



**NTNU – Trondheim**  
Norwegian University of  
Science and Technology

# Developing Software Quality in KBE Implementations

**Lars Feiring Barlindhaug**

Master of Science in Engineering and ICT

Submission date: June 2012

Supervisor: Ole Ivar Sivertsen, IPM

Co-supervisor: Geir Iversen, Aker Solutions KBeDesign

Norwegian University of Science and Technology  
Department of Engineering Design and Materials



THE NORWEGIAN UNIVERSITY  
OF SCIENCE AND TECHNOLOGY  
DEPARTMENT OF ENGINEERING DESIGN  
AND MATERIALS

**MASTER THESIS SPRING 2012  
FOR  
STUD.TECHN. LARS BARLINDHAUG**

**DEVELOPING SOFTWARE QUALITY IN KBE IMPLEMENTATIONS  
Videreutvikling av programvarekvalitet i KBE-implementationer**

Knowledge Based Engineering (KBE) is a software technology that enables companies to significantly improve their competitive edge in design and engineering, both in large and small projects. It allows design rules and experience from previous projects to be reused in novel cases.

AML (Adaptive Modeling Language) is a programming language and development environment for KBE applications.

AUnit, a tool supporting testing practises for AML was developed in the candidate's project assignment last fall. To affect actual improvement, the tool and related practises must be implemented in the standard practises of AML development teams.

The assignment includes:

1. Expand the AUnit framework to the point where it is a practical and easy-to-use tool that can be a natural part of AML development teams' standard practises.
2. Introduce AUnit and related practises to AML development teams
  - a. Study best practises for software testing, including Continuous Integration and Test-driven development.
  - b. Create supportive and introductory material to aid the adoption of testing practises related to AUnit. For example a "get started" manual or an introductory training for a development team. Justify the selected approach.
  - c. Pilot AUnit in a real life environment

The thesis should be written as a research report with summary both in English and Norwegian, conclusion, literature references, table of contents, etc. During preparation of the text, the candidate should make efforts to create a well arranged and well written report. To ease the evaluation of the thesis, it is important to cross-reference text, tables and figures. For evaluation of the work a thorough discussion of results is appreciated.

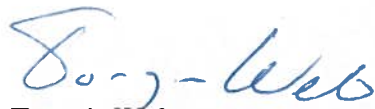
Three weeks after start of the thesis work, an A3 sheet illustrating the work is to be handed in. A template for this presentation is available on the IPM's web site under the menu "Undervisning". This sheet should be updated when the Master's thesis is submitted.

The thesis shall be submitted as two paper versions. One electronic version is also requested on a CD or a DVD, as a pdf-file.

The contact persons are:

From IPM: Qazi Sohail Ahmad  
Ole Ivar Sivertsen

From industry: Geir Iversen (Aker Solutions)  
[geir.iversen@akersolutions.com](mailto:geir.iversen@akersolutions.com)



Torgeir Welo  
Head of Division



Ole Ivar Sivertsen  
Professor/Supervisor



Institutt for produktutvikling  
og materialer

# Abstract

The report is written to show as to what extent test-driven development (TDD) and continuous integration (CI) can be used on KBE models and how a unit testing framework for KBE models can be developed.

Test-driven development (TDD) and continuous integration (CI) has changed the way software is tested. Software testing was often a separate process at the end of a project. It is now being worked on during the entire development period. TDD and CI relies on unit tests. Unit tests are done by dividing the code into the smallest possible units and testing each of them independently. This master's thesis asks how these practices can be used for testing knowledge based engineering (KBE) models.

A unit testing framework for the Adaptive Modeling Language (AML), AUnit, has been developed. It is explained in detail and an introductory guide to using AUnit for testing KBE models in AML is included. AUnit was used to perform TDD and CI on different KBE models, both creating new models and testing existing ones.

Testing KBE models differ to a large degree from testing regular object-oriented software. Different approaches for unit testing and TDD has been performed on several KBE models. It was concluded that the basic attributes in KBE models cannot be unit tested in a sensible way. This includes adding any superclasses and simple parameters like height and width. Without including these attributes, unit testing cannot fully be performed on KBE models using AUnit. However, the models can highly benefit from having unit tests for the logic in the model, which is where the most severe bugs will be. When the attributes are implemented in the model, test-driven development (TDD) can be performed on the models.

Automatic continuous integration (CI) has been performed on a KBE model and the basic principles of CI have been accounted for. CI for KBE models does not differ much from other software projects, so its focus is reduced.



# Sammendrag

Denne rapporten er skrevet for å vise hvordan test-drevet utvikling (test-driven development, TDD) og kontinuerlig integrasjon (continuous integration, CI) kan brukes for å teste kunnskapsbaserte engineering modeller (knowledge based engineering models, KBE) og hvordan et enhetstestingsrammeverk for KBE modeller kan utvikles.

Test-drevet utvikling (TDD) og kontinuerlig integrasjon (CI) er nye måter å teste programvare på. Testing av programvare var tidligere en separat prosess på slutten av prosjektet. Nå blir det jobbet med under hele utviklingsperioden. TDD og CI er avhengig av enhetstester, enhetstester lages ved å dele koden opp i små enheter og teste de hver for seg. Denne masteroppgaven ønsker å finne ut hvordan TDD og CI kan brukes for å teste kunnskapsbaserte engineering modeller (KBE modeller).

AUnit er et enhetstestingsrammeverk for Adaptive Modeling Language (AML) som har blitt utviklet av forfatteren. Det er beskrevet i detalj og en introduksjonsguide som forklarer hvordan man kan teste KBE modeller er inkludert. AUnit ble brukt til å utføre TDD og CI på forskjellige KBE modeller, både nye modeller og eksisterende.

Å teste KBE modeller er i stor grad forskjellig fra å teste vanlig, objekt-orientert, programvare. Forskjellige fremgangsmåter har blitt utført på flere KBE modeller. Det har blitt konkludert med at man ikke kan enhetsteste de mest grunnleggende attributtene i KBE modeller på en fornuftig måte. Dette inkluderer superklasser modellen arver fra og enkle parametere som for eksempel høyde og bredde. Selv om KBE modeller ikke kan fullstendig enhetstestes med AUnit, kan de dra stor nytte av å ha enhetstester for logikken i modellen, som er der de mest alvorlige feilene inntreffer. Når disse attributtene er på plass i modellen kan man utføre test-drevet utvikling (TDD).

Automatiserte kontinuerlig integrasjonstester (CI) har blitt utført på en KBE modell og prinsippene bak CI er blitt redegjort for. CI for KBE modeller fungerer ganske likt som for andre software systemer, så fokuset på CI i denne oppgaven er redusert.





# Contents

<b>Abstract</b>	<b>iii</b>
<b>Sammendrag</b>	<b>v</b>
<b>Preface</b>	<b>ix</b>
<b>Acronyms</b>	<b>xi</b>
<b>Glossary</b>	<b>xiii</b>
<b>List of Figures</b>	<b>xvii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Motivation . . . . .	2
1.2 Research questions . . . . .	3
1.3 Related work . . . . .	4
1.4 Structure . . . . .	4
<b>2 Test-Driven Development</b>	<b>5</b>
2.1 TDD Principles . . . . .	6
2.2 Usefulness of TDD . . . . .	7
<b>3 Continuous Integration</b>	<b>9</b>
3.1 Continuous Integration build process . . . . .	10
3.2 Continuous integration in practice . . . . .	13
3.3 Developing KBE-models . . . . .	14
<b>4 Getting started with AUnit</b>	<b>17</b>
4.1 Writing a unit test . . . . .	17
4.2 Working the AUnit GUI . . . . .	18
4.3 Using AUnit from the command line . . . . .	19
4.4 Other AUnit uses . . . . .	20

<b>5</b>	<b>Unit testing KBE models</b>	<b>21</b>
5.1	Challenges . . . . .	21
5.2	Testing approaches . . . . .	23
5.3	Testing examples . . . . .	27
5.4	Testing at Aker Solutions KBeDesign . . . . .	45
<b>6</b>	<b>AUnit</b>	<b>47</b>
6.1	Overall structure . . . . .	48
6.2	Core . . . . .	48
6.3	GUI . . . . .	49
6.4	Print . . . . .	53
<b>7</b>	<b>Results and discussion</b>	<b>55</b>
7.1	Unit testing KBE models and test-driven development . . . . .	55
7.2	Unit testing at Aker Solutions KBeDesign . . . . .	59
7.3	Continuous integration . . . . .	59
7.4	AUnit . . . . .	60
<b>8</b>	<b>Conclusion</b>	<b>63</b>
8.1	Further work . . . . .	64
	<b>Bibliography</b>	<b>65</b>
	<b>A Introduction to AML</b>	<b>69</b>
	<b>B Software testing</b>	<b>75</b>
	<b>C AUnit Reference</b>	<b>79</b>
	<b>D TDD approaches</b>	<b>83</b>
	<b>E AUnit source code</b>	<b>91</b>
	<b>F Bottle KBE model</b>	<b>135</b>
	<b>G Beam KBE model</b>	<b>139</b>
	<b>H Bookshelf KBE model</b>	<b>143</b>
	<b>I Luva Spar test code</b>	<b>163</b>

# Preface

This thesis marks the completion of my Master of Science degree in Engineering and ICT from the Department of Engineering Design and Material (IPM) at Norwegian University of Science and Technology (NTNU). It is a continuation of the work I did on my project report [7] in 2011.

The report is written to show as to what extent test-driven development and continuous integration can be used on KBE models and how a unit testing framework for KBE models can be developed.

I would like to thank Professor Ole Ivar Sivertsen and Qazi Sohail Ahmad from IPM for great help and inspiration. I would like to thank all the team members at Aker Solutions KBeDesign, especially Geir Iversen, for all the help and support. I would also like to thank TechnoSoft Inc. (TSI) for granting me an AML development license for this work.



# Acronyms

**AI** Artificial Intelligence.

**AML** Adaptive Modeling Language.

**CAD** computer aided design.

**CI** continuous integration.

**CLI** command line interface.

**GUI** graphical user interface.

**JVM** Java Virtual Machine.

**KBE** knowledge based engineering.

**SWEBOK** Software Engineering Body of Knowledge.

**TDD** test-driven development.

**TSI** TechnoSoft Inc..

**VCS** version control system.



# Glossary

## **AUnit**

AUnit is a unit testing framework for testing KBE models written in AML. Unit testing frameworks are usually referred to as xUnit systems. A convention for language specific systems is to replace the x with the first letter in the language, so there is JUnit for Java and NUnit for .NET. For AML the testing framework is therefore named AUnit.

## **Knowledge Based Engineering**

### **AML**

The Adaptive Modeling Language [57] is a programming language developed by TechnoSoft Inc. (TSI) that is used to create KBE models. An introduction to AML can be found in appendix A.

### **KBeDesign**

KBeDesign [29] is a department under the Aker Engineering & Technology [2] subdivision of Aker Solutions which creates KBE product models which are used internally in many of Aker Solutions' projects.

### **Technosoft Inc.**

TechnoSoft Inc. (TSI) [58] is the software company that develops and sells AML.

## **Engineering**

### **Computer aided design (CAD)**

Computer aided design is the use of computers to aid in the development, change and analysis of a design [41].

# Computer Science

## Data type

A data type is set to describe what kind of data a variable contains. For example if the data type is set to an integer the variable holds whole numbers or the boolean data type is used for true/false values.

## Dynamically typed languages

Dynamically typed languages like AML does not set the data type directly in the code. Instead, the program is allowed to change data types after it is compiled.

## eXtreme Programming (XP)

Extreme programming is a software methodology that focuses on short development cycles and frequent releases. This results in responsive software projects where it is easier to adapt to new requirements [9].

## Lisp

Lisp's name is derived from LISt Processing. Lists are one of the fundamental data structures in Lisp. It was developed by John McCarthy at MIT in 1958 [64].

## Common Lisp

Common Lisp is an effort to standardize the different Lisp implementations or dialects. It was published as an ANSI standard in 1994 [62].

## Java

Java is an object-oriented programming language originally developed at Sun Microsystems in 1995. It is mostly inspired by C and C++, but with less low-level functionality and a simpler structure. Java applications are platform independent as they run on a Java Virtual Machine (JVM), not directly in the operating system [63].

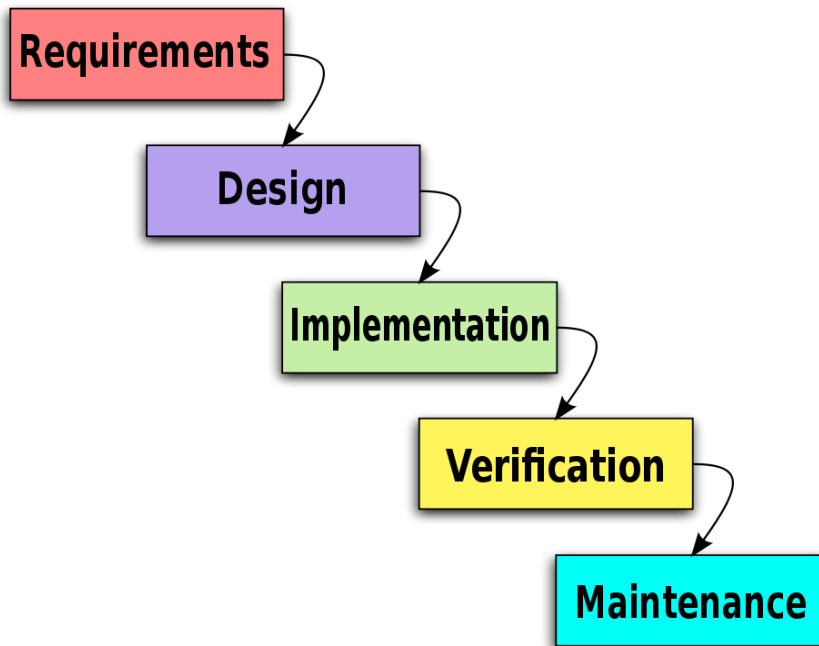
## Refactoring

Refactoring is the process of changing and restructuring a portion of existing code without changing how it behaves for the user or other parts of the program. The system should be functional after each refactoring step, which should be as small as possible. [22].

## Superclass

A superclass is a class other classes are inheriting from, it can also be called a parent class. All KBE models must have at least one superclass, the default, top-level, superclass is `object`.





**Figure 1:** The waterfall model, from [53].

### Waterfall model

The waterfall model is a sequential design process that dates back to 1956 [26]. It describes how a software process goes through a set of stages, from requirements to analysis and program design to coding and testing [48]. The waterfall model has been subjected to substantial criticism [34, 46], saying that software projects need to redo all the steps described in the waterfall model numerous times during the course of the project. The waterfall model is slowly replaced by more agile and iterative design processes.

# **Petroleum industry**

## **Access platforms**

Access platforms are used to get access to valves, monitor gauges or replace parts [31, p. 7]. They have a strict design based on rules from NORSOK, NS and DNV [32, p. 68] as well as internal company-specific standards [32, p. 12]. The placement of the valves and gauges can change countless times during a project.

## **Aker Solutions**

Aker Solutions [3] is a Norwegian oil services company. It works in sections like construction, engineering, maintenance and modifications for different oil fields all over the world.

## **DNV**

Det Norske Veritas is an independent foundation that does classification, risk management and technical advisory for the oil and gas industry [16].

## **NORSOK**

NORSOK, Norsk sokkels konkurranseposisjon or Norwegian Offshore Competitive Position in English is creating standards for the Norwegian oil and gas industry. The standards are managed by Norwegian Standard [44].

## **NS**

Norsk Standard or Norwegian Standard in English is responsible for most of the standardization in Norway [43].

## **Spar platform**

A spar platform is resting on a single large cylinder [42] and is used in deep waters [25].

# List of Figures

1	The waterfall model, from [53]. . . . .	xv
2.1	Test-driven development cycle . . . . .	5
2.2	Simplified test-driven development cycle, red-green-refactor . . . . .	6
2.3	Stress positive feedback loop. Inspired by figure 25.1 in [8, p. 124] . . . . .	7
3.1	Continuous integration, inspired by figure 1-1 in [17, p. 5]. . . . .	10
3.2	Comic, programmers waiting for the code to compile [38]. . . . .	12
3.3	Lava-lamps during a successful build, from [13]. . . . .	13
3.4	FinalBuilder successfully testing the beam model. . . . .	14
3.5	FinalBuilder running a failing test for the beam model. . . . .	15
4.1	AUnit GUI screenshot . . . . .	19
5.1	Missile model with spherical and conical nose . . . . .	22
5.2	Bottle model drawn in AML . . . . .	27
5.3	Illustration of the bottle showing the relationship between the body and the end cap's height and placement. . . . .	29
5.4	Front view of a beam with vertical studs and floorboards on top. . . . .	31
5.5	Front view of a beam with studs. . . . .	32
5.6	Beam with a width of 5, space of 1 and support beam width of 1 . . . . .	35
5.7	Beam with a width of 6, space of $\frac{2}{3}$ and stud width of 1 . . . . .	36
5.8	Beam with a width of 5, showing the x-coordinates. . . . .	38
5.9	Beam with a width of 6, showing the x-coordinates. . . . .	41

5.10	A bookshelf model. . . . .	44
5.11	A broken bookshelf model, a shelf is inserted into the top part of the frame. . . . .	45
6.1	Relationship between the core, GUI and print modules. . . . .	47
6.2	AUnit test results tree structure . . . . .	48
6.3	UML sequence diagram showing how a test is execute inside the core module. . . . .	49
6.4	AUnit object tree . . . . .	50
6.5	AUnit GUI running the beam tests . . . . .	51
6.6	UML sequence diagram showing a user starting AUnit and running a test. . . . .	52
6.7	AUnit GUI test in .NET . . . . .	52
6.8	AUnit GUI test in AML . . . . .	54
7.1	Process for unit testing KBE models . . . . .	59
7.2	A notification pop-up window in AML. It is used when input parameters are wrong. . . . .	60
A.1	AML S-expression evaluation of $(* (+ 1 4) (+ 4 5))$ . . . . .	70
A.2	Bookshelf model structure . . . . .	72

# Chapter 1

## Introduction

KBE models make it possible for engineers to create intelligent product models based upon rules and experience. A KBE model can scale a product based on size or load in a matter of minutes by updating the internal structure of the model. For example, it will add internal beams to sustain a higher load capacity. This can save many man-hours in engineering projects. During the course of ordinary product development projects, components are often re-sized and adjusted as new features are needed. Using KBE models, the components can be modified in minutes instead of being created from scratch every time there is a change [36].

Examples of KBE models are access platforms and stair towers. They have a strict design based on rules from Norsok, NS and DNV [31, p. 68] as well as internal company-specific standards [31, p. 12]. Additionally, the structure of these components are often changed during a project to accommodate adjustments in the platform layout.

AML is a KBE system developed by TechnoSoft Inc. (TSI), initially based on Common LISP. For readers not familiar with AML it is recommended to read appendix A which gives an introduction to AML.

Unit tests are one of the forms of software testing a program goes through. Unit tests are made by finding the smallest possible units of code that can be tested separately [1, Sec. 2.1.1] [11, p. 137]. In addition there is integration testing which is done to make sure that the different parts of the program interact with each other as expected. Finally there is a system test which checks the functionality of the complete, finished, program. Different testing techniques are discussed in appendix B.

AML does not have any features for unit testing. The problem text calls for a framework that is tightly integrated with AML development teams' standard

practices. A fully functional unit testing framework (AUnit), which can be used on KBE models created with TSI's Adaptive Modeling Language (AML), will be created. The foundation of AUnit was created during the author's project report [7].

In order to create an easy to use testing framework, AUnit will be expanded with a graphical user interface (GUI) that can take in a test suite<sup>1</sup> and display the results. Also AUnit's core functionality will be improved and updated to work optimally with both the GUI and the command line interface (CLI).

AUnit has been tested in an AML development team, Aker Solutions' KBeDesign department. These experiences are used to further improve the framework and create introductory material for AUnit. The introductory material found in chapter 4 is meant for KBE developers wishing to incorporate unit tests into their daily routine. An important factor is looking at how unit tests can be written for KBE models, and where it differs from regular software development.

The report uses terms from several different fields with many different terms, so the reader is encouraged to use the glossary found at page xiii actively.

## 1.1 Motivation

Having a unit testing framework available for AML makes it possible to test the KBE models created with AML as well as other AML programs.

All software is tested, both during its production and before the finished product is released. Tests can be run on all or parts of a program and verify that it works as expected. Every time something new is added or updated in the project, the program is tested again. Without unit testing, these tests are usually based on manually executing parts of the code and verifying that no errors are thrown and that the output is as expected.

Unit tests are a formalization of the tests all programmers do when developing software, by writing the tests down in a way that the computer understands. The advantage with this formalization process is that the tests can be run with little effort again and again. Often when developing complex applications, changes in one part of the program can have unexpected effects on seemingly unrelated parts elsewhere in the program. With manual tests the developer might not think to test these unrelated parts. The error will be discovered later (maybe even after the program is released) and will be harder to fix since the programmer is not sure

---

<sup>1</sup>A set of tests are referred to as a test suite.

what caused the error. With unit tests all the tests can easily be run after every change in the code, checking that all parts of the code always work as expected.

