ***and*** (LOA ***or*** LMF ***or*** (AIS ***and*** B-XYZ) ***or*** (REA ***and*** B-ABC)))

Branch Delay

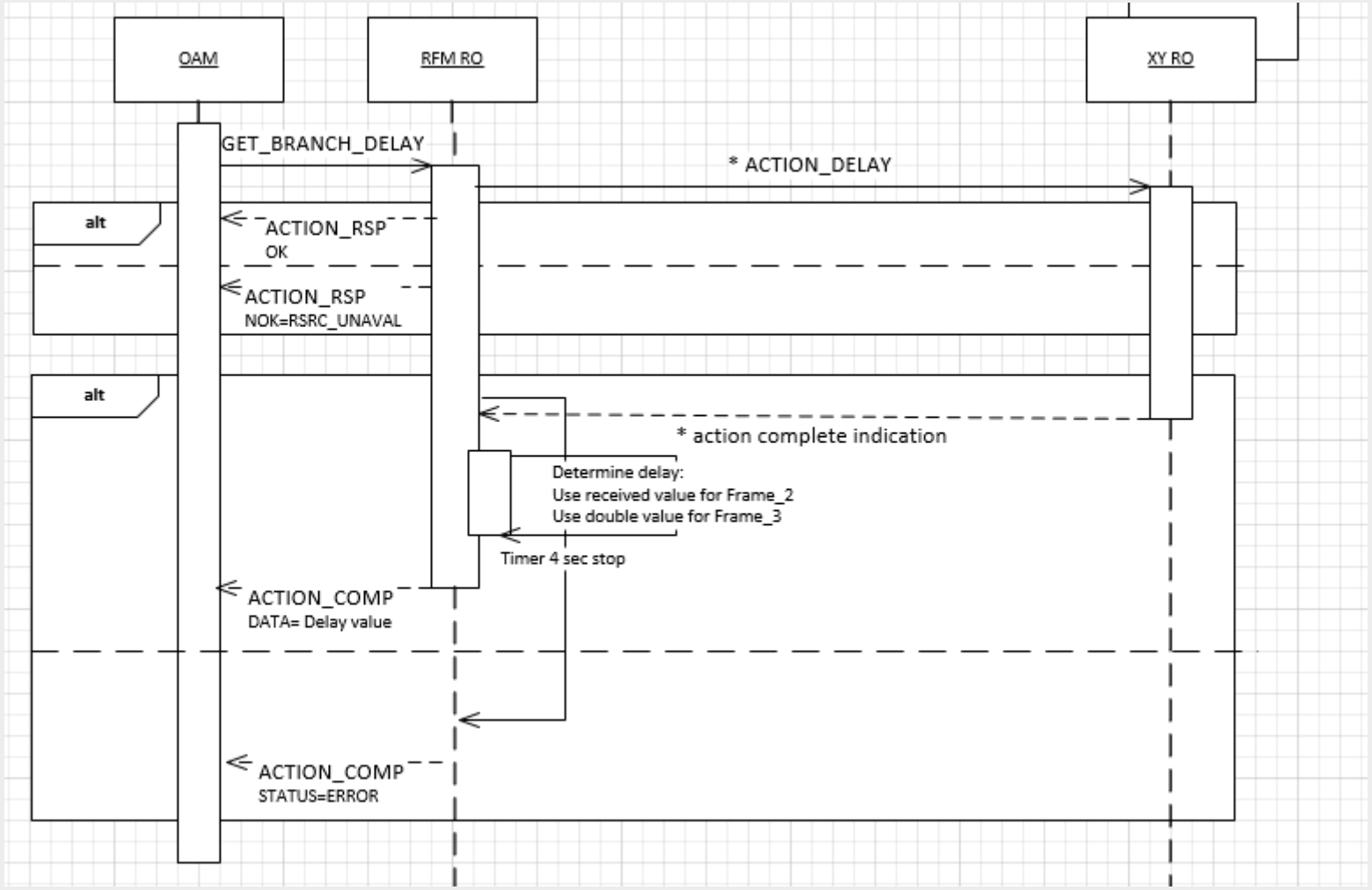
<Requirement 124-2. Branch Delay>

The OAM sends to RFM RO “GET_BRANCH_DELAY” action request, and as a result, the RFM RO requests delays of all associated XY units. The RFM RO requests delay from each XY RO via “ACTION_DELAY” message. In the case where RFM unit is connected to a XY unit, the RFM RO monitors the state and status changes of its XY ROs and for the absence (i.e. supervisory notification) of the XY RO. After RFM RO has indicated acceptance of the “GET_BRANCH_DELAY” action request, a timer (4 sec) is started to wait for the XY ROs replies. If the timer is expired, RFM RO sends the action complete indication with the status byte set to ERROR. If the GET_BRANCH_DELAY is received and the XY is already absent or not operationally enabled, the RFM RO sends an ACTION_RSP response with the NAK_CAUSE set to NAK_RSRC_UNAVAILABLE. Otherwise, the RFM RO return to OAM the response with delay value for XY RO with Frame2, and double value for XY RO with Frame3.

<End. Requirement 124-2. Branch Delay>

Branch Delay

Iteration 1: Use sequence diagram



Exercises

Exercises

Req-1. Select the error messages of the same type that are logged during two days of operations. All messages are stored in daily log files.

Req-2. The X-MANAGER requests to change driver attributes and Y-SERVICE sends back the confirmation. The cache should always hold in-memory the 10 most recently requested driver types.

Req-3. The module to select two Access Points with the best throughputs from all available

More Exercises - 1

<requirement 123>

If the restart reason indicates that the controller restart was unplanned, the SERVER is responsible for retrieving the restart-context trace log and save it at the predefined location. The SERVER determines whether the controller reset was unplanned based on the reset reason provided by the LEC MW service. If the application reset reason is anything other than Software upgrade or NEM, the reset is considered unplanned. If the reset reason indicates the reset was unplanned, during its initialization the SERVER will send a FILE AVAILABLE event to the X-MANAGER over the HRAL global Event Channel. The location of the CCM restart context file is defined in the reset_config file under parameter “CCM_RESTART_LOC”. SERVER waits until after it sends the ACN containing the HW Type attribute before sending the FILE AVAILABLE event to the X-MANAGER with the restart context file location.