AUnit also allows developers to do test-driven development (TDD). TDD is a software development method where the unit tests are written before the code. TDD is a way of programming that ensures that unit tests will always be available for every part of the code, because the code is written based on the tests. It has also been showed that TDD leads to better code that is easier to maintain [37, p. 4].

Continuous integration (CI) is another software development method that relies on unit testing. The principle of CI is that software should be automatically fully tested and integrated into a real-life environment after every change in the code. CI leads to faster integration time and less time spent on fixing difficult bugs, because the bugs are discovered earlier before they get too difficult to fix.

Having unit tests in place can combat some of the unwanted problems that arise because of the localization differences between different teams. For example, Aker Solutions KBeDesign is in Norway and TSI in North America. A software update from TSI can cause unexpected changes to Aker Solutions' product models. With a set of unit tests and a continuous integration environment, TSI can detect any conflicts themselves and make the changes before releasing the software update into Aker Solutions' production environment.

## 1.2 Research questions

The research questions defined below will give the background for the work done in this report.

**RQ1:** How can unit tests for AML KBE models be written in a concise manner using known input and output parameters?

**RQ2:** How can test-driven development be used for developing KBE models in AML?

**RQ3:** How can continuous integration be used in an environment developing KBE models with AML?

## 1.3 Related work

As to the author's knowledge, there has not been any research or development done on unit testing KBE models. However, this report is based on reasearch from two existing fields, software testing and knowledge based engineering.

The first software testing principles are from *The Art of Software Testing* [39] by Myers et al. and *Software testing techniques* [10] by Beizer. More than twenty years later, Beck developed unit testing and the first unit testing framework, JUnit. He later laid out the principles for test-driven development in *Test-driven development: by example* [8]. Continuous integration for KBE models is based on Fowler's articles [20, 21, 22, 23].

Several papers on KBE technology has been published in the last ten years, mainly from Stokes [55], Cooper and Rocca [14] and Milton [36]. In 2012 Rocca published an article [47] describing KBE, i.e. how it differs from computer aided design (CAD) systems and some of the challenges the technology is facing.

## 1.4 Structure

Chapter 2 and 3 describes how test-driven development and continuous integration works. How the unit testing framework, AUnit, is used to write unit tests for AML-based KBE models is explained in chapter 4. Then in chapter 5 AUnit is used for doing TDD on several KBE models, including models used in real life environments. Chapter 6 gives a detailed look at the inner workings of AUnit. The results are discussed in chapter 7.

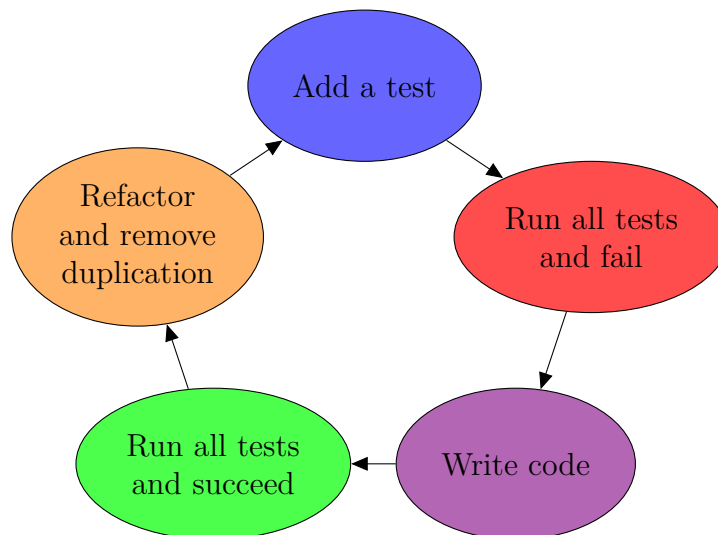


# Chapter 2

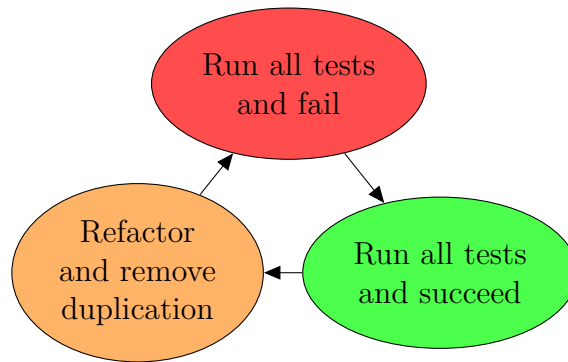
## Test-Driven Development

Test-driven development is a coding style created by Kent Beck [8], one of the main contributors to the JUnit unit testing framework for Java and eXtreme Programming (XP).

This chapter will look at the principles behind TDD and its usefulness in software projects. Then follows a discussion on how TDD can be used for developing KBE models.



**Figure 2.1:** Test-driven development cycle



**Figure 2.2:** Simplified test-driven development cycle, red-green-refactor

## 2.1 TDD Principles

The process of test-driven development is to write unit tests before the programmer writes any code. After the test is written the goal is to make it succeed<sup>1</sup>. After the test has succeeded the programmer refactors the code to remove any duplication inside the code and between the code and the test. New code should only be written to refactor<sup>2</sup> the existing code or to make a test pass. One should never write a new test if another test is already failing.

A simpler way to look at the test-driven development cycle is “red/green/refactor” (see figure 2.2). Kent Beck [8, p. x] refers to this as the TDD mantra:

### Red

Write a test before writing new code. The test will fail and be “red”.

### Green

Make the test succeed, turn green, taking as many shortcuts as necessary.

### Refactor

Remove any duplication in the code necessary to make the test go green.

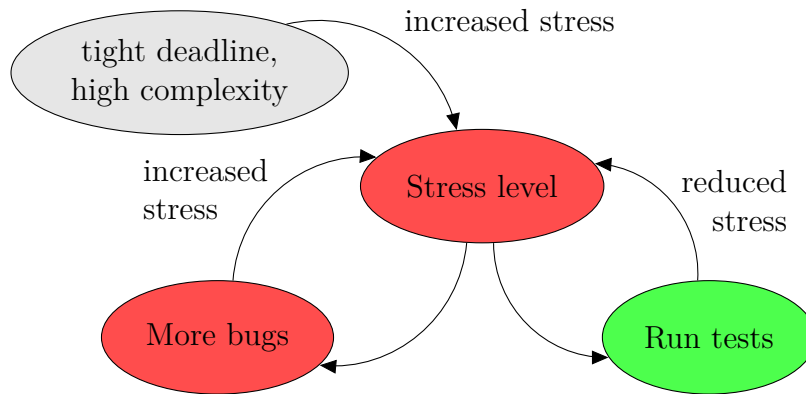
### 2.1.1 Duplication

After the creation of a test it is allowed, or even encouraged, to take shortcuts. For example by the program returning the expected values directly, causing temporary

---

<sup>1</sup>Making tests succeed is often referred to as “go green”. This comes from the status bar in JUnit that turns from red to green when all tests succeed.

<sup>2</sup>See the glossary on page xiii.



**Figure 2.3:** Stress positive feedback loop. Inspired by figure 25.1 in [8, p. 124]

duplication. A finished piece of code, however, must not contain any information that exists in the unit test, but be refactored until all the duplication is removed.

## 2.2 Usefulness of TDD

With TDD the programmer is forced to think in detail about how the unit being developed will act, what it will take as input and what will be the output. Research [24] has shown that this leads to code that is better structured and easier to expand and refactor.

Another important point is that when the stress level rises as the project comes closer to the deadline, the developer will not have time to do as much testing as needed and bugs will appear. The developer will become even more stressed which results in a negative feedback loop, as seen on the left side in figure 2.3. On the other hand, if the project uses TDD the tests can easily be run as the stress level rises. Developers can check all the modules for errors when a single module has been changed. A stressed programmer can easily overlook a conflict that arises somewhere else in the code. This way the unit tests will work to reduce the stress [8, p. 124].

Several research projects are conducted on the efficiency and quality of software done with TDD compared to the traditional waterfall model. George and Williams [24] did a test with 24 professional programmers that they split into two groups developing the same Java application, one using TDD and the other a waterfall-like<sup>3</sup> approach creating the tests after the code. They found that the

<sup>3</sup>See the glossary on page xiii for a definition of the waterfall model.

TDD projects passed 18 percent more functional black box tests, but used 16 percent more development time.

There are a few validity issues with George and Williams's research. With only 24 programmers working in pairs of two, it leaves only 6 pairs doing each programming style. There were also great variations between the pairs, the time to complete the exercise was varying from 200 to over 400 minutes and between 12 and 20 (out of a total 20) test cases passed. The programmers working with TDD ended up with a set of working unit tests while the waterfall group had some unit tests, but not full coverage, even though they were encouraged to write unit tests. This makes it unfair to compare the time differences between the two groups since the TDD programmers created more useful code. George and Williams also has a hypothesis that code written in TDD is easier to maintain and have better design than using traditional software development methods [24, p. 1137].

Other research backs up George and Williams' hypothesis. Müller and Padberg have established that using TDD leads to faster error fixing [37, p. 4]. Given that 85 to 90 percent of software development budgets go to maintaining existing code [18, p. 17], code quality and maintainability has a clear advantage over faster development time.

Also, Maximilien and Williams [33] finds TDD highly beneficial. While developing a complex software system at IBM they experienced a 50 percent drop in error defects compared to a similar system developed earlier that wrote unit tests after implementing the code [33, p. 6]. There are several uncertainties with this comparison. Even though the two software projects are similar, they are not equal. They had different requirements and people, also the newest project may have gained experience from the first.

# Chapter 3

## Continuous Integration

The first part of this chapter will give an introduction to CI and an explanation of how it works. Section 3.2 will discuss the usefulness of CI for KBE-projects. A brief account of how CI works for KBE projects will be given in section 3.3.

Continuous integration is the process of automating the building and testing process in software development. It originates from an article Fowler and Foemmel wrote in 2000 [23] that Fowler revised in 2006 [21].

“If it hurts, do it more often.”

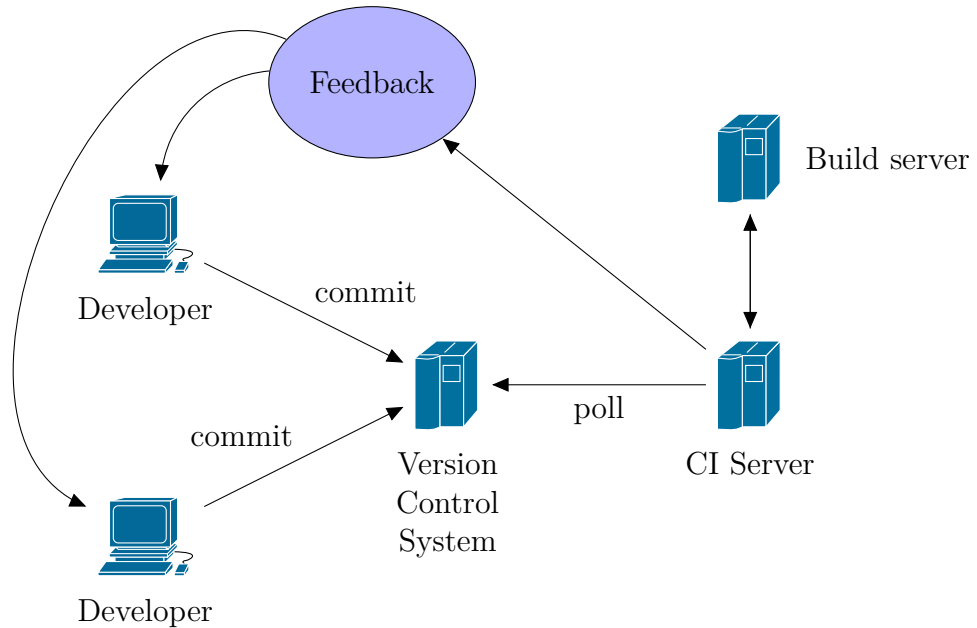
Fowler, *Frequency Reduces Difficulty* [20]

Continuous integration was conceived to avoid the indeterminately long integration processes common in large software projects. Integration is among the last phases in a software development project where all the different parts of the software are joined together and put under integration tests to verify that they can interact with each other as planned [21].

Fowler’s way of avoiding these long and expensive integrations is to integrate very often<sup>1</sup>. When a programmer has done some work on a project and is satisfied with the result, he builds it on his computer. Then the code is committed into a common code repository where it is checked out into a real-life test environment. Here unit and integration tests are performed on the software. If these tests succeed, the developer can continue his work, if something fails he must fix it immediately.

---

<sup>1</sup>Continuous integration is not technically correct since it does not mean to integrate all the time (continuously) only often [17, p. xxii].



**Figure 3.1:** Continuous integration, inspired by figure 1-1 in [17, p. 5].

CI can either be done manually, with custom scripts or a dedicated Continuous Integration server. A CI server automatically looks for new additions to the code repository, builds the code and sends the test results to the developer via email.

When the frequency of difficult tasks like integration is decreased, each time will be less painful. During the course of long software projects, specifications change numerous times and parts of the software has often evolved in different directions. The cost of making fundamental architectural changes is much larger than in the beginning of the project. If the integration process is done sooner these changes can be done as early as possible. Additionally, when a process like integration is done more often, the development team will become better at integration and be able to learn from past mistakes [20]. Having short iterations, adjusting the course as the project advances and learning from previous mistakes are some of the cornerstones of extreme programming [9].

### 3.1 Continuous Integration build process

The following section will discuss the practices of continuous integration from Fowler's article [21].

### 3.1.1 Code repository

A code repository is maintained by using a version control system where each developer can commit code into the project, revert to an earlier stage or merge conflicting changes. For CI to work the repository needs to be used actively by the developers - committing after every change in the software.

The code repository should contain everything the build machine needs to build the software.

There are many version control systems (VCSs) to choose from, and most of them are free. Going into a thorough discussion of VCSs is outside the scope of this report, but I will mention three of the most popular systems. Concurrent Versions System (CVS) [15] is one of the first mainstream versioning systems; being developed in 1990 the software has aged and is generally replaced by Apache Subversion (SVN) [5] which was created as a successor to CVS in 2000. Five years after the initial development on SVN, Linus Torvalds developed Git [50] for aiding the development of the Linux kernel. Git focuses on being fast with highly reliable branching and merging capabilities. This eases the process of keeping a main branch that contains code that it always working while a developer can do experiments or bug fixes for previous releases on different branches.

It is important to note that the team must decide on a common coding standard; indentations and code comments as well as the structure of the code. Having this in place is vital for being able to have code that can easily be read and modified by other team members.

### 3.1.2 Automated build

The entire build process should be automated to a simple process that does not require user interaction. There are several tools available for creating build scripts. For Java, Maven [6] and Ant [4] are often used, for .NET Nant [40] and MS-Build [35] are available. There are also language independent tools<sup>2</sup> available, like FinalBuilder [19], which can build and test software from almost any source.

### 3.1.3 Testing the build

The build should be self-testing using a set of unit tests. Unit tests have already been discussed in chapter 2 and 5.

---

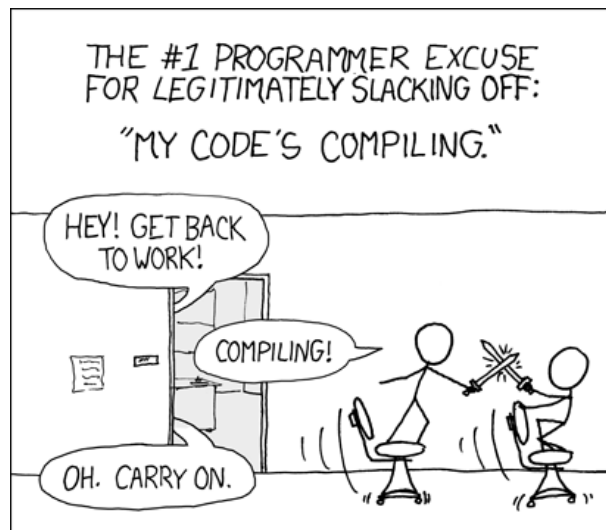
<sup>2</sup>Some building tools, like Ant, can also be used for different platforms using plugins.

### 3.1.4 Code commits

Each developer should commit changes to the main repository at least once a day. Every time a commit is made, a build should be checked out into the integration environment and go through all the tests. The integration environment should resemble the production environment as closely as possible.

### 3.1.5 Build time

For a continuous integration process to be effective, the build must be fully automated and not be too time consuming. A CI build should never last more than 10 minutes according to the extreme programming guidelines. If it does, the tests should be optimized until they take less than ten minutes [9].



**Figure 3.2:** Comic, programmers waiting for the code to compile [38].

This is because the programmer should still have the changes made to the code fresh in his mind in case the build fails. If the programmer has started working on a new task, it will be harder to look back at the last problem and find the bug. With long build times, the best solution might be what we see in figure 3.2 avoiding that the developer becomes too occupied with a new problem to fix the bug.

A way to decrease the build time is to use smart build tools that analyze what has changed and needs to be rebuilt and leaves the unmodified code alone. AML has this inbuilt in the compiler so that *compile-system* will only build the files that





**Figure 3.3:** Lava-lamps during a successful build, from [13].

has changed since the last build by default [60]. However, tests have to be run to account for unexpected domino effects when changing the code.

### 3.1.6 Feedback

Finally, it is important that the entire team can get feedback from the integration tests when they want it. Some use lights or lava-lamps showing if the build is currently integration correctly or not, see figure 3.3. In addition websites can give deeper insight in where the problem lies and show statistics of how well the build is working over time. E-mail notifications are also a nice way to be notified of a builds success, but they should be targeted to the developer(s) that sent the build, not the entire team.

## 3.2 Continuous integration in practice

There are several research papers that have good experiences after implementing continuous integration. In [56], Stolberg went from a waterfall testing process to a completely integrated testing environment in about a year. He found it to be faster than the waterfall method [56, p. 373] after the initial transition period was over. Karlesky et al. shows in [28] that continuous integration can work well for embedded systems.

Description	Status
Start aml Left Click [ -> * / ]	Completed
Wait For Process [ xemacs.exe ]	Completed
Delay 1 second	Completed
Run aml - Left Click [ -> * / ]	Completed
Run AML GUI - Left Click [ -> * / ]	Completed
Confirm GUI [ -> * / ]	Completed
Select text field - Left Click [ -> * / ]	Completed
Type Text [ (load-system :aunit) ] to [ xemacs -> MainWindow ]	Completed
Send Key [ Enter ] to [ xemacs -> MainWindow ]	Completed
Delay 2 seconds	Completed
Type Text [ (load-system :aunit-print) ] to [ xemacs -> MainWindow ]	Completed
Send Key [ Enter ] to [ xemacs -> MainWindow ]	Completed
Delay 2 seconds	Completed
Type Text [ (first-setup) ] to [ xemacs -> MainWindow ]	Completed
Send Key [ Enter ] to [ xemacs -> MainWindow ]	Completed
Define Variable (Deprecated) [Project: Aunit-Results]	Completed
Type Text [ (run-script "D:\workspace_win\src\beam\beam-test.aml") ] to [ ... ]	Completed
Send Key [ Enter ] to [ xemacs -> MainWindow ]	Completed
Wait For File [ D:\workspace_win\src\beam\output\results.html ]	Completed
Check If [ D:\workspace_win\src\beam\output\pass ] Exists	Completed

Figure 3.4: FinalBuilder successfully testing the beam model.

### 3.3 Developing KBE-models

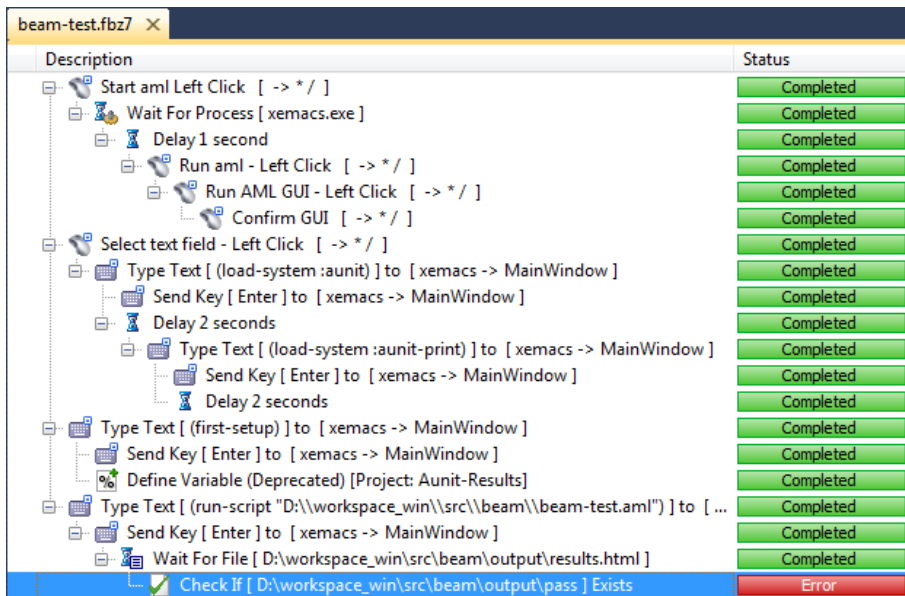
This section will discuss RQ3:

**RQ3:** How can continuous integration be used in an environment developing KBE models with AML?

A continuous integration environment for KBE models using the FinalBuilder [19] software is set up. It was found that CI poses few challenges when used on KBE-models compared to other software projects. Therefore this section is substantially shorter than the previous chapter discussing TDD.

FinalBuilder is CI software that is meant to work for any kind of programming language. It can check out source code from all the common version control system (VCS), execute programs and learn from the users mouse clicks to test and perform GUI operations.

First, FinalBuilder is set up to interact with the local VCS. In this case it is set up with Subversion [5] to check out the latest source code from the repository. Then it is set up to start the *xemacs.exe* process and from there start the AML process. Next it loads the AUnit systems by selecting the command line interface and typing the necessary commands (*load-system :aunit*). The AUnit commands are run in the same way to set up AUnit and run the test scripts. Finally, it waits



**Figure 3.5:** FinalBuilder running a failing test for the beam model.

for AUnit to finish the testing by checking if the output files have changed. When it is finished, it will check for a *pass* file that will be in the output directory if every test has passed. If it doesn't find the *pass* file, FinalBuilder will know that some tests have failed.

FinalBuilder cannot read directly from AMLs command line so in order to make sure that the loading of the systems is finished a wait time of 2 seconds is added between the commands so that AML has time to respond.



# Chapter 4

## Getting started with AUnit

This chapter will give a step by step guide to working with AUnit. It is assumed that the reader has some basic AML knowledge either from the AML introduction in appendix A or the “AML Basic Training Manual” [59]. Appendix C gives more thorough information on all of AUnit’s features.

### 4.1 Writing a unit test

AUnit introduces three basic commands for writing unit tests:

**defaunit** surrounds the entire test script with a single parameter that sets the test name.

**deftest** creates a test method that can contain one or more tests. A special name, *set-up* is reserved for setting up any prerequisites for the tests, this method will run before every deftest method.

**check-\*** methods are used to do tests. The most common, *check-equals*, is used to compare two parameters. See appendix C.1 for a list of all the check statements available.

```

1 (defaunit "Sample_test "
2   (deftest "set-up"
3     (print "setting_up_test")
4   )
5
6   (deftest 'test-int-equal
7     (check-equals (+ 2 2) 4)
8   )
9
10  (deftest 'test-int-float-equal
11    (check-equals 2 2.0)
12  )
13
14  (deftest 'test-string-equal
15    (check-equals "hi" "hi")
16  )
17 )

```

**Listing 4.1:** Sample test code for AUnit

On line 1 there is a `defaunit` statement enclosing the entire test and naming it `Sample test`. Following that there is a `deftest` `set-up` method that will run before each of the tests. In more advanced examples this is normally used for creating a model to test on. In this simple test it is not necessary, so it prints out a string to demonstrate how it works.

The first test is found on line 6. Inside the `deftest` there is a `check-equals` statement that checks if 4 is the same as  $2 + 2$ . Hopefully it is. The tests on line 10 and 14 works similarly, checking if a floating point number can equal an integer and if it can compare two strings. All these tests will pass in AUnit.

## 4.2 Working the AUnit GUI

After the system files are loaded, the AUnit GUI (see figure 4.1) can be started from the AML command line interface with the `(aunit)` command. From the GUI it is possible to either select a `tests.def` file which contains a list of test scripts using the `Load def` button or test scripts can be loaded individually with the `Browse` button.

When tests are loaded, `Run all tests` will execute all the tests loaded into AML, `Run selected test` will only execute the test that is selected. `Save to def file` saves a list of all the loaded tests. When a `.def` file is already loaded it will be used, if not it will save the list to a `tests.def` file in the same folder as the tests.

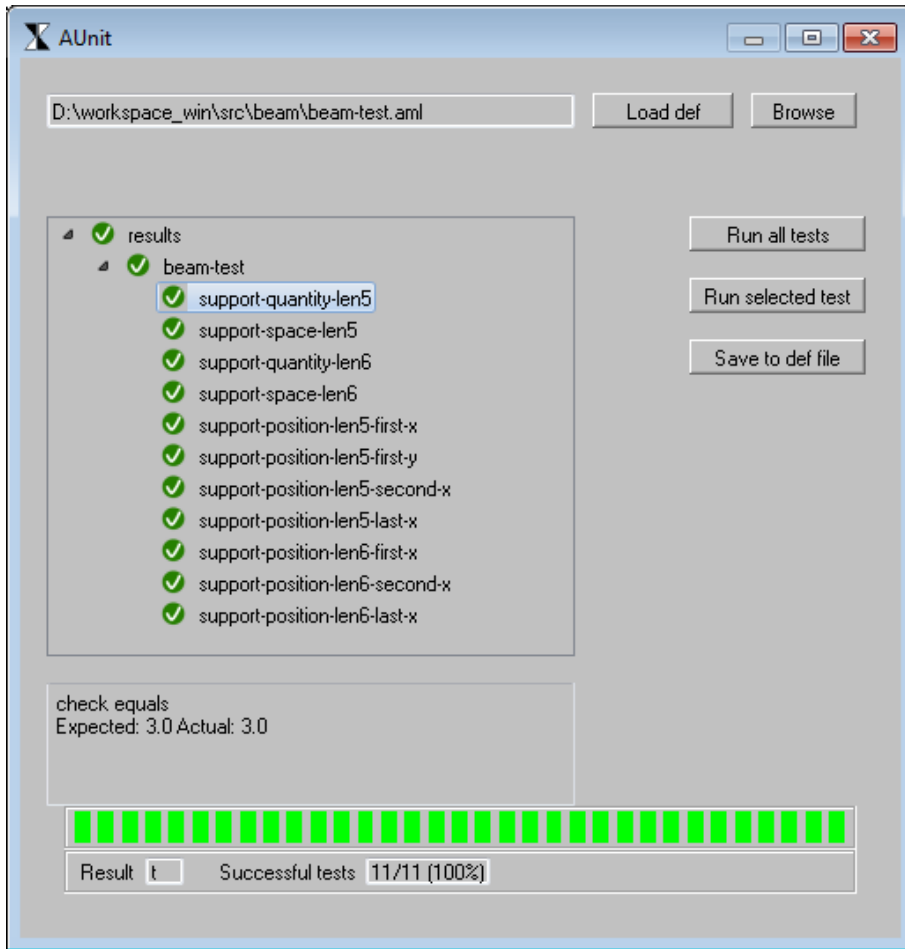


Figure 4.1: AUnit GUI screenshot

After a test has finished the bottom of the AUnit window will change to reflect the results of the tests. When all tests succeed, the bar will go from red to green. The textbox above shows the results of the individual tests inside the test method that can be selected in the tree.

### 4.3 Using AUnit from the command line

AUnit can also be used directly from the command line. This is normally a better option for expert users and when running AUnit from other applications, for example continuous integration servers.

First AUnit has to be prepared with the `(first-setup)` command. Then, a test

can be loaded with:

```
(run-script 'C:\\path\\to\\test\\class-to-test-test.aml')
```

The results are available in the output directory, located next to the test script.

## 4.4 Other AUnit uses

AUnit can be used to create a failing test case, for example when one of TechnoSoft Inc. (TSI)'s customers, like KBeDesign at Aker Solutions, needs TSI to fix a bug or request a new feature in AML. It is often easier to create a failing test than to describe exactly what went wrong when discovering a bug. The customer creates a unit test that will trigger the bug and TSI can use the test while fixing the bug. This way TSI and KBeDesign can easily relate to the same criteria for when a bug is actually fixed.



# Chapter 5

## Unit testing KBE models

As shown in chapter 2, many projects show great results using TDD. It is therefore interesting to know if TDD can be used in KBE projects. This chapter will discuss how to test KBE models written in AML by exercising different approaches. It aims to answer RQ1 and RQ2:

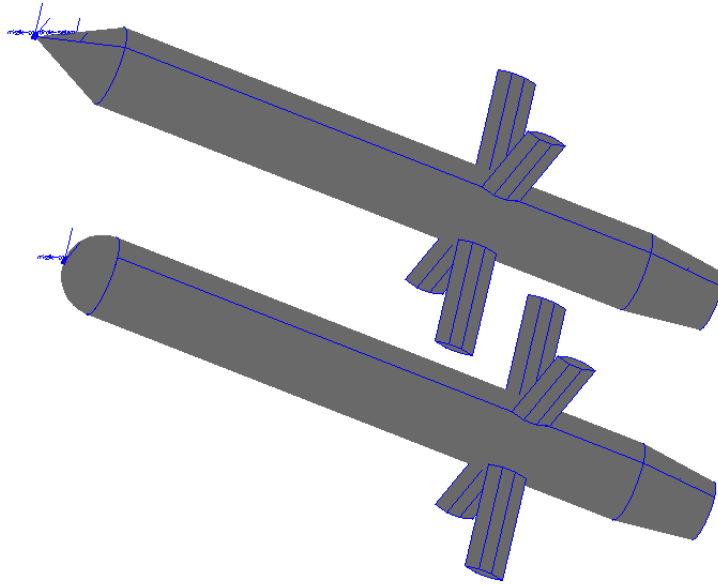
**RQ1:** How can unit tests for AML KBE models be written in a concise manner using known input and output parameters?

**RQ2:** How can test-driven development be used for developing KBE models in AML?

### 5.1 Challenges

Several problems arise when one tries to write unit tests for KBE models. The models are complex and the expected output is usually not known beforehand, it is calculated by AML. In order for unit tests to be easy to write and update, they must compare a set of known or easily calculated parameters.

In regular software development the developer can expect a certain result from the code he writes and formalizes these expectations into a test. A Java example will be used to show this. Listing 5.1 tests a `Car` class that has a `drive` method and a `position` parameter. To test the `drive` method, the developer can check if the position of the `Car` has changed according to what was given as input to the `drive` method.



**Figure 5.1:** Missile model with spherical and conical nose

```
Car car = new Car(); //position is initially set to 0.  
assertEquals(5, car.drive(5).position);
```

**Listing 5.1:** Java code for testing a Car class

AML and other languages based on Lisp are dynamically typed. This means that a property's type<sup>1</sup> is not set at compile time. In the Java example above the `position` variable in the `Car` class is set to the integer data type and if it is attempted to set it to a string (e.g. "five") or a boolean (e.g. `False`) Java will give an error message.

In AML a property can start out as a boolean value (e.g. `nil`<sup>2</sup>) and later change to an integer (e.g. `5`). For KBE-languages like AML the dynamic typing is extended to apply to object types as well as data types [47, p. 166]. This is a useful feature when a model uses different parts depending on other parameters. An example is seen in listing 5.2 where the missile's nose changes based on a keyword given as input to the model (see figure 5.1). It can also be changed automatically, for instance the length of the missile can determine which nose is suitable.

<sup>1</sup>See the glossary on page xiii for a description of data types.

<sup>2</sup>Simply put, `nil` is the boolean value equivalent of `False` in AML and Lisp.

```
(nose :class (case !missile-nose-type
              (sphere 'spherical-nose-class)
              (cone 'open-conical-nose-class)
              (t 'spherical-nose-class))
)
```

**Listing 5.2:** AML code for selecting the missile nose type, from [59]

KBE models can also use an arbitrary number of sub objects that can have different parameters and even be of different classes. This makes the object tree of a KBE model impossible to predict before the model is instantiated as both the number of objects, the type of objects and the depth of the tree is dynamic [47, p. 166]. When testing part by part using the-references<sup>3</sup> in a dynamic tree, the tests will be broken as soon as a parent object is renamed, removed or inserted. It also makes it impossible to reuse the unit tests if any of the classes are used in a different place. This chapter will look at different testing approaches taking these limitations into account.

## 5.2 Testing approaches

### 5.2.1 Testing parameters

**Hyp1:** KBE models can be unit tested by directly checking that each property has the expected value.

The hypothesis will be tested by doing an example checking that the diameter and type of the bottle body is correct. The test can look like this:

---

<sup>3</sup>See appendix A.1.4 for explanation of the-referencing in AML.

```

(defaunit "bottle-test"
  (deftest 'set-up
    (load "D:\\workspace_win\\src\\polygon\\src\\bottle.aml")
    (create-model 'bottle :class 'bottle-par-class)
  )

  (deftest 'body-diameter
    (check-equals
      (the bottle body diameter)
      2.0)
  )

  (deftest 'body-class
    (check-equals
      (type-of (the bottle body))
      'open-cylinder-object)
  )
)

```

**Listing 5.3:** Testing the parameters in a bottle model

```

(define-class bottle-par-class
  :inherit-from(object)
  :properties(
  )
  :subobjects(
    (body :class 'open-cylinder-object
          diameter 2.0
          )
  )
)

```

**Listing 5.4:** AML code for the bottle's body

It was established in chapter 2 that tests should not contain any duplication between the actual model code and the test script. With tests like the one above (listing 5.3) we have the diameter (2.0) and superclass (`open-cylinder-object`) duplicated. They are present in both the test and the model code (listing 5.4).

The duplication has several disadvantages. The main issue is that the tests only check that the superclass and diameter properties are typed correctly in the code. If both are misspelled, for example if the developer used a `cylinder-object` instead of `open-cylinder-object`, the test will not discover the error. Instead, the tests could lead the developer to think that everything is working when it is not. The second issue is that the duplication makes it a more elaborate process to change even minor details in the model. When making a change the exact same change

has to be made twice. Finally, the tests themselves are very time consuming to write compared to the value they add to the KBE model.

The complete example can be found in appendix D.1.

### 5.2.2 Testing geometry

**Hyp2:** KBE models can be unit tested by calculating their volume and surface area and comparing these values to the volume and surface areas of the objects in the KBE model.

The next hypothesis aims to avoid duplication between tests and production code. The tests check that the object's surface area and volumes are correct. With an `open-cylinder-object`, AML's `volume-of-object` will give us the surface area of the object. We know that the surface area for a cylinder is:

$$A = \pi dh \tag{5.1}$$

Equation 5.1 is then implemented in the test script in listing 5.5, checking that the specified superclass and default values return an object with the surface area of a cylinder.

```
(defaunit "polygon-test"
  (deftest 'set-up
    (clear)
    (load "D:\\workspace_win\\src\\polygon\\src\\polygon.aml")
    (create-model 'bottle :class 'bottle-class)
  )

  (deftest 'cylinder-surface
    (check-delta
      (volume-of-object (the bottle body))
      (* pi 2 5)
      0.0001
    )
  )
)
```

**Listing 5.5:** Testing the surface area of a bottle model

This testing process was even more time consuming than the previous hypothesis. The models get more complex as they are developed, they will turn into non-standard geometrical shapes that will take a lot of effort to calculate by hand.

Using this approach the developer will do the calculations AML is doing himself instead of depending on AML's modeling engine. This turns into code that test AML's functionality more than they test the model.

The second hypothesis is tried out in on the whole bottle in appendix D.2.

### 5.2.3 Testing logic

**Hyp3:** KBE models can be tested by checking the return values from just the conditionals, loops and calculations.

Finally, a third approach is performed based on Beck's list of four main types of code that should be tested [8, p. 278]:

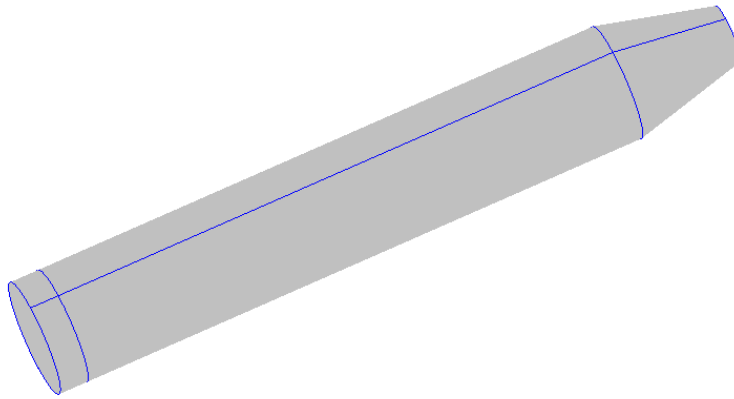
- Conditionals (if/else/case)
- Loops (for/while)
- Operations (calculations)
- Polymorphism<sup>4</sup>

Following these recommendations it is not necessary to directly test all the class instances and subobjects in a KBE model. For example, the positioning of an object relative to a different object can be calculated by hand. The actual position can be compared with the calculated outcome. This results in simple, but effective, unit tests for all calculations in a model. The following section will create tests based on the following hypothesis. A KBE model can be unit tested by checking only conditionals, loops and operations.

This hypothesis will be tried out in different examples. First a simple bottle model that has a few calculations to position its objects. Then follows a more complicated beam model in section 5.3.2 where there are both engineering rules and positioning calculations. In section 5.3.3 it is tested on an existing KBE-model of a bookshelf made by KBE professionals at Aker Solutions KBeDesign. Finally, it is tried on a small part of Luva's spar platform.

---

<sup>4</sup>Polymorphism is the use of overloaded methods. For example the draw model in AML can be called in the same way on both a box and a cylinder object, but it will draw different objects. This can often replace *if* and *case* statements. For example, by creating a draw method in each object's class instead of a large draw class that has a different case statement for each object.



**Figure 5.2:** Bottle model drawn in AML

These models are in many ways simple, and the TDD-process might seem a bit elaborate since the final solution is sometimes clear without going through all the steps. The examples are meant to illustrate the process and how TDD can be done in small steps. When using TDD in other projects, what is being done in each cycle should be adjusted with regards to experience and complexity.

## 5.3 Testing examples

The following section features three examples of testing the logic in KBE models.

### 5.3.1 Testing a bottle model

When creating a bottle as seen in figure 5.2 one option is to start with a body cylinder, as shown in listing 5.6. This can be done without the need for any calculations or other code functions mentioned by Beck in [8], and thus no testing is required. We add an *open-cylinder-object* as a subobject, which uses the diameter and height from the class' properties.

```

(define-class bottle-3-class
  :inherit-from (object)
  :properties (
    diameter (default 2.0)
    body-height (default 5.0)
    bottom-height (default 0.5)
  )
  :subobjects (
    (body :class 'open-cylinder-object
      diameter ~diameter
      height ~body-height
    )
  )
)

```

**Listing 5.6:** AML code for creating the body of a bottle

The model gets more complicated when we add an end cap on the body. This needs to be oriented to the end of the body cylinder. The AML code for the end cap is shown in listing 5.7. The code is added below the *body* subobject in listing 5.6. The height and diameter is found in the properties in listing 5.6.

```

(bottom :class 'cylinder-object
  diameter ~diameter
  height ~bottom-height
)

```

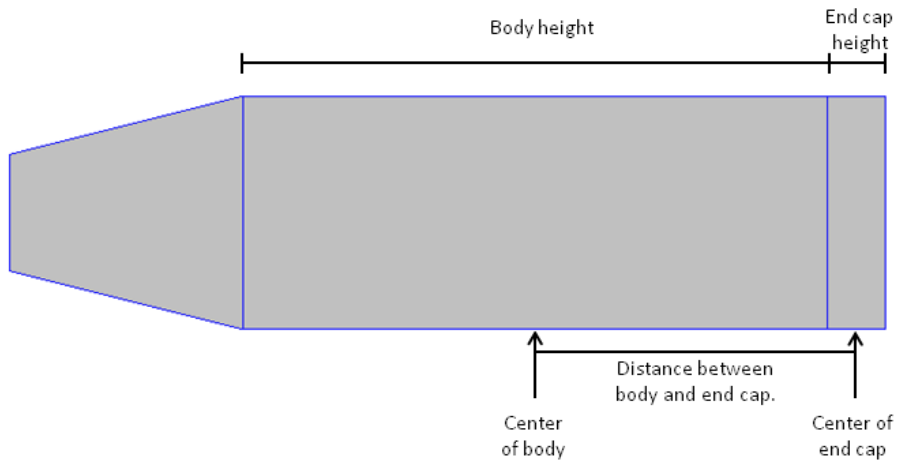
**Listing 5.7:** AML code for creating the end cap of a bottle.

Then the orientation can be calculated using TDD. Starting with the test seen in listing 5.8, it checks that the orientation of the end cap cylinder matches the body cylinder. First, there is a set-up method that is called before every unit test is run. The set-up method in listing 5.8 clears AML's canvas and loads the bottle model.

AML refers to the center of objects when placing them. In relation to the center of the main body, the end cap should be half the body's height plus half the end cap's height below the body's reference point. Each object is placed in the origin by default. The necessary distance between the two object's centers can be calculated with formula 5.2.

$$dist = \frac{h_{body}}{2} + \frac{h_{endcap}}{2} \quad (5.2)$$





**Figure 5.3:** Illustration of the bottle showing the relationship between the body and the end cap's height and placement.