<requirement 123>

More Exercises - 2

<Start: AL-046543-7560>

The optimal storage for historical data can be safely limited to two days (600 records). Two attributes: GOOD_DATA_SAMPLES = M and AVG_QP_FOR_SAMPLES= N are provided that are settable via the SDF file.

When the averaged qualityPercent of the records in the input data set is greater than or at least equal to N and scanning the first 6 hours of the input data set have found at least 12 records with a qualityPercent of >= 30% and scanning the last 6 hours of the input data set have found at least 12 records with a qualityPercent of >= 30% then the criteria for good data have been met.

When the averaged qualityPercent of the records in the input data set is less than N, immediately raise a FLR to indicate that M or more records were found in shortHistory.txt file, but the averaged quality Percent fell below the N threshold to indicate the criteria for good data have not been met for linear regression. If the input data set from the shortHistory.txt file doesn't contain at least M records then indicate the criteria for good data have not been met for linear regression.

<End: AL-046543-7560>

Conclusion

Further reading

Embedding testability , Professional Tester magazine, issue 27, June 2014; 8-15; Presents an approach to test embedded systems

QA of testing, Professional Tester magazine, issue 28, August 2014; 9-12; Describes the process that guaranties the test automation in parallel with code development

From test techniques to test methods, Professional Tester magazine, issue 29, November 2014; 4-14; Presents test design methods for all software models from expressions to state machine, syntax, instruction set

Everything You Always Wanted to Know About Test, But Were Afraid to Ask Professional Tester magazine, issue 32, June 2015
Provides often Q&A on how to automate testing and do it in parallel with development

Tower of Babel insights, Professional Tester magazine, issue 35, 15-18; December 2015. Proposed standards that make requirements testable

Finita la Commedia

Thank you for attending this session

Vielen Dank für die Teilnahme an dieser Sitzung

Send your comments or questions to
gregory.solovey@nokia.com