Since the body height is set to 5 and the bottom is set to 0.5, it should be placed  $5/2 + 0.5/2 = 2.75$  below the body, see figure 5.3. This is checked by comparing the global positioning vectors of the body and the end cap.

```
(defaunit "bottle-test"
  (deftest 'setup
    (clear)
    (load "D:\\workspace_win\\src\\polygon\\src\\bottle-3.aml")
  )
  (deftest 'bottom-location-default
    (create-model 'bottle
      :class 'bottle-3-class)
    (check-list-diff
      (convert-coords (the bottle bottom) '(0 0 0)
        :from :local :to :global)
      (convert-coords (the bottle body) '(0 0 0)
        :from :local :to :global)
      (list 0.0 0.0 -2.75))
    )
  )
)
```

**Listing 5.8:** Test for checking the location of the bottle's end cap.

The `convert-coords` seen in listing 5.8 is an AML function that converts the coordinates of the object from local coordinates to global coordinates so that they can be compared between different objects. It returns a list of x, y and z-coordinates.

Using TDD, this test should succeed as quickly as possible. Therefore the z-coordinate is set directly to -2.75 in the orientation parameter, as shown in listing 5.9.

```
(bottom :class 'cylinder-object
  diameter ~diameter
  height ~bottom-height
  orientation (list
              (translate (list 0 0 -2.75)))
)
```

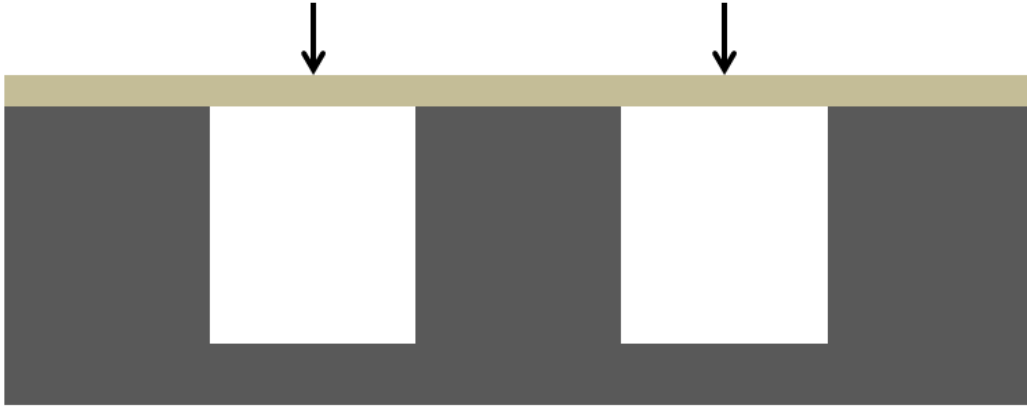
**Listing 5.9:** AML code for preliminary position for the bottle's end cap.

This result in successful tests, but the bottom will not be placed correctly for other height parameters. This calls for an additional test. Setting the heights to 10 and 2, the bottom should be placed at -6 ( $\frac{10}{2} + \frac{2}{2} = 6$ ) from the body. This is tested in listing 5.10.

```
(deftest 'bottom-location
  (create-model 'bottle
    :class 'bottle-3-class
    :init-form (list
              'body-height 10.0
              'bottom-height 2.0))
  (check-list-diff
    (convert-coords (the bottle bottom) '(0 0 0)
                   :from :local :to :global)
    (convert-coords (the bottle body) '(0 0 0)
                   :from :local :to :global)
    (list 0.0 0.0 -6.0))
)
```

**Listing 5.10:** Test for checking the location of the bottle's end cap with a body height of 10 and bottom height of 2.

In order to get this test to succeed it is necessary to use the height values to calculate the end cap's position. Changing the orientation parameter to use the bottle's parameters in equation 5.2 instead of returning a value directly makes both tests succeed.



**Figure 5.4:** Front view of a beam with vertical studs and floorboards on top.

```
orientation (list
  (translate
    (list
      0
      0
      (-(+ (half (the superior superior body height))
          (half ^height)))))
```

**Listing 5.11:** AML code for positioning the bottle's end cap.

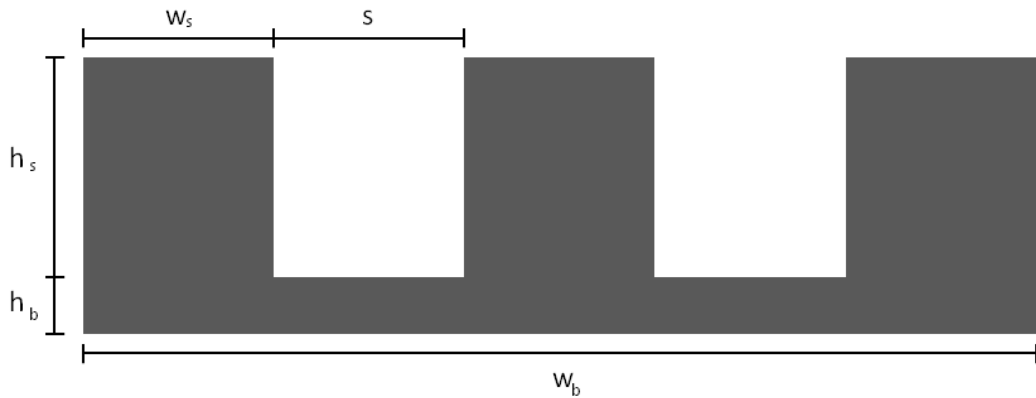
The process for the top of the bottle is similar. The complete source code and test scripts can be found in appendix F.

### 5.3.2 Testing a beam model

In this section a simple beam is applied as an example on how to write unit tests for KBE models made with AML. The complete source and test code can be found in appendix G.

This model will create a beam with studs that might be used to hold up a floor as seen in figure 5.4. The floorboards are dimensioned for a certain weight when they have a maximum distance between each support. The KBE model must find the optimal number of studs and place them correctly. If the space between the studs is too large the floorboards will give in, at the same time it is costly to use more studs than necessary.

The beam has a certain number of studs ( $n_s$ ) on it. It has a varying width,  $w_b$  and between every stud of width  $w_s$  there should be a space not greater than



**Figure 5.5:** Front view of a beam with studs.

$s_{max}$ . There should always be studs at the start and ends of the beam. The space between the studs might be less than  $s_{max}$  so  $s$  is introduced as the actual space between the beams.

The AML code of for the properties in the beam class will be as in listing 5.12. It contains the height ( $h_b$ ), width ( $w_b$ ) and depth ( $d_b$ ) of the beam and the height ( $h_s$ ), width ( $w_s$ ) and depth ( $d_s$ ) of the studs as well as the maximum space between them ( $s_{max}$ ). Only  $w_b$ ,  $w_s$  and  $s_{max}$  are necessary to calculate the number of support beams.

```
(define-class beam-tdd-class
  :inherit-from(object)
  :properties(
    beam-height (default 0.1)
    beam-width (default 10)
    beam-depth (default 0.5)

    stud-height (default 1.0)
    stud-width (default 0.5)
    stud-depth (default 0.5)

    space-between-studs (default 1.0)
  )
  :subobjects(
  )
)
```

**Listing 5.12:** AML code for creating a beam class.

Now the set-up code for the unit tests can be created as seen in listing 5.13. This code is running before each test. It creates a beam with a length of 5 and a space of

1 between the studs that have a width of 1. An additional beam of length 6 with a height of 0.2 is also created. For the second beam the studs cannot be distributed evenly, so the space between them will be less than the maximum space (1)<sup>5</sup> This way it is not necessary to create two models before every test when only one of the models is needed.

```
(deftest 'set-up
  (clear)
  (load "D:\\workspace_win\\src\\beam\\beam.aml")
  (create-model 'beam
    :class 'beam-tdd-class
    :init-form (list
      'beam-width 5
      'beam-height 0.1
      'space-between-studs 1
      'stud-width 1
    )
  )

  (create-model 'beam6
    :class 'beam-tdd-class
    :init-form (list
      'beam-width 6
      'beam-height 0.2
      'space-between-studs 1
      'stud-width 1
    )
  )
)
)
```

**Listing 5.13:** Test code for setting up a beam class.

The beams class needs two subobjects, the main beam and the studs. The main beam uses the built in box-object class directly, so it is added to the subobjects to the beam class. It will be added without any tests since it doesn't use any logic.

```
(main-beam :class 'box-object
  height ^beam-height
  width ^beam-width
  depth ^beam-depth)
```

**Listing 5.14:** Main-beam sub-object

---

<sup>5</sup>Alternatively, if running time is crucial, the developer can create two test scripts, one for a beam length of 5 and another for a beam length of 6.

For the studs, a separate stud-class will be created in listing 5.15. It gets the space-between-studs, height, width and depth from the beam class through the use of the default keyword. The class-expression tells the series-object class that it is going to create boxes.

```
(define-class stud-class
  :inherit-from(series-object)
  :properties(
    space-between-studs (default 1.0)

    height (default 1.0)
    width (default 0.5)
    depth (default 0.5)

    class-expression 'box-object
  )
  :subobjects()
)
```

**Listing 5.15:** Stud class

The stud is added to the beam class similarly to the main beam:

```
(stud :class 'stud-class
  height ^stud-height
  width ^stud-width
  depth ^stud-depth)
```

**Listing 5.16:** Studs sub-object

With the beam class in place and the foundations of a stud class ready, it is necessary to derive the equations for calculating the number of studs ( $n_s$ ) and the space between them ( $s$ ).

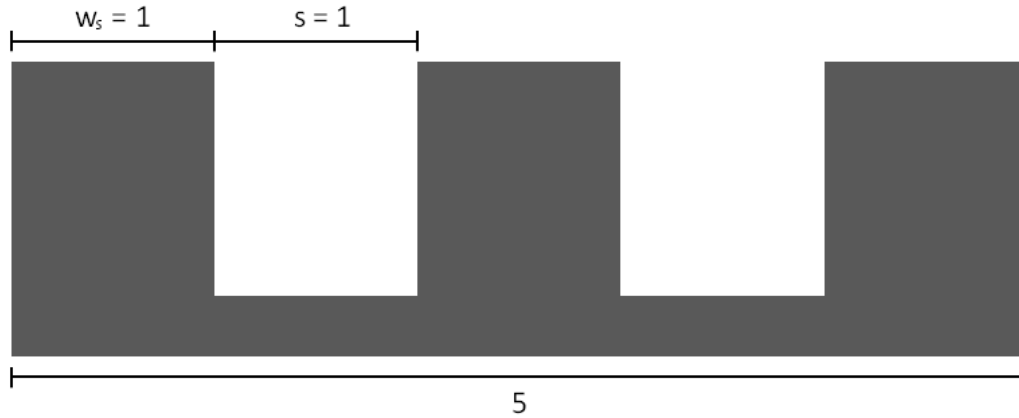
When there are  $n_s$  studs there will be  $n_s - 1$  spaces between them and the total length of the studs and spaces will equal the beam width ( $w_b$ ).

$$w_b = n_s w_s + (n_s - 1)s \quad (5.3)$$

The equation is solved for  $n_s$  to find the number of beams.

$$w_b + s = n_s(w_s + s) \quad (5.4)$$

$$n_s = \frac{w_b + s}{w_s + s} \quad (5.5)$$



**Figure 5.6:** Beam with a width of 5, space of 1 and support beam width of 1

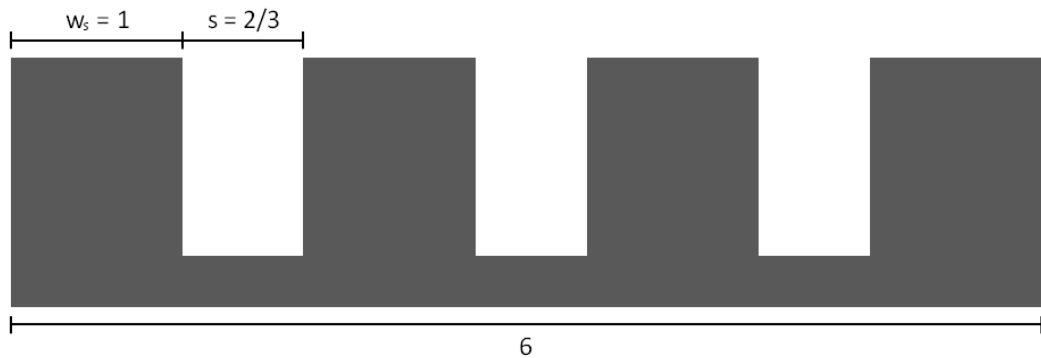
For beams where the given  $s_{max}$  and  $w_s$  does not add up in 5.3 for whole numbers of  $n_s$  we need to add one more beam, resulting in an  $s < s_{max}$ , when it does add up  $s = s_{max}$ . From this it is known that  $s \leq s_{max}$ . Equation 5.3 takes all occurrences into account. This is done by taking equation 5.5, replacing  $s$  with  $s_{max}$  and as a result of this substitution rounding the value up.

$$n_s = \left\lceil \frac{w_b + s_{max}}{w_s + s_{max}} \right\rceil \quad (5.6)$$

Now that  $n_s$  is known the only remaining unknown is  $s$ . Again, taking equation 5.3 as a starting point, but solving it for  $s$  instead of  $n_s$ .

$$s = \frac{w_b - n_s w_s}{n_s - 1} \quad (5.7)$$

Quantity is the first property implemented, it uses equation 5.6. Since this property will calculate the number of studs, it needs a unit test. The following test checks for the simplest scenario where the maximum space and width of the studs can be easily distributed over the beam length ( $s = s_{max}$ ). Since a beam length of 5 is used, with a maximum space of 1 and stud width of 1 the model should create  $\frac{5+1}{1+1} = 3$  studs according to equation 5.6 (see figure 5.6).



**Figure 5.7:** Beam with a width of 6, space of  $\frac{2}{3}$  and stud width of 1

```
(deftest 'stud-quantity-len5
  (check-equals
    (the beam stud quantity)
    3)
)
```

**Listing 5.17:** Test code for checking the number of studs on a beam of length 5.

Setting the quantity property to 3 will pass the test. It is also necessary to check that the studs are calculated correctly when they don't distribute evenly with the maximum space ( $s < s_{max}$ ). When the beam length is increased by 1 from 5 to 6, we should have an additional stud (4 in total) and the space between them should decrease (see figure 5.7).

```
(deftest 'stud-quantity-len6
  (check-equals
    (the beam6 stud quantity)
    4)
)
```

**Listing 5.18:** Test code for checking the number of studs on a beam of length 6.

With the quantity simply set to 3 at all times, this test will fail, so equation 5.6 is implemented in the stud class as follows:



```
quantity (ceiling
          (/
            (+ (the superior superior beam-width)
              ^space-between-studs)
            (+ ^width
              ^space-between-studs)))
```

**Listing 5.19:** Quantity calculation for the studs

This makes both tests pass!

With the quantity of studs in place, it is time to test and implement the rules for the space between them, equation 5.7. Using the first scenario when the beam length is 5 and the stud width is 1, there should be 3 studs and the space between them should be 1.

```
(deftest 'stud-space-len5
  (check-equals
    (the beam stud space)
    1)
  )
```

**Listing 5.20:** Test code for checking the spacing between the studs on a beam of length 5.

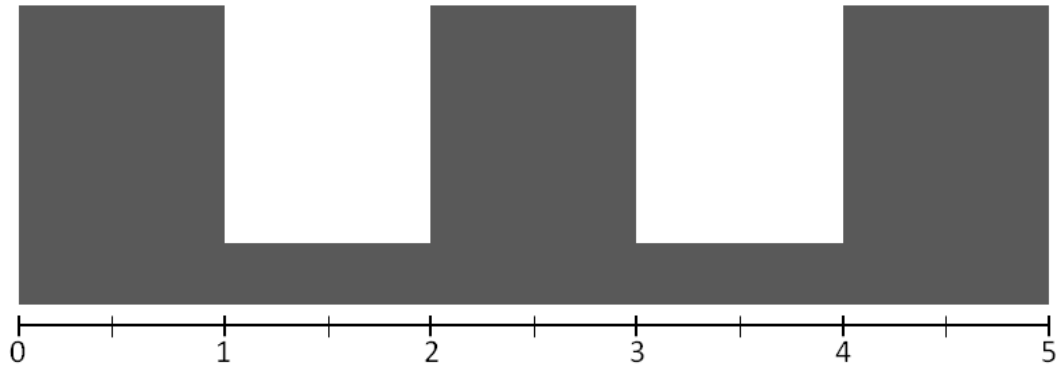
Adding a space property set to 1 to the stud class will pass the test.

When the beam length does not add up with the number of studs and the spaces between them, the space between the studs is reduced so that everything fits on the beam. With 4 studs on the main beam of length 6, the space between them is calculated using equation 5.7,  $s = \frac{6-4 \cdot 1}{4-1} = \frac{2}{3}$ . Another test is created:

```
(deftest 'stud-space-len6
  (check-equals
    (the beam6 stud space)
    (/ 2 3))
  )
```

**Listing 5.21:** Test code for checking the spacing between the studs on a beam of length 6.

To pass both space-tests equation 5.7 is implemented in the stud class.



**Figure 5.8:** Beam with a width of 5, showing the x-coordinates.

```
space (/
  (-
    ^^beam-width
    (* ^quantity ^width))
  (- ^quantity 1))
```

**Listing 5.22:** Space calculation for the studs

Now that the quantity and spacing is in order it is time to place the studs on the main beam.

To avoid too much repetition in this report, the tests for x, y and z-axis placement are created at the same time, although they could be done separately. Since the stud is placed with regard to its center it should be placed half of its width inside the beam, with a width of 1 this will be 0.5. The y-value adds half of the stud height with the height of the beam and will be the same for all the studs. The beam's height is 0.1 and the stud's height is 1,  $0.1 + \frac{1}{2} = 0.6$ . The z-value is always set to 0.

```

(deftest 'stud-position-len5-first-x
  (check-equals
    (first (convert-coords
            (the beam stud stud-0000)
            '(0 0 0) :from :local :to :global))
    0.5)
  )
(deftest 'stud-position-len5-first-y
  (check-equals
    (second (convert-coords
             (the beam stud stud-0000)
             '(0 0 0) :from :local :to :global))
    0.6)
  )
(deftest 'stud-position-len5-first-z
  (check-equals
    (third (convert-coords
            (the beam stud stud-0000)
            '(0 0 0) :from :local :to :global))
    0.0)
  )

```

**Listing 5.23:** Checking the x-axis placement of the first stud.

The beam-support-class inherits the series-object class which creates any number of objects and allows for individual placement of each. In the init-form property the initial values for each support beam is set. The tests in listing 5.23 can easily be passed by setting the x-axis position of all the studs to 0.5, the y-axis to 0.6 and z-axis to 0.

```

init-form '(orientation (list
                        (translate
                         (list
                          0.5
                          0.6
                          0))))

```

**Listing 5.24:** Setting the x-axis placement of the first stud.

The y and z positions will be the same for all the studs, so the next tests will only test the x-axis value. First the second stud is tested to see if the space between them is calculated correctly. With the width of the first stud (1) plus the space between the studs (1) and half of the current stud's width (0.5) it should be placed 2.5 from the edge.

```
(deftest 'stud-position-len5-second-x
  (check-equals
    (first (convert-coords
            (the beam stud stud-0001)
            '(0 0 0) :from :local :to :global))
      2.5)
  )
```

**Listing 5.25:** Setting the x-axis placement of the second stud.

To get this test to pass, the orientation parameter is adjusted to take into account which stud it calculates the position for. The `index` is used to know which stud the position is calculated for, it starts at 0 for the first beam.

```
init-form '(orientation (list
                        (translate
                          (list
                            (+
                              0.5
                              (* !index 1)
                              (* !index 1)
                            )
                          0.6
                          0))))
```

**Listing 5.26:** Setting the x-axis placement of the first stud.

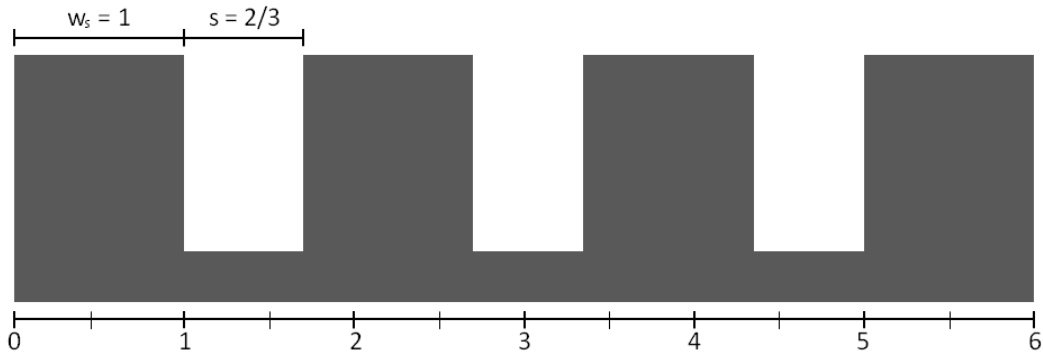
Another test is added to check that the last stud is aligned correctly with the end of the beam, it should be positioned at the end of the beam (the beam length) minus half of its width.

```
(deftest 'stud-position-len5-last-x
  (check-equals
    (first (convert-coords
            (the beam stud stud-0002)
            '(0 0 0) :from :local :to :global))
      4.5)
  )
```

**Listing 5.27:** Setting the x-axis placement of the last stud.

This test passes without any adjustments to the stud class.

The same tests are repeated for the beam with a length of 6. The tests for the first beams x, y and z values are done together like with the last beam.



**Figure 5.9:** Beam with a width of 6, showing the x-coordinates.

```
(deftest 'stud-position-len6-first-x
  (check-equals
    (first (convert-coords
            (the beam6 stud stud-0000)
            '(0 0 0) :from :local :to :global))
    0.5)
  )
(deftest 'stud-position-len6-first-y
  (check-equals
    (second (convert-coords
            (the beam6 stud stud-0000)
            '(0 0 0) :from :local :to :global))
    0.7)
  )
(deftest 'stud-position-len6-first-z
  (check-equals
    (third (convert-coords
            (the beam6 stud stud-0000)
            '(0 0 0) :from :local :to :global))
    0.0)
  )
)
```

**Listing 5.28:** Checking the x, y and z-axis placement of the first stud.

The x and z-tests pass, but the y-tests fail because this beam has a different height than the first one. Instead of using 0.6 directly as the y-axis value it is necessary to calculate it from the height of the beam and the stud:

```
(+
  (half ^height)
  (^ ^beam-height))
```

With this calculation in place all the tests from listing 5.28 are passing.

For the x-axis value for the second stud it is necessary to add the width of the previous stud (1), the space between them (2/3) and half of the beam width.

```
(deftest 'stud-position-len6-second-x
  (check-delta
    (first (convert-coords
            (the beam6 stud stud-0001)
            '(0 0 0) :from :local :to :global))
      ;Stud w, space, half stud w
      (+ 1 (/ 2 3) 0.5)
      0.0001)
  )
```

**Listing 5.29:** Checking the placement of the second stud.

The space between the beams is different from the last beam, so this test will not pass since the code uses 1 directly as the space. With the following calculation for the x-axis value, the-referencing the space and stud width will make the test pass.

```
(+
  (half ^stud-width)
  (* !index ^space)
  (* !index ^stud-width))
```

The last test checks that the fourth stud is positioned to the right edge of the beam. This is done by subtracting half of the stud width from the beam length ( $6 - 0.5 = 5.5$ ).

```
(deftest 'stud-position-len6-last-x
  (check-equals
    (first (convert-coords
            (the beam6 stud stud-0003)
            '(0 0 0) :from :local :to :global))
      5.5)
  )
```

**Listing 5.30:** Checking the placement of the last stud.

The test passes without any changes to the code.

Finally, the code for the stud placement will look like this:

```

init-form '(orientation (list
                        (translate
                         (list
                          (+
                           (half ^stud-width)
                           (* !index ^space)
                           (* !index ^stud-width))
                          (+
                           (half ^height)
                           (^ ^beam-height))
                          0))))

```

**Listing 5.31:** AML code for placing the studs.

### 5.3.3 Testing a bookshelf model

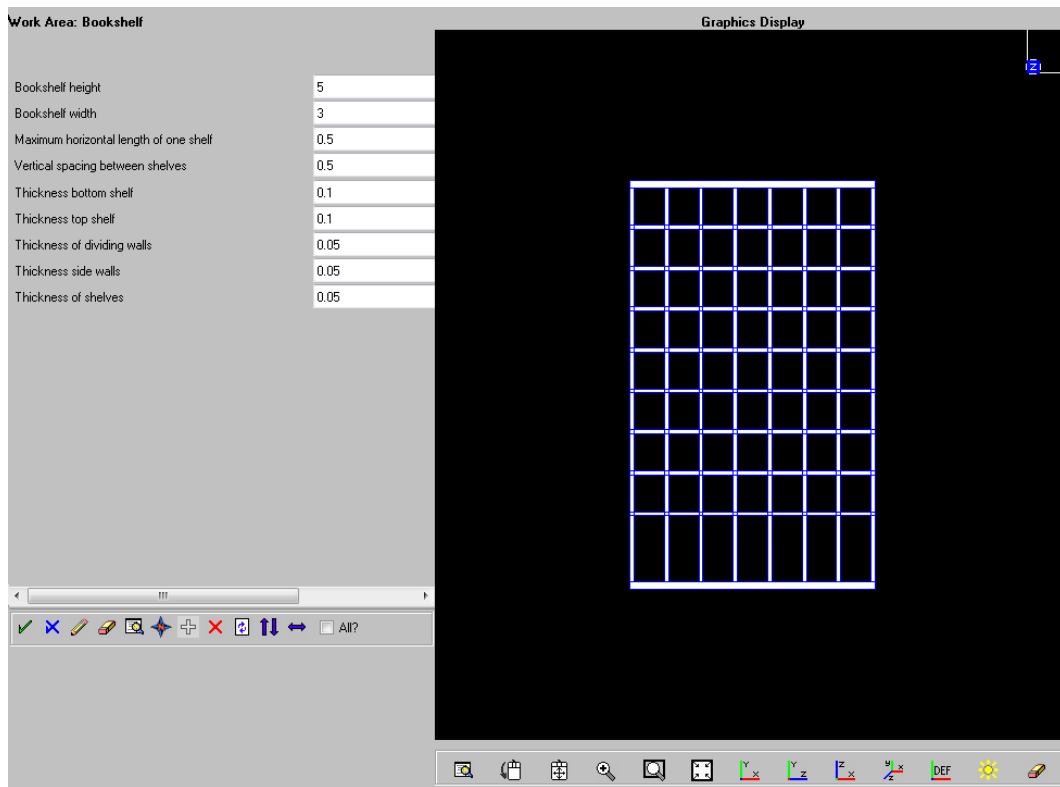
This bookshelf model [30] was created by KBeDesign to demonstrate KBE technology for the LinkedDesign [32] EU-project. It creates a bookshelf that can be automatically designed based on parameters for width, height, shelf and frame thickness as well as a maximum width for each shelf. There is also a parameter for setting the vertical space, so that shelves can be designed for everything from CDs to encyclopedias. If the vertical space cannot be distributed evenly over the height of the bookshelf, extra space is placed in the bottom shelf. See figure 5.10 for the default version of the bookshelf. The complete source code can be found in appendix H.1.

Since the bookshelf model was created before it was decided to unit test it, the tests were not written using TDD, but based on the existing code. Unit tests were made to test all the formulas in the bookshelf and they all worked as expected.

It was not possible to check the input verification rules in *kbe-bookshelf-data-model-class* without generating notification boxes when the rules were broken. The tests can be enabled, but having the user click through a number of alert boxes each time a test is done is not a good option.

All the tests succeeded, but when the bookshelf was drawn, it did not look correct in all cases. With a large lower or upper shelf the inner shelf overlapped the outer frame as seen figure 5.11.

An extra set of tests checking that the original width and height parameters equaled the actual parameters of the model were added and the faults discovered. It turned out that the lower and upper shelf was added on top of the model, in addition to the height input parameter. This means that the formulas themselves were wrong, not taking into account the outer frame. The unit tests were



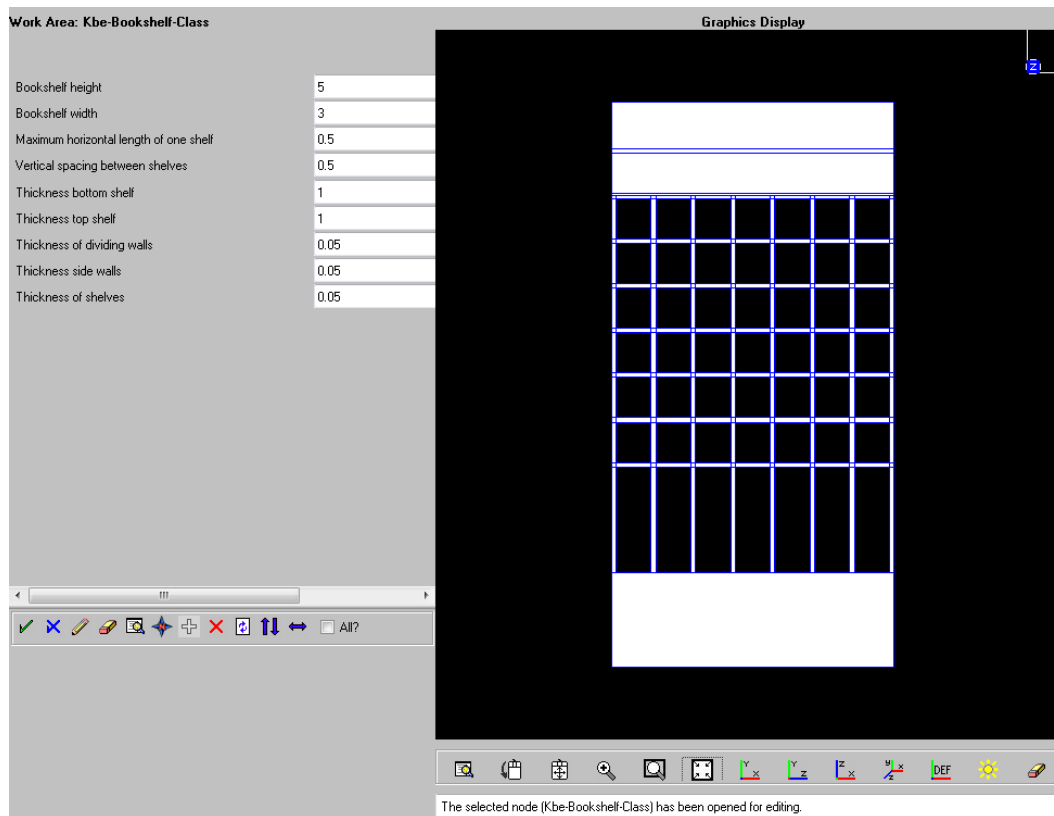
**Figure 5.10:** A bookshelf model.

edited to reflect the proper formulas and the code was adjusted in a TDD manner until all the tests succeeded.

As mentioned in section 5.1, a problem that can arise when testing KBE-models is that the number of subobjects is not known beforehand. As seen in the bookshelf, the number of shelves and inner walls vary depending on the parameter given. This means that these subobjects cannot be tested using direct the-referencing in AML. However, testing all the calculation done in the code will include testing all calculations done for the subobjects. The positioning of the shelves and inner walls are stored in lists that can be iterated over, independent of their size.

An example of this is the testing of the shelf's height and coordinates. First, the code creates a list of height values for each shelf, then a *shelf-coords* list is made calculating the coordinates of each shelf. The complete test code can be found in appendix H.2





**Figure 5.11:** A broken bookshelf model, a shelf is inserted into the top part of the frame.

## 5.4 Testing at Aker Solutions KBeDesign

A testing session took place at Aker Solutions KBeDesign department in April 2012. Aker Solutions are creating a spar platform for Statoil's Luva<sup>6</sup> field. This KBE-model was tested by the author together with Geir Iversen and Johannes Molland from the KBeDesign team.

With only a short time available, it was not feasible to test the entire model. One component, the space manager was chosen. It controls the different planes used in the model, like the x, y and z-axis and its minimum and maximum values (the outer bounds). It was chosen because it is a key feature to the project, but can be troublesome.

Before the tests could run it was necessary to remove some coupling with other

<sup>6</sup>Luva is a gas field in northern Norway where production will start in 2016. Officially renamed to Aasta Hansteen, but still referred to as the Luva project [54].

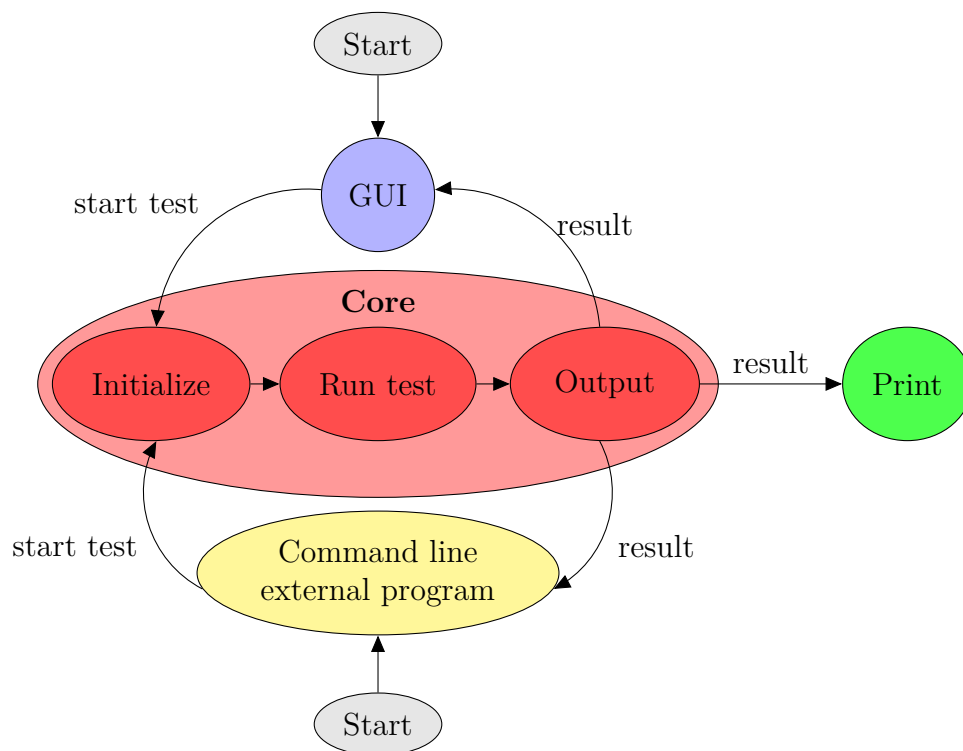
classes, making it possible to instantiate *kbe-spar-space-manager-class* on its own. The complete test code is available in appendix I.

With the limited screen resolution on the meeting room projector it was not possible to have AML's command line interface, the XEmacs windows for programming and the AUnit GUI window up at the same time. It was found that tests were done much faster when they could be done in the same window as AMLs command line interface. Therefore it was found necessary to have a better output from the command line interface of AML for expert users prioritizing speed over simplicity.

# Chapter 6

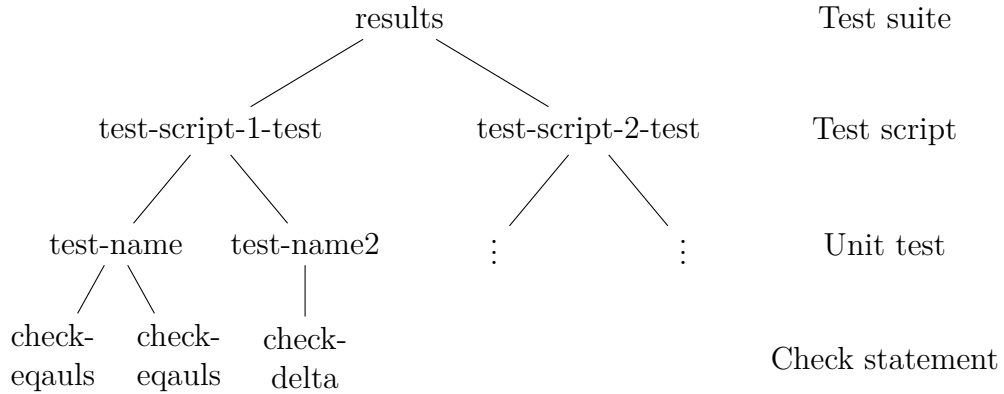
## AUnit

AUnit's foundations were made during a summer job at Aker Solutions KBeDesign and the project thesis [7]. This chapter will explain the technology behind AUnit.



**Figure 6.1:** Relationship between the core, GUI and print modules.

## Test script equivalent:



**Figure 6.2:** AUnit test results tree structure

## 6.1 Overall structure

AUnit is currently comprised of three components; the AUnit core which runs and evaluates the tests, the AUnit GUI which creates a graphical user interface for AUnit and the AUnit print module which prints the test results to text and html files. AUnit is made so that the core can be run independently and the GUI module can be added when needed. Currently, the core needs some functionality from the print module to run the tests.

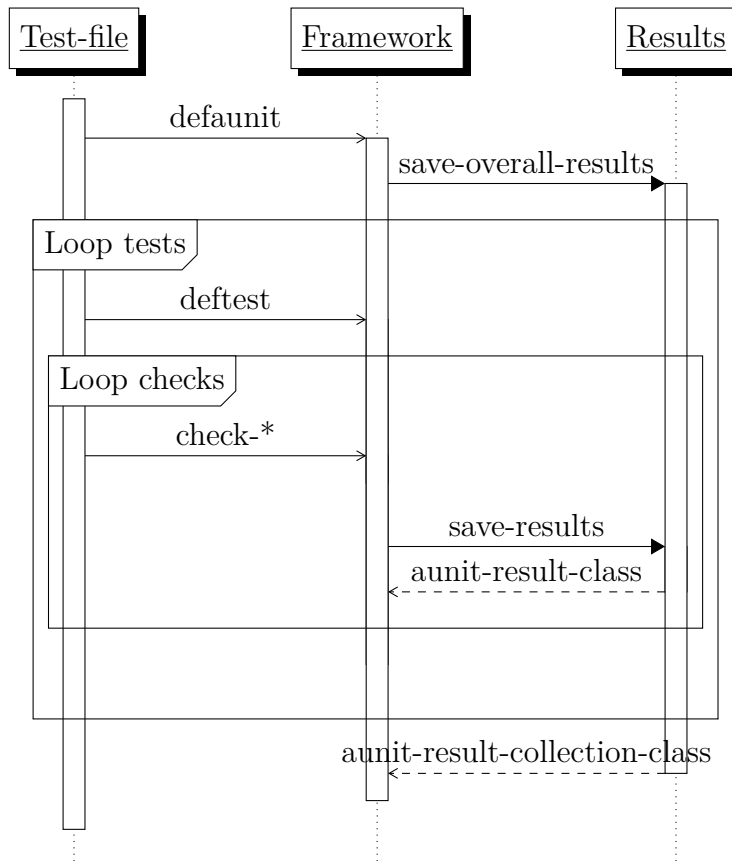
The entrance and exit routes of AUnit are illustrated in figure 6.1. AUnit can be started either from the GUI or via AMLs command line interface. From there the tests are executed in the core. The core sends the results to the command line, back to the GUI or the print module which creates reports.

The results from the tests are saved in a tree-structure as shown in figure 6.2. The three first layers are displayed in the test tree in the GUI-application. The fourth layer is shown in the text box when a user clicks on a unit test.

## 6.2 Core

A substantial amount of work is done to simplify the core functionality in relation to creating the GUI and to ease further development of AUnit.

An instance of `aunit-framework-class` is loaded for every test script. It holds



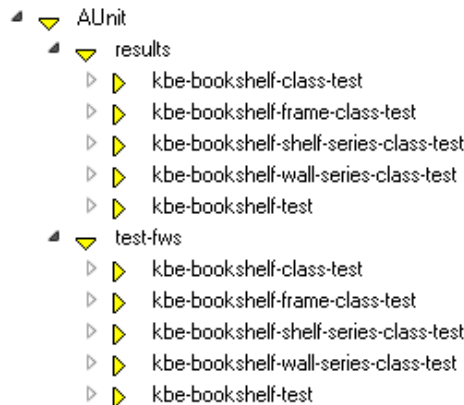
**Figure 6.3:** UML sequence diagram showing how a test is execute inside the core module.

all the information on the test script like its name and file path. An `aunit-result-collection-class` is created to hold all the test results. The test script executes macros defined in the framework, like `defaunit` and `deftest`, see figure 6.3. These macros then save the results to the result-object.

The tests and the results are saved in two separate objects under the main AUnit instance, one for the results and one for the frameworks holding the test information, see figure 6.4.

## 6.3 GUI

The AUnit GUI was created to make AUnit handle multiple, hierarchical, unit tests for entire product models. For AUnit to be an effective tool it was decided to create a GUI that has the ability to take in multiple tests and give immediate



**Figure 6.4:** AUnit object tree

feedback in the same window. The components are completely separate of the command line interface used for development. This allows the developer to rerun all the unit tests with the click of a button, a necessity for TDD as discussed in chapter 2.

On the left side of the window there is a tree that displays the loaded tests. The test tree uses AML’s `ui-model-tree` class, which displays AML objects in a tree structure. The models are filtered to only show the test objects by setting the root object to the results model and the object class to `aunit-display-class`. This ensures that the tree will only show objects that inherit from `aunit-display-class`. It is an empty class that only inherits from the top-level `object` class. With AMLs multiple inheritance features it can be inherited in addition to any other classes.

The `Load def` and `Browse` buttons triggers the `aunit-gui-browse-class` that loads one or more test scripts into AUnit. The two different buttons open the same file browser, but they are filtered for `.def` and `.aml` files respectively.

At the bottom of the window there is a status bar that turns green when all the tests pass, otherwise it remains red. It started as a progress bar turning greener as more tests passed, but it was replaced with a bar that is entirely red or green. This is done because only a single failing test out of hundreds should be treated as severely as one failing test out of five. A progress bar will not show this incident very clearly. With tests independent of each other we might have one failing test for each error [8, p. 149] and a single failing test will do very little impact on a progress bar. Below the status bar there is a field showing the number of successful tests as well as the percentage of successful tests.

`defaunit` creates a list of all the `deftests`. If there is a `set-up` function, it is inserted between every `deftest` function.

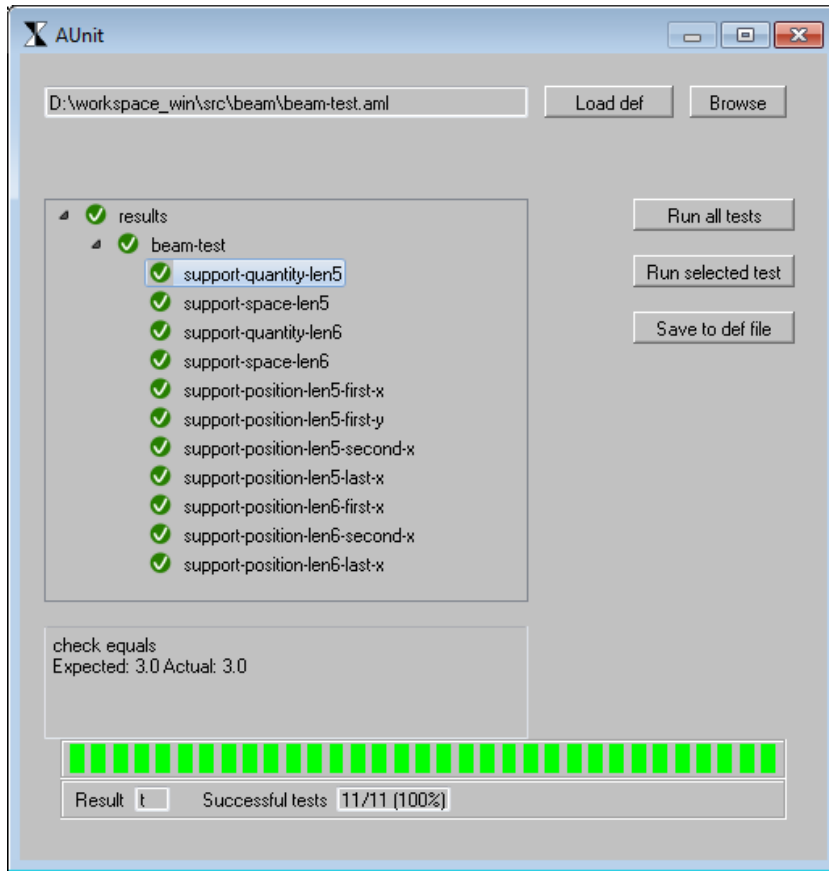


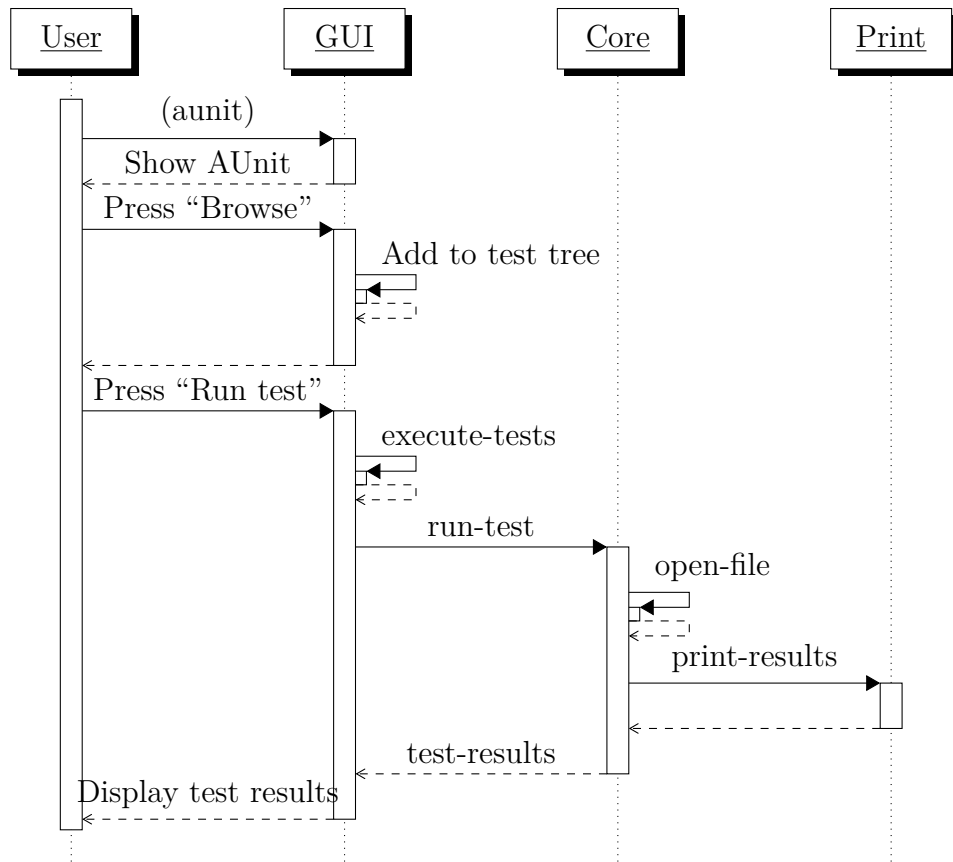
Figure 6.5: AUnit GUI running the beam tests

### 6.3.1 Selecting technology for the AUnit GUI

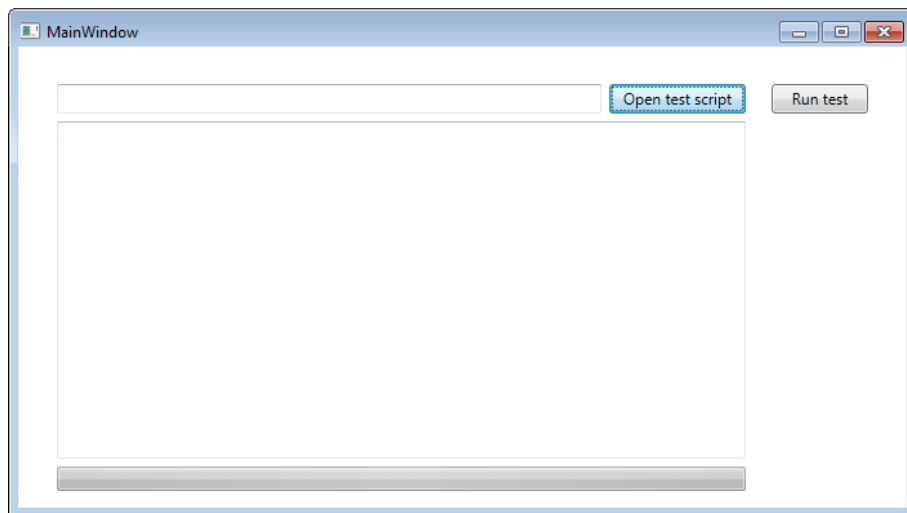
Two main technologies for the AUnit GUI were considered. Either using Microsoft .NET's WPF (Windows Presentation Platform) and the C# programming language or AML's own user interface modules. To find the most suitable technology a preliminary version was created using both technologies.

.NET's WPF has a simple drag and drop based interface in Visual Studio that makes it quick and easy to create GUIs. A few hours work resulted in a simple window as seen in figure 6.7. It includes a file browser, an output box and a progress bar showing if the tests succeeded or failed - all premade modules provided by WPF.

The problem with the .NET solution was to connect it with AML, which is needed to run the tests. Using .NET's `Process.StartInfo` class it was possible to start an instance of AML and redirect the input and output to the .NET application.



**Figure 6.6:** UML sequence diagram showing a user starting AUnit and running a test.



**Figure 6.7:** AUnit GUI test in .NET



However, this does not open the AML canvas needed to draw the models. And even more importantly, only one AML instance can be used at a time without requiring an extra license. Switching between AML instances used for testing and development makes it practically impossible to use the unit tests while programming, like in TDD. Based on this knowledge an effort was made to see what could be done with AML's inbuilt user interface functions.

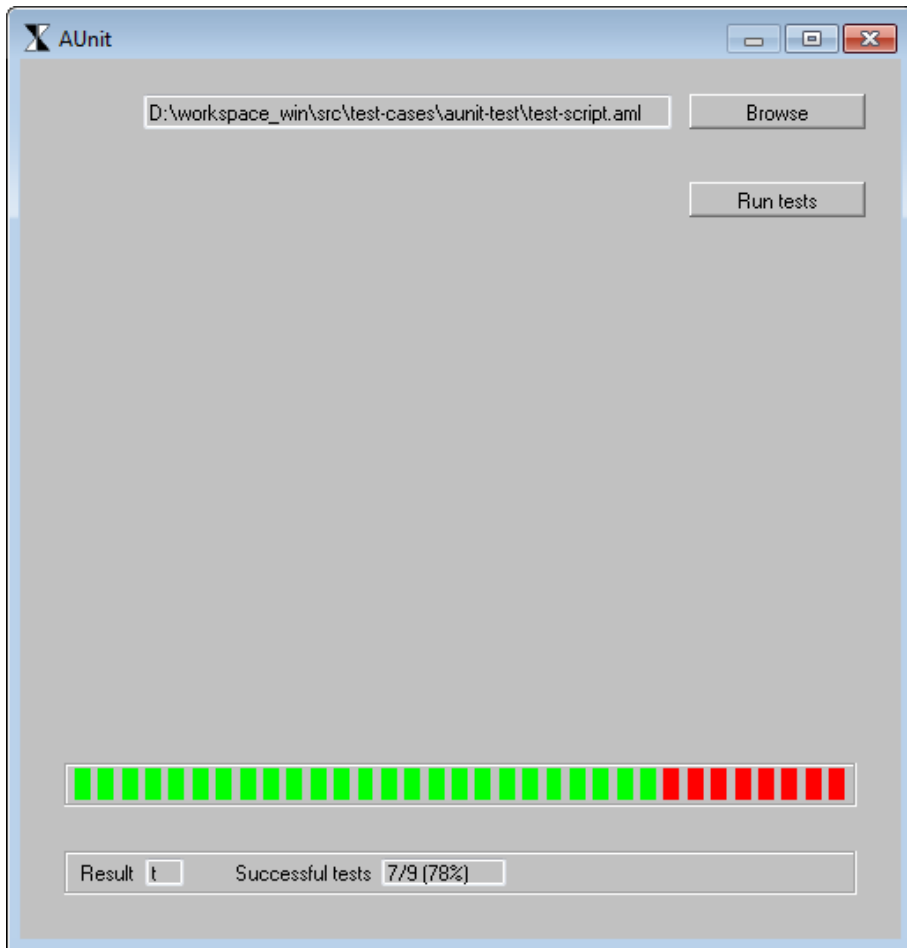
Using the same technology for the GUI and the core of the program avoids the linking issues that .NET has with AML. The challenge with a GUI that is created entirely in AML is the lack of drag and drop systems and fewer inbuilt modules that makes it necessary to write more code for the GUI application.

AML has some user interface elements available similarly to .NET's WPF. Adding a file browser and output window was relatively easy. However, in xUnit the status bar turning from red to green as tests pass is vital. AML does not have any pre-made status bar module so it was necessary to create one using colored boxes. Having done this the AML GUI (see figure 6.8) was just as good as the one made in .NET, but also with a solid connection to the AUnit core.

## 6.4 Print

The print feature was included in the core functionality of AUnit before this report was started. During this project it was separated from the core and made into an independent module, although it is still required by the core.

The print component loops through the result of a test and prints it to a text- and html-file. It has two classes, one containing functions that print html-codes and another that evaluate the test-result object. This separation makes it easier to add new report formats without interfering with the more advanced evaluation code.



**Figure 6.8:** AUnit GUI test in AML

# Chapter 7

## Results and discussion

AUnit has been tested with several KBE-models using both test-driven development (TDD) and continuous integration (CI).

### 7.1 Unit testing KBE models and test-driven development

In the introduction, the following question was asked:

**RQ1:** How can unit tests for AML KBE models be written in a concise manner using known input and output parameters?

In chapter 5 it was shown that unit tests for KBE models cannot be written in the same manner as when doing regular software development. KBE models do a lot of complex calculations that is not known by the developer beforehand. Normally, when writing unit tests, the output can easily be calculated given a certain input.

KBE models also have a different structure than what is seen in other programming projects. The back-bone of a KBE model are classes that have properties and subobjects and they can in turn inherit more properties and subobjects from other classes. When discussing testing it is beneficial to divide the properties and subobjects in a class into two main categories: *Simple properties* like height and length from the input and *existing subobjects* that are pre-made classes. On the other hand there are properties and subobjects that contain logic and calculation, referred to as *logic properties* and *logic subobjects*. In addition a KBE model has functions and methods similar to what can be found in Java and C#.

Three hypotheses were tried on a KBE bottle model:

**Hyp1:** KBE models can be unit tested by directly checking that each property has the expected value.

First, testing every parameter in the model proved to be a very time consuming exercise. In order to check that a property is correct it must be typed into the test as well as in the KBE model. This leads to duplication in the code and test script which makes it more difficult to adapt the test as the code evolves and just double work that just tests that the properties are written correctly in both places. Unit tests should give a certain input and expect an output, without duplicating anything from the code under test.

**Hyp2:** KBE models can be unit tested by calculating their volume and surface area and comparing these values to the volume and surface areas of the objects in the KBE model.

The duplication is taken into consideration in the second hypothesis. By checking the geometry of a KBE model by having the tests calculate the expected volume or surface area and comparing it to the actual volume or surface area in the KBE model, it should be possible to verify that the model is as expected. This approach solves the duplication-issues with the first hypothesis, but they turn out to be just as time-consuming to write. In addition the complexity of the tests is increasing as the model evolves into non-standard geometric shapes. The developer has to do the KBE-system's work, and this leads to that the tests verify AML's functionality, not the KBE-model.

**Hyp3:** KBE models can be tested by checking the return values from just the conditionals, loops and calculations.

A third hypothesis is posed, testing only the logic in the AML-code. This does not result in any duplication and at the same time it is simple and quick to write the tests. The problem with this approach is that simple properties like superclasses and inherited properties are not tested, but must be visually inspected.

Another problem is that the number of subobjects in a model will change depending on the parameters given. This became evident in testing the bookshelf model in section 5.3.3 where the number of shelves and dividers vary greatly. It is not necessary to find each subobject and test them. Instead, the programmer can test

that the code creating the subobjects work correctly. In the bookshelf model there are lists containing the positioning of the shelves, these lists are calculated in listing 7.1. They can be used to test that the shelves are placed correctly, instead of checking the orientation parameter of each shelf.

```
shelf-height-list-1 (let (
  (a
    (list
      (* -1 (+
        (- ^^vertical-spacing-shelves-input
          (half ^^thickness-of-shelves-input))
          (half ^^thickness-top-shelf-input))))))
    (var (* -1
      ^^vertical-spacing-shelves-input))
      )
    (loop for i from 2 to ^^number-of-shelves
      do (push var a))
  a)
  shelf-height-list (reverse ^shelf-height-list-1)
shelf-coords (loop for i from 0 to (- (length ^shelf-height-list) 1)
  sum (nth i ^shelf-height-list) into dist-sum
  collect (multiply-vector-by-scalar ^wall-direction dist-sum)
)
```

**Listing 7.1:** Calculation of shelf position in the bookshelf model

**RQ2:** How can test-driven development be used for developing KBE models in AML?

Test-driven development was performed on the first two KBE examples; the bottle model in section 5.3.1 and the beam in section 5.3.2. Unit testing a KBE-model differs from other unit tests in that large parts of the code cannot be broken into satisfactory units. When the code cannot be broken into units one cannot create unit tests for it and test-driving its development becomes impossible. Therefore TDD for KBE models involves writing code before the tests, it does not allow for a fully test-driven development of KBE models. The developer must, as soon as he tries to write some logic, an equation or a function, write unit tests first and then refer to the regular TDD steps.

One alternative could be to write the same unit tests mentioned above, but start with the tests for the logic first as per the standards in TDD. Then the simple properties can be added when they are needed. This can cause some problems

when the properties are only used by the class' sub- or superclasses. Take for example the bottle example from section 5.3.1. The diameter values are not used by any of the calculations, and can therefore be excluded without the tests failing. Since most properties in AML have a default value, they will instead of being set to their desired value be set to the default value. If the model was developed in a test-driven fashion they could much easier be forgotten than if all the needed properties were to be written first.

It is important to acknowledge this weakness in unit testing KBE models. They are not complete and cannot fully verify a model. Instead they must be used as a tool to minimize the risk of logical and calculation errors. Missing and misplaced simple properties must be discovered through visual inspection of the model. The testing framework can help with this by providing image comparison methods or showing a 3D-model directly in the testing framework. Knowing that the logical and calculation errors can be much more severe than missing properties, the unit tests have a great significance for the accuracy of the KBE models.

Logical errors are more severe than misspelling and omissions because they will normally be much harder to discover without proper tests. A calculation with an error in the formula or a boolean value that should be true instead of false will in most cases not cause any errors when executing the model. Instead the model itself will be wrong. For other errors like inheriting the wrong, or a non-existing superclass it will cause the program to give an error message or the model will be noticeably different from what was intended.

However, one can never be sure that a test can discover all possible bugs in a program. Myers' quote below means that tests should be used to find errors, not verify that there are no more errors in a program.

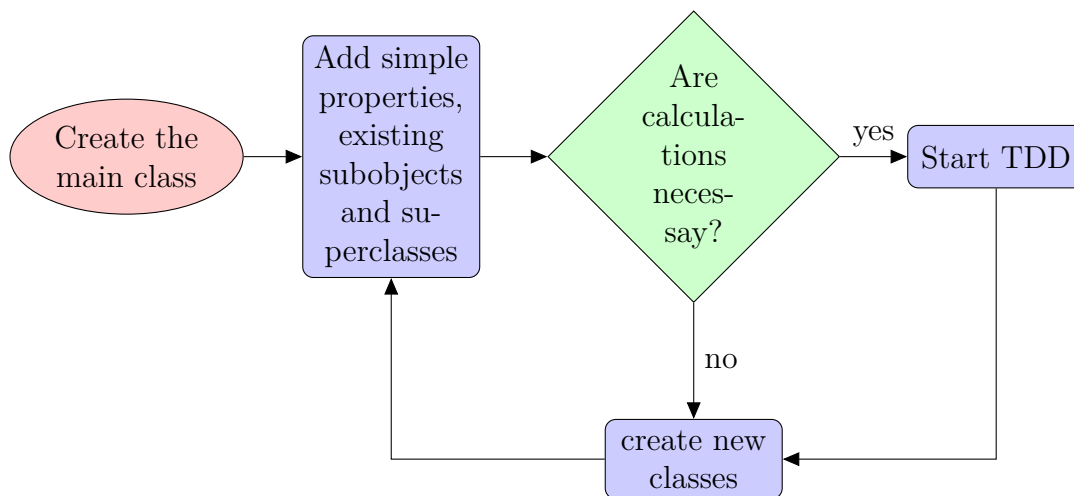
“Testing is the process of executing a program with the intent of finding errors.”

Myers et al. *The Art of Software Testing* [39, p. 6]

### 7.1.1 Step by step

This section will summarize the experiences from the different projects AUnit has been exercised on using the third hypothesis. A step by step guide to creating unit tests for KBE models is shown. The process is summarized in figure 7.1.

- Create the main class. Set inheritance, simple properties and any subobjects that uses already existing classes



**Figure 7.1:** Process for unit testing KBE models

- If any calculations are necessary, write unit tests and follow the steps for TDD as described in chapter 2.
- Create any other necessary classes and repeat.

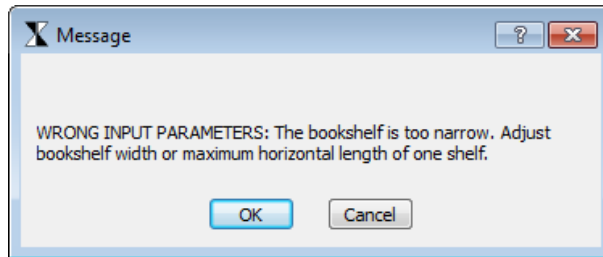
## 7.2 Unit testing at Aker Solutions KBeDesign

There was only one major problem that surfaced during the piloting of AUnit at KBeDesign (section 5.4). In KBeDesign’s code for the Luva project they had tight connections between some components in the model, which made it impossible to run a single component independently. Especially with AML’s powerful the-referencing system it is easy to fall into temptation and tightly integrate your code with the rest of the program. Having separate units of code also greatly encourages reuse of code in different projects.

## 7.3 Continuous integration

**RQ3:** How can continuous integration be used in an environment developing KBE models with AML?

Continuous integration (CI) is used to avoid long integration processes at the end of a development-project by integrating the code into a real-life environment after



**Figure 7.2:** A notification pop-up window in AML. It is used when input parameters are wrong.

every change in the code. CI requires a code repository with code that can be automatically built as well as unit tests and a system to execute the tests and give feedback to the right people.

Performing CI on KBE-models is not different from other software projects. Therefore CI will not be discussed in detail in this report. The problems that emerged while doing CI for KBE models could apply to most other software projects. The main challenge is to find a CI tool that can interact with AML and configuring it properly.

## 7.4 AUnit

There have been a few development processes with the unit testing framework used for testing KBE models, AUnit. It has been split into three modules, a print and GUI module in addition to the core. All of the modules are built using only AML.

AUnit is a working unit testing framework, but there are still several things that can be added or improved, see section 8.1 for some suggestions for further work on AUnit.

### 7.4.1 Verifying input parameters

When testing the bookshelf's input parameter verification rules, AML generated a warning pop-up box when a rule was broken. These pop-up boxes have to be closed manually by the user. With several tests the testing process becomes very manual when several pop-up windows have to be closed every time a test is running. This could be solved by refactoring the code so that the rule checking for correct input is in a separate function that can be tested by AUnit and called from the function



that generates the pop-up warning. However, the method takes in a bookshelf model as a parameter, so a model with the wrong input-parameters has to be created in order to test it. This means that the pop-up box will still be triggered by the initial creation of the model in the set-up function. Another solution might be to have AUnit suppress any warning boxes generated while testing.



# Chapter 8

## Conclusion

AUnit has developed into a fully featured unit testing framework with an easy to use graphical user interface (GUI) as well as a command line interface for expert users and program interfaces.

Although TDD can be done for KBE models, it is not viable to follow the steps of TDD too closely. To efficiently create KBE models it is necessary to set some properties and subobjects before the TDD-process can begin. This form of TDD has been done successfully for different KBE models, also existing KBE models made by KBE professionals has been successfully tested. Because some properties and subobjects are created before the unit tests, a KBE model cannot be fully unit tested in its current form, only the logic in the model is unit tested. The early parameters need visual inspection to be verified. However, the benefit of testing KBE models' logic is substantial as this is where the most severe errors are found.

CI for KBE models works very similar to other programming applications, and existing CI-servers like FinalBuilder [19] can be used.

## 8.1 Further work

The following section will outline some suggestions for further work with unit testing KBE models and development on AUnit.

AUnit can be run from both the command line and a GUI application. It started out giving feedback through text- and html-reports and the latest developments have focused on the GUI application. When tests are run from the command line it would be of great advantage to the users if the results could be printed directly to the user in addition to any reports.

There are a number of check methods available for checking different structures and data types. The handling of these check methods can be improved to make them more intelligent. If each check method could intelligently handle more check cases it would be easier to have an overview of existing methods and add new functionality when needed, for example by using keyword-based input parameters.

For more complicated tests it can be highly beneficial to create local variables. In AML, any local variables are created and accessed inside a let statement. AUnit will not find any checks inside a let statement, instead it will interpret this as a set-up command. This can be solved by recursively checking the test method's content to separate set-up and other non-testing expressions from the test methods.

The AUnit GUI has a tree structure for displaying the loaded tests and their results. It should be possible to manipulate the order of the tests in the tree and create new root nodes to group tests together. Also it should be possible to clear the tree and load a new set of tests.

AUnit has a check-image feature that can compare a screenshot of a KBE model with an existing screenshot. This feature could be built into the GUI, showing the two screenshots.

AUnit should be able to suppress any alerts generated by AML. For example when testing input verification rules that alert the user with pop-up boxes. This is discussed in section 5.3.3 and 7.4.1

The report features a successful implementation of continuous integration (CI) in FinalBuilder. Still there is room for improvements in the robustness of the execution by having tighter integration between AML/AUnit and FinalBuilder, relying less on external files and pressing AML's GUI buttons. Also there are many things that can be done in relation to notifying developers of how their builds are doing in the integration tests.

# Bibliography

- [1] A. Abran et al. “Guide to the software engineering body of knowledge (SWEBOK)”. In: Institute of Electrical and Electronics Engineers, 2004. Chap. 5 Software Testing. URL: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4425813>.
- [2] Aker Solutions. *Aker Engineering & Technology (Fornebu)*. <http://www.akersolutions.com/en/Utility-menu/About-us1/Corporate-structure/Engineering/Aker-Engineering--Technology-AS/>. [Online; accessed 10-May-2012].
- [3] Aker Solutions. *Start page - Aker Solutions*. <http://www.akersolutions.com/>. [Online; accessed 10-May-2012].
- [4] Apache Software Foundation. *Apache Ant - Welcome*. <http://ant.apache.org/>. [Online; accessed 27-March-2012].
- [5] Apache Software Foundation. *Apache Subversion*. <http://subversion.apache.org/>. [Online; accessed 12-March-2012].
- [6] Apache Software Foundation. *Welcome to Apache Maven*. <http://maven.apache.org/>. [Online; accessed 27-March-2012].
- [7] L. Barlindhaug. *Quality Assurance of KBE Software*. Project report. 2011.
- [8] K. Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.
- [9] K. Beck and C. Andres. *Extreme programming explained: embrace change*. second. Addison-Wesley Professional, 2004.
- [10] B. Beizer. *Software testing techniques*. Dreamtech Press, 2002.
- [11] I. Burnstein. *Practical Software Testing: A Process-oriented Approach*. New York, NY, USA: Springer Inc., 2003.
- [12] C. Chapman and M. Pinfold. “The application of a knowledge based engineering approach to the rapid design and analysis of an automotive structure”. In: *Advances in Engineering Software* 32.12 (2001), pp. 903–912.
- [13] Clark M. *Pragmatic Automation*. <http://www.pragmaticautomation.com/cgi-bin/pragauto.cgi/Monitor/Devices/BubbleBubble-BuildsInTrouble.rdoc>. [Online; accessed 21-May-2012].
- [14] D. Cooper and G. L. Rocca. “Knowledge-based techniques for developing engineering applications in the 21st century”. In: *Proceedings of the 7th AIAA Aviation Technology, Integration and Operations Conference, Belfast, Northern Ireland*. 2007.

- [15] CVS. *Concurrent Versions System - Summary*.  
<http://savannah.nongnu.org/projects/cvs>. [Online; accessed 12-March-2012].
- [16] DNV. *Tjenester for risikostyring*.  
<http://www.dnv.no/>. [Online; accessed 8-March-2012].
- [17] P. Duvall, S. Matyas, and A. Glover. *Continuous integration: improving software quality and reducing risk*. Addison-Wesley Professional, 2007.
- [18] L. Erlikh. “Leveraging legacy system dollars for e-business”. In: *IT Professional 2.3* (2000), pp. 17–23.
- [19] FinalBuilder. *VSoft Technologies > Home*.  
<http://www.finalbuilder.com/>. [Online; accessed 27-March-2012].
- [20] M. Fowler. *Refactoring Home*.  
<http://refactoring.com/>. [Online; accessed 10-May-2012].
- [21] M. Fowler. *Continuous integration*.  
<http://www.martinfowler.com/articles/continuousIntegration.html>. [Online; accessed 23-January-2012]. 2006.
- [22] M. Fowler. *Frequency Reduces Difficulty*.  
<http://martinfowler.com/bliki/FrequencyReducesDifficulty.html>. [Online; accessed 30-January-2012]. 2011.
- [23] M. Fowler and M. Foemmel. *Continuous integration*.  
<http://www.martinfowler.com/articles/originalContinuousIntegration.html>. [Online; accessed 30-January-2012]. 2000.
- [24] B. George and L. Williams. “An initial investigation of test driven development in industry”. In: *Proceedings of the 2003 ACM symposium on Applied computing*. ACM, 2003, pp. 1135–1139.
- [25] GlobalSecurity. *Spar Platform*.  
<http://www.global-security.org/military/systems/ship/platform-spar.htm>. [Online; accessed 1-May-2012].
- [26] G. Hopper. “The interlude 1954-1956”. In: *Symposium on Advanced Programming Methods for Digital Computers, Washington, DC, OHR Symposium Report ACR-15*. 1956, pp. 1–2.
- [27] IEEE Computer Society. Standards Coordinating Committee. *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries*. Tech. rep. Institute of Electrical and Electronics Engineers, 1991.
- [28] M. Karlesky et al. “Mocking the embedded world: Test-driven development, continuous integration, and design patterns”. In: *Proc. Emb. Systems Conf, CA, USA*. 2007.
- [29] KBeDesign. *KBeDesign*.  
<http://kbedesign.com/>. [Online; accessed 10-May-2012].
- [30] KBeDesign. “Bookshelf example”. KBE example made for the LinkedDesign project. 2012.
- [31] P.-O. Korsnes. “Analysis of Access Platforms”. MA thesis. Norwegian University of Science and Technology (NTNU), 2010.
- [32] LinkedDesign. *Home*.  
<http://www.linkeddesign.eu/>. [Online; accessed 24-April-2012].

- [33] E. Maximilien and L. Williams. “Assessing test-driven development at IBM”. In: *Software Engineering, 2003. Proceedings. 25th International Conference on*. IEEE. 2003, pp. 564–569.
- [34] S. McConnell. *Code complete*. O’Reilly Media, Inc., 2009.
- [35] Microsoft. *MSBuild Reference*.  
<http://msdn.microsoft.com/en-us/library/0k6kkbsd.aspx>. [Online; accessed 27-March-2012].
- [36] N. Milton. *Knowledge technologies*. Vol. 3. Polimetrica sas, 2008.
- [37] M. M. Müller and F. Padberg. “About the return on investment of test-driven development”. In: *EDSER-5 5th International Workshop on Economic-Driven Software Engineering Research*. Citeseer. 2003.
- [38] R. Munroe. *xkcd: Compiling*.  
<http://xkcd.com/303/>. [Online; accessed 16-April-2012].
- [39] G. J. Myers et al. *The Art of Software Testing*. second. John Wiley & Sons, 2004. ISBN: 978-0-471-46912-4. URL: <http://eu.wiley.com/WileyCDA/WileyTitle/productCd-0471469122.html>.
- [40] NAnt. *NAnt - A .NET Build Tool*.  
<http://nant.sourceforge.net/>. [Online; accessed 27-March-2012].
- [41] L. Narayan et al. *Computer aided design and manufacturing*. PHI Learning Pvt. Ltd., 2008.
- [42] *NaturalGas.org*.  
[http://www.naturalgas.org/naturalgas/extraction\\_offshore.asp](http://www.naturalgas.org/naturalgas/extraction_offshore.asp). [Online; accessed 01-May-2012].
- [43] Norsk Standard. *Norsk Standard*.  
<http://www.standard.no/>. [Online; accessed 8-March-2012].
- [44] Norsk Standard. *Petroleum*.  
<http://www.standard.no/petroleum>. [Online; accessed 8-March-2012].
- [45] R. Osherove. *The Art of Unit Testing*. Manning Publications Co., 2009.
- [46] D. Parnas and P. Clements. “A rational design process: How and why to fake it”. In: *Formal Methods and Software Development (1985)*, pp. 80–100.
- [47] G. Rocca. “Knowledge based engineering: Between AI and CAD. Review of a language based technology to support engineering design”. In: *Advanced Engineering Informatics (2012)*.
- [48] W. Royce. “Managing the development of large software systems”. In: *proceedings of IEEE WESCON*. Vol. 26. 8. Los Angeles. 1970.
- [49] G. Schreiber et al. *Knowledge engineering and management: the CommonKADS methodology*. the MIT Press, 2000.
- [50] Scott Chacon. *Git - Fast Version Control System*.  
<http://git-scm.com/>. [Online; accessed 12-March-2012].
- [51] P. Seibel. *Practical Common Lisp*. APress, 2005. ISBN: 1590592395.
- [52] D. E. Shasha and C. A. Lazere. *Out of their minds: the lives and discoveries of 15 great computer scientists*. Copernicus Series. Copernicus, 1998. ISBN: 9780387982694.

- [53] P. Smith. *File:Waterfall\_model\_(1).svg - Wikimedia Commons*. [http://commons.wikimedia.org/wiki/File:Waterfall\\_model\\_\(1\).svg](http://commons.wikimedia.org/wiki/File:Waterfall_model_(1).svg). [Online; accessed 14-May-2011].
- [54] Statoil. *Looking for suppliers in Northern Norway for Aasta Hansteen*. <http://www.statoil.com/en/OurOperations/FarNorth/Pages/SuppliersForAastaHansteen.aspx>. [Online; accessed 30-April-2012].
- [55] M. Stokes. *Managing Engineering Knowledge - MOKA: Methodology for Knowledge Based Engineering Applications*. Professional Engineering Publishing, 2001. ISBN: 1860582958.
- [56] S. Stolberg. "Enabling Agile Testing through Continuous Integration". In: *Agile Conference, 2009. AGILE'09*. IEEE. 2009, pp. 369–374.
- [57] TechnoSoft Inc. *TechnoSoft Inc: Adaptive Modeling Language*. <http://www.technosoft.com/aml.php>. [Online; accessed 10-May-2012].
- [58] TechnoSoft Inc. *TechnoSoft Inc: Home*. <http://www.technosoft.com/>. [Online; accessed 10-May-2012].
- [59] TechnoSoft Inc. *AML Basic Training Manual*. V3.04. TechnoSoft Inc., 2007.
- [60] TechnoSoft Inc. *AML Reference Manual*. 5.0B5. TechnoSoft Inc., 2010.
- [61] TechnoSoft Inc. *AML Reference Manual*. 5.0B5. TechnoSoft Inc., 2010.
- [62] Wikipedia. *Common Lisp — Wikipedia, The Free Encyclopedia*. [http://en.wikipedia.org/w/index.php?title=Common\\_Lisp&oldid=448448036](http://en.wikipedia.org/w/index.php?title=Common_Lisp&oldid=448448036). [Online; accessed 12-September-2011].
- [63] Wikipedia. *Java (programming language) — Wikipedia, The Free Encyclopedia*. [http://en.wikipedia.org/w/index.php?title=Java\\_\(programming\\_language\)&oldid=489990352](http://en.wikipedia.org/w/index.php?title=Java_(programming_language)&oldid=489990352). [Online; accessed 1-May-2012].
- [64] Wikipedia. *Lisp (programming language) — Wikipedia, The Free Encyclopedia*. [http://en.wikipedia.org/w/index.php?title=Lisp\\_\(programming\\_language\)&oldid=449939607](http://en.wikipedia.org/w/index.php?title=Lisp_(programming_language)&oldid=449939607). [Online; accessed 12-September-2011].



# Appendix A

## Introduction to AML

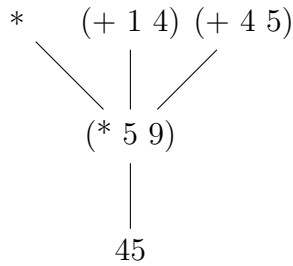
This is an updated and improved version of the AML introduction chapter from the author's project report [7].

In order to have a good understanding on how to test code written in the AML it is important to be familiar with AML's basic code constructors. This chapter will give a brief introduction to AML and show how KBE models are created in AML. It is based on TechnoSoft Inc. (TSI)'s documentation; the *AML Basic Training Manual* [59] and AML's *Reference Manual* [60]. Seibel's *Practical Common Lisp* [51] is used as reference for general Lisp features.

### A.1 AML basics

AML is a Lisp based programming language developed and owned by TSI, an American company focusing on engineering software. AML is completely independent from any Lisp implementations as it has its own compiler written from scratch.

Lisp was designed by John McCarthy in 1958 and has been the de facto language for Artificial Intelligence (AI) research, a term that John McCarthy coined as early as 1956 [52, p. 24-30]. According to Schreiber et al. [49, p. 6], Knowledge systems, which KBE is a subset of, is the most successful commercial result of AI. This creates a natural link between LISP-based programming languages like AML and KBE systems.



**Figure A.1:** AML S-expression evaluation of  $(* (+ 1 4) (+ 4 5))$

### A.1.1 S-expressions

S-expressions (or parenthesized lists) are the building blocks of AML. An S-expression can contain an infinite number of elements, separated by a whitespace. A valid AML-form of an S-expression is an empty list or a list that has a function or method name as its first element. The name *list* calls the list function that creates a list, the symbol  $+$  calls the  $+$  function that adds two numbers together.

In fact an AML program is an S-expression containing a series of other S-expressions. This gives the AML-objects a tree-like structure. The recursive nature of the programming language is the key to its success in KBE and AI.

As an example we will see how AML evaluates a calculation.

```
(* (+ 1 4) (+ 4 5))
```

The process is shown in figure A.1. First it evaluates the innermost arguments,  $(+ 1 4)$  and  $(+ 4 5)$  to 5 and 9 respectively. When all the arguments are evaluated, AML call the multiplication function and we get the answer, 45.

### A.1.2 nil values

AML does not have Boolean values like true/false known from most other programming languages. Instead it has a different concept; *nil* values. For example, in an if-sentence *nil* will cause the false-expression to be evaluated and everything that is *not nil* will cause the true-expression to be evaluated. There is a convention to use *t* to specify that the expression is not nil.

```
(if nil "True" "False") ;;Returns "False"
(if t "True" "False") ;;Returns "True"
(if anything "True" "False") ;;Returns "True"
```

### A.1.3 Class definitions

Classes are the backbone of AMLs object oriented structure. A class definition specifies the class' inheritance, its properties and subobjects. Inheriting a class, known as a superclass makes all methods, properties and subobjects of that class and its superclasses available. A class can inherit from more than one superclass. Subobjects (children) creates an instance of a class, these classes will be child-classes to the class containing the subobjects [61, Ch. 2]. Parameters work as the model's attributes [12, p. 907].

All class names should end with a “-class” suffix.

```
(define-class class-name-class
  :inherit-from (object)
  :properties (
    property-name (default default-property-value))
  :subobjects (
    (subject1 :class 'child-class-name-class)))
```

### A.1.4 The referencing

The referencing is used to access a class' properties and subclasses. It starts from the root node and finds a path down the model tree corresponding to the keywords given as parameters. If a path cannot be found it will go one level up in the tree and attempt to find a valid path. By using the *superior* keyword one can determine how many levels a the-reference should go up the tree before it starts looking for a path.

Given the structure of the bookshelf as shown in figure A.2, a the-reference a bookshelf-shelves-0003 will look like this:

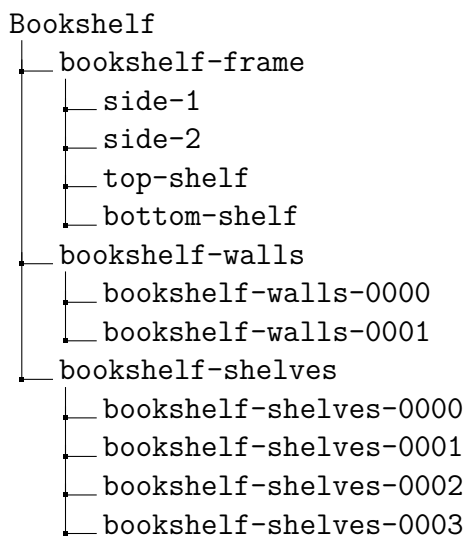
```
(the Bookshelf bookshelf-shelves bookshelf-shelves-0003)
```

If a property in a shelf needs to access a property in the wall, the superior keyword is used.

```
(the superior superior bookshelf-walls bookshelf-walls-0000)
```

### A.1.5 AML subroutines

There are three main types of subroutines in AML; functions, methods and macros.



**Figure A.2:** Bookshelf model structure

### A.1.5.1 Function

A function is the simplest subroutine. It takes in a set of arguments and returns the last AML statement executed. It is loaded into the AML memory and can be called at anytime from anywhere in the code.

Defining a function:

```
(defun function-name (arg1 arg2)
  ;function body)
```

Calling a function:

```
(function-name arg1 arg2)
```

### A.1.5.2 Method

A method is connected to a specific class. This class is specified as the second argument in the method declaration. A method can only be called with an instance of this class, or a subclass of that class. If a method is defined in both a class and a subclass and called from a subclass, the method in the subclass will override the method in the parent class [61, Ch. 2].

A method can make a *the-reference* to the class' properties and subobjects.

Defining a method:

```
(define-method method-name class-name-class (arg1 arg2)
  ;method body)
```

Calling a method:

```
(method-name class-name-class arg1 arg2)
```

### A.1.5.3 Macro

Macros differ from functions and methods. A function or method is evaluated when the AML program is executed. With a macro, the AML compiler will take in the input parameters and expand the macro into the code in all the places where it is called. When the compiler is finished the code will only contain functions and methods, all the macros will be expanded into s-expressions [51, Ch. 4]. Many of the main building blocks in AML are macros, for example the for-loop.

Using a macro definition allows the programmer to improve the language and create new functions on the same level as the pre-made AML functionality.

Defining a macro:

```
(defmacro macro-name (arg1 arg2)
  ;macro body)
```

Calling a macro:

```
(macro-name arg1 arg2)
```



# Appendix B

## Software testing

This is an updated and improved version of the software testing chapter from [7]. This chapter will look at different testing methods and different levels of software testing.

“Testing is the process of executing a program with the intent of finding errors.”

Myers et al. *The Art of Software Testing* [39, p. 6]

This may seem obvious, but it is important to emphasize that the objective of software testing is to find bugs, not to verify that the software is bug-free.

### B.1 Testing pitfalls

A problem with testing is that it can give the programmer a false sense of security. A positive result in a test does not mean that the code is error-free. There can be aspects not covered by the test or there can be bugs passing because of errors in the test itself.

Errors in the test are often brought up when discussing disadvantages regarding software testing. There are two possible outcomes from an error. If the error causes a false positive<sup>1</sup>, the developer will be warned of the error when he finds out that his own code is correct. If the test doesn't find an existing bug, nobody

---

<sup>1</sup>A false positive is when a test finds a bug which is not really a bug, but correct behavior.

will be notified, given Myers' statement above, this is not a problem. We do not aim to find every single bug, just as many as possible.

One should also keep in mind that unit tests are a great deal less complex than the code they are testing<sup>2</sup> and therefore much less likely to contain an error than the code under test.

### B.1.1 Pesticide paradox

Boris Beizer coined the term *pesticide paradox* in his book *Software Testing Techniques* [10, p. 9]. It creates a parallel between how insects become resistant to pesticides and how software bugs will not be discovered by using the same tests again and again.

A developer will run a unit test until it passes and the bug is dead. There is a chance that during these iterations the developer has created new bugs. If a unit has already been buggy, it has a higher probability of containing new bugs. Either because it contains especially complex code or that new bugs appears while the developer tries to fix existing ones [11, p. 139].

Since the code is written to comply with the unit test, the new bug will not be discovered by the same set of tests. Therefore the tests need to always be rewritten as the code is expanding.

## B.2 Testing levels

The Software Engineering Body of Knowledge (SWEBOK) [1, Sec. 2.1] defines three different conceptual testing levels; unit, integration and system testing.

### B.2.1 Unit testing

Unit testing is testing of the smallest form of code that is possible to test separately from the rest of the code [1, Sec. 2.1.1] [11, p. 137].

---

<sup>2</sup>Tests should have a cyclomatic complexity of 1 [45, p. 178], this means that a unit test should not contain any logic.



## B.2.2 Integration testing

“Integration testing is specifically aimed at exposing the problems that arise from the combination of objects.”

Beizer, *Software Testing Techniques* [10, p. 21]

Integration testing is used to make sure that the interaction between the components in the program is working. There are several ways to do integration testing. Bottom-up starts with first integrating the components interacting with the core of the program and then running a set of tests. The testing continues, adding components interacting with the newly integrated layer. By adding layer by layer all the way to the user interface of the program, the software is fully integration tested. Top-down works similarly to bottom down testing, but starts with the user interface and continues downwards to the core [1, Sec. 2.1.2] [27].

Adding all the components at once is usually not recommended and referred to as “big bang” testing [1, Sec. 2.1.2].

## B.2.3 System testing

During system testing the code is completely integrated. The whole system is tested to make sure that it works in accordance with the requirements, usually non-functional requirements. It is also used to verify the integration with external systems. Most failures should already have been identified in the previous testing levels. [1, Sec. 2.1.3]

### B.2.3.1 Functional and non-functional requirements

Functional requirements are set to define how the software should work. For example how the output should be formatted or that a missile should have both a conical and spherical nose.

Non-functional requirements are measured on the finished software. They often concern security, reliability, maintainability and performance [1, Sec. 1.3].

## **B.3 Testing methods**

### **B.3.1 Black box and white box testing**

Black box and white box testing are two of the main testing strategies [11, p. 65]. With black box testing the test cases are based solely on input and output data [1, Sec. 3]. The tester does not know what happens inside the program during the execution. With white box testing the tester looks at the inner workings of the program to write test cases.

Unit testing can be written with both the black box and white box testing strategy and both strategies should be used to ensure completion.

# Appendix C

## AUnit Reference

This chapter contains an in-depth look into AUnit's functions. Some of it is new to this report, other parts are from the author's project report [7].

### C.1 Check functions

The check functions are the AML unit testing framework's assertions. The testing functions built into AUnit are listed below. It is also possible for the user to define new check functions, either at company or project level.

#### C.1.1 check-equals

Check-equals is the most basic test statement. It takes in two parameters and uses AML's *equalp* function to see if they are equal or not<sup>1</sup>. Integer and float data types, for example 2 and 2.0 will be treated as equals and strings will match regardless of case.

---

<sup>1</sup>Where JUnit and NUnit require an assertion method for each data type, like string, integer and float, AML only requires one. This is because it is a dynamically typed language. Dynamically typed language means that variables' data types are not statically declared in the code. Instead it is read from the value, we get that 1, 1.0 '1' and "1" is an integer, float, character and string, respectively.

### C.1.2 check-equals-strict

Check-equals-strict works similarly to check-equals, but uses AML's equal function, so string comparison will be case sensitive and 2 will not match 2.0.

### C.1.3 check-nil and check-not-nil

We learned from section A.1.2 that AML does not have typical Boolean values like true and false. Instead we will create methods that check if an expression is *nil* or *not nil*. *check-nil* passes if the function under test returns *nil*, *check-not-nil* will pass the test for any expression except *nil*. We will use this in an example testing that the value of *display-coord-systems?* is set correctly.

```
(deftest 'test-set-coord-systems
  (change-value (the missile display-coord-systems?) t)
  (check-not-nil (the missile display-coord-systems?)))
(deftest 'test-disable-coord-systems
  (change-value (the missile display-coord-systems?) nil)
  (check-nil (the missile display-coord-systems?)))
```

Note that *display-coord-systems?* could be set to anything except *nil* in the first example and the test will pass. The code will also be treated the same by the AML interpreter.

### C.1.4 check-delta

Check-delta is similar to check-equals, but it takes a third parameter, *delta*. The *delta* parameter allows for some uncertainty in the calculations without the test failing. For example working with floating point numbers, the test can fail after insignificant changes because of rounding the numbers in large calculations. The test will pass if the actual value is within  $\pm\delta$  of the expected value. This is how JUnit's assertEquals function works for decimal numbers. In the example below we test if the body mass is within 0.5 of 16888.

```
(deftest 'test-missile-body-mass
  (check-delta
    (the missile missile-body-mass) 16688 0.5))
```

### C.1.5 check-range

Check-range works similarly to check delta, but instead of defining a delta parameter the range is given explicitly by a range-min and range-max variable.

```
(deftest 'test-missile-body-mass
  (check-range
    (the missile missile-body-mass) 16687.5 16688.5))
```

### C.1.6 check-list-diff

Check-list-diff is used to check the difference between two lists. It takes in two lists and checks if the difference between the values matches a third list.

Check-equals can be used to compare lists, but will not find that lists like (list 1 2 3) and (list 1.0 2.0 3.0) are equal.

```
(check-list-diff
  (convert-coords (the bottle bottom) '(0 0 0)
    :from :local :to :global)
  (convert-coords (the bottle body) '(0 0 0)
    :from :local :to :global)
  (list 0.0 0.0 -2.75))
```

### C.1.7 check-list-delta

Check-list-delta is used to compare two lists. The third parameter is a delta parameter that can set an uncertainty.

```
(check-list-delta
  '(0.0 4.5 0.0)
  (first (the bookshelf bookshelf-frame top-shelf position))
  0.000001)
```



# Appendix D

## TDD approaches

### D.1 Testing model parameters

One way to test the bottle is to test all the properties and check if they are as expected.

Starting with checking the bottle body diameter, it is necessary to first load the bottle file and create the model.

```
(defaunit "bottle-test"  
  (deftest 'body-diameter  
    (load "D:\\workspace_win\\src\\polygon\\src\\bottle.aml")  
    (create-model 'bottle :class 'bottle-class)  
    (check-equals  
      (the bottle body diameter)  
      2.0)  
    )  
  )  
)
```

When running the test AML will give an error message that no bottle class exists. So we create the bottle class and add a body subobject. To make the test pass we also need to set the diameter property to 2.0 in the subobject. At this moment it is not necessary to assign the subobject to a specific class in order to pass the test, so we use the top-level class object.

```

(define-class bottle-par-class
  :inherit-from(object)
  :properties(
  )
  :subobjects(
    (body :class 'object
          diameter 2.0
          )
    )
  )

```

All the tests are succeeding, so the final step in the TDD-process remains, refactor to remove duplication. The diameter is set to 2.0 in both the test and the model, this is duplicate code and will cause trouble when the customer demands a bigger bottle. We need to change this later to check the diameter against the input values.

With the code above we have an object with a diameter of 2, but it does not look like a bottle, in fact it has no shape at all since it only uses the object class. So, in order to obtain a bottle shape we want a cylindrical body. To do this we can add a test checking for the class type of the body. Since the test now have two test methods using the bottle model it is possible to extract the load and create-model statements to a set-up method to avoid duplication.

```

(defaunit "bottle-test"
  (deftest 'set-up
    (load "D:\\workspace_win\\src\\polygon\\src\\bottle.aml")
    (create-model 'bottle :class 'bottle-par-class)
  )

  (deftest 'body-diameter
    (check-equals
      (the bottle body diameter)
      2.0)
    )

  (deftest 'body-class
    (check-equals
      (type-of (the bottle body))
      'open-cylinder-object)
    )
  )
)

```

In the test above the body-class method will fail since it is expecting an open-cylinder-object and it gets an object. This also discovers if the class is set to for example a regular cylinder-object (a cylinder with caps) or an open-cone-object. They both have a diameter property, but gives the wrong shapes.



This is a simple change in our bottle model, we set the class of the body to open-cylinder-object and the test passes.

```
(define-class bottle-par-class
  :inherit-from(object)
  :properties(
  )
  :subobjects(
    (body :class 'open-cylinder-object
           diameter 2.0
           )
  )
)
```

Finally the height of the body must be set to 5. This is done by adding a test similar to the diameter test.

```
(deftest 'body-height
  (check-equals
    (the bottle body height)
    5.0)
)
```

And adding the height property to the body and setting it to 5.0 passes the test.

```
(define-class bottle-par-class
  :inherit-from(object)
  :properties(
  )
  :subobjects(
    (body :class 'open-cylinder-object
           diameter 2.0
           height 5.0
           )
  )
)
```

So, how much value does these unit tests add to the model? They are simple and easy to write, but there is severe duplication in all of the tests. If the developer mistakenly thinks the bottle body should be a cylinder-object instead of an open-cylinder-object, this class will be used in both the test and the model, without any chance of discovering the error before looking at the model and seeing that there is no way to use it. This is not a big problem with the other properties. As mentioned, the diameter and height can be replaced with values derived from the input parameters.

Given that we want a cylinder there are no more efficient ways to tell AUnit that we want a cylinder than to create the cylinder itself in AML. Therefore our test case would just be a repetition of the production code.

The problem here is the test that checks for the open-cylinder-object. From the testing perspective we are not interested in what class is used, we want to have an object with the correct geometry. So what if instead it is tested for the surface area<sup>1</sup> of the cylinder?

## D.2 Testing model geometry

We know that the surface area should be:

$$A = 2\pi rh \tag{D.1}$$

$$A = \pi dh \tag{D.2}$$

So a useful test might be:

```
(defaunit "polygon-test"
  (deftest 'set-up
    (clear)
    (load "D:\\workspace_win\\src\\polygon\\src\\polygon.aml")
    (create-model 'bottle :class 'bottle-class)
  )

  (deftest 'cylinder-surface
    (check-delta
      (volume-of-object (the bottle body))
      (* pi 2 5)
      0.0001
    )
  )
)
```

The check-delta function is used to avoid that any rounding errors to fail the test. This test will only succeed if AML draws a cylinder, for a cone the surface area will be smaller and for the object AML is unable to calculate the surface area.

---

<sup>1</sup>We use surface area and not volume. This is because the cylinder is a hollow object with open ends. AML can only calculate the volume of solid objects.

This test is assuming the diameter and height are set to the correct values, the goal of the test is to discover irregularities in the shape. Therefore it can use the properties directly from the model to avoid any duplication.

```
(deftest 'cylinder-surface
  (check-delta
    (volume-of-object (the bottle body))
    (* pi (the bottle body diameter) (the bottle body height))
    0.0001
  )
)
```

The test above checks that the surface area of the bottle body is as expected, without duplicating any of the code used in the model. It is time to move to the next test.

The bottle needs an end-cap at the bottom. It can be created from a solid cylinder placed at the bottom. The diameter is set to the same as the bottle and the height to 0.5.

Again it is necessary to test that the bottom is drawn in the right shape. The bottom is a solid object so AML will find the volume of it when using the volume-of-object method, therefore we need the formula for the volume of a cylinder.

$$V = \pi r^2 h \tag{D.3}$$

$$V = \pi \left(\frac{d}{2}\right)^2 h \tag{D.4}$$

Using equation D.4 it is possible to create a test for the volume of the bottom.

```
(deftest 'bottom-volume
  (check-delta
    (volume-of-object (the bottle bottom))
    (* pi
      (expt (half (the bottle bottom diameter)) 2)
      (the bottle bottom height))
    0.0001
  )
)
```

AML will first give an error message saying that it is missing the “bottom” sub-object. Creating a bottom subobject like this will pass the test:

```
(bottom :class cylinder-object
  diameter (default 2.0)
  height (default 0.5)
)
```

Unfortunately this is not sufficient. Taking a look at the bottle object we see that the bottom of the bottle is placed in the middle of the bottle body. We need to test for position as well as geometry.

The following test checks for the bottoms position relative to the body of the bottle. Since the reference point is in the middle of the objects it needs to be positioned half of the body's height plus half of the bottoms height below the body.

```
(deftest 'bottom-location
  (check-equals
    (+ (third (convert-coords (the bottle bottom) '(0 0 0)
      :from :local :to :global))
      (half (the bottle bottom height))
      (half (the bottle body height))
    )
    (third (convert-coords (the bottle body) '(0 0 0)
      :from :local :to :global))
  )
)
```

From this test it is simple to adjust the location of the bottom:

```
orientation (list
  (translate
    (list
      0
      0
      (- (+
          (half (the superior superior body height))
          (half (^height))))))
)
```

For creating the top of the bottle it is necessary for a surface area calculation. For creating a simple bottle shape a truncated cone is sufficient.

Given the height  $h$ , start diameter  $D$  and end diameter  $d$  we get the following formula.

The formula for the surface area of a truncated cone is as follows. The top and

bottom areas are ignored since they are not in our model.

$$A = \pi (s(R + r)) \tag{D.5}$$

$$A = \pi \left( s \left( \frac{D}{2} + \frac{d}{2} \right) \right) \tag{D.6}$$

$s$  is the slant height of the cone, to formula to calculate it is as follows:

$$s = \sqrt{\left(\frac{D}{2} - \frac{d}{2}\right)^2 + h^2} \tag{D.7}$$

Then D.7 is inserted into D.6:

$$A = \pi \left( \sqrt{\left(\frac{D}{2} - \frac{d}{2}\right)^2 + h^2} \left(\frac{D}{2} + \frac{d}{2}\right) \right) \tag{D.8}$$

From this it is possible to create a test case for the volume of the top.

```
(defest 'top-surface
  (check-delta
    (volume-of-object (the bottle top))
    (* pi
      (sqrt
        (+ (expt (the bottle top height) 2)
          (expt (- (half (the bottle top start-diameter))
                  (half (the bottle top end-diameter)))
              2)))
        (+ (half (the bottle top start-diameter))
          (half (the bottle top end-diameter))))
      0.0001
    )
  )
```

From this we can create an open-truncated-cone-object to use as a top.

```
(top :class 'open-truncated-cone-object
      start-diameter ~^diameter
      end-diameter ~^end-diameter
      height ~^top-height
  )
```

The location test is similar to the one used for the bottom.

```
(deftest 'top-location
  (check-equals
    (- (third (convert-coords (the bottle top)
      '(0 0 0) :from :local :to :global))
      (half (the bottle top height))
      (half (the bottle body height)))
    (third (convert-coords (the bottle body)
      '(0 0 0) :from :local :to :global))
  )
)
```

```
orientation (list
  (translate
    (list
      0
      0
      (+ (half (the superior superior body height))
        (half (^height))))))
```

From this example we see that it is not trivial to do unit testing of models. Even with simple shapes used in the example the formulas used in the tests become large and it is just as likely that there will be an error in the tests as in the model. However, these problems are easy to discover if the test and model is refined until they match.

The positioning tests are simpler and seem to work well, but here it is easy to make the same mistake in both the test and model resulting in a passed test, but wrong orientation.

# Appendix E

## AUnit source code

Source code for AUnit

### E.1 Core

```
;;;-----  
;;; System   : :aunit  
;;; Purpose  : AML Unit testing framework  
;;;  
;;;  
;;; Author   : Lars Barlindhaug  
;;;  
(define-system :aunit  
  :files '(  
    "aunit-results.aml"  
    "aunit-file-io.aml"  
    "aunit-test-definitions.aml"  
    "aunit-framework.aml"  
    "aunit-run-tests.aml"  
    "kbe-file-comparison-functions.aml"  
  )  
)
```



```
(in-package :aml)

;;;-----
;;; Class    : aunit-display-class
;;; Inherit  : object
;;; Purpose  : Empty class, controls which classes are
;;;           : displayed in the tree.
;;; Notes    : Class is empty with purpose.
;;; Author   : Lars Barlindhaug
;;;
(define-class aunit-display-class
  :inherit-from(object)
  )

;;;-----
;;; Class    : aunit-result-class
;;; Inherit  : object
;;; Purpose  : Holds the results of a test
;;; Notes    :
;;; Author   : Lars Barlindhaug
;;;
(define-class aunit-result-class
  :inherit-from (object)
  :properties (
    result nil
    test-name ""
    type-of-test nil
    tested-object nil
    actual-value nil
    expected-value nil
  )
  )

;;;-----
;;; Class    : aunit-result-collection-class
;;; Inherit  : aunit-display-class
;;; Purpose  : Holds the results of all the tests
;;;           : in a test set.
;;; Notes    :
;;; Author   : Lars Barlindhaug
;;;
(define-class aunit-result-collection-class
  :inherit-from (aunit-display-class)
  :properties (
    result nil
    test-name ""
    successful-tests nil
    total-tests nil
    tests (list)
    setup-cmds (list)
  )
  )

;;;-----
;;; Class    : aunit-result-test-class
;;; Inherit  : property-object
;;; Purpose  : Holds the results of each test.
;;; Notes    :
;;; Author   : Lars Barlindhaug
;;;
(define-class aunit-result-test-class
  :inherit-from (aunit-result-collection-class)
  )

;;;-----
```

```

;;; Function   : find-test-from-result
;;; Purpose    : Finds a test-object from a results object.
;;; Arguments  : result (aunit-result-class)
;;; Returns    : test object (aunit-framework-class)
;;; Author     : Lars Barlindhaug
;;
(defun find-test-from-result (result)
  (let ((test-name (object-name result))
        (test-node (the superior superior test-fws (:from result)))
        (pos (position test-name (subobjects test-node))))
    (if pos
        (nth pos (children test-node))
        (format t "Error finding test ~a in ~a~%" test-name (subobjects test-node))))))

```

```

;;;-----
;;; Function   : find-result-from-test
;;; Purpose    : Finds a result-object from a test-object.
;;; Arguments  : test object (aunit-framework-class)
;;; Returns    : result (aunit-result-class)
;;; Author     : Lars Barlindhaug
;;
(defun find-result-from-test (test)
  (let ((test-name (object-name test))
        (result-node (the superior superior results (:from test)))
        (pos (position test-name (subobjects result-node))))
    (if pos
        (nth pos (children result-node))
        (format t "Error finding result ~a in ~a~%" test-name (subobjects result-node))))))

```

```

(in-package :aml)

;;;-----
;;; Class    : aunit-filo-io-class
;;; Inherit  : object
;;; Purpose  : Writes the output files and reads the
;;;           : expected values files
;;;           : Default location is C:\Data.
;;; Notes    :
;;; Author   : Lars Barlindhaug
(define-class aunit-file-io-class
  :inherit-from (object)
  :properties (
    folder-path (default "C:\\Data")
    expected-folder-path (default (format 'nil "~a\\expected"
^folder-path))
  )
)

(defvar *expected-values* nil)

;;;-----
;;; Method    : load-expected-values
;;; Purpose    : loads the expected values from file.
;;; Arguments  : -
;;; Returns    : list of expected values
;;; Author     : Lars Barlindhaug
;;;
(define-method load-expected-values aunit-file-io-class ()
  (let ((expected-results-file-path (format 'nil "~a\\results.txt" (the e
xpected-folder-path))))
    (with-open-file
      (expected-results-file expected-results-file-path :direction :inp
ut :if-does-not-exist nil)
      (loop for line = (read-line expected-results-file nil :eof)
        until (equal line :eof)
        for eachline = (read-from-string line)
        collect eachline))))

```

```

(in-package :aml)

;;;-----
;;; Class    : aunit-test-definitions-class
;;; Inherit  : object
;;; Purpose  : Sets the output and expected folder paths.
;;; Notes    :
;;; Author   : Lars Barlindhaug
(define-class aunit-test-definitions-class
  :inherit-from (object)
  :properties (
    output-folder-path (default "C:\\Data\\output")
    expected-folder-path (default "C:\\Data\\expected")
  ))

;;;-----
;;; Macro    : check-equals
;;; Purpose   : Compares two values
;;;           : and sends the result to report-result.
;;;           : int with same value and strings in
;;;           : different cases will return t
;;; Arguments : actual-value
;;;           : expected-value (optional)
;;; Returns   : List of tested values and result bool.
;;; Author    : Lars Barlindhaug
;;;
(defmacro check-equals (actual-value &optional (expected-value-arg 'nil))
  (let ((expected-value
        (if expected-value-arg
            expected-value-arg
            (find-expected-value actual-value))))
    ;(format 't "check-equals ~a ~a~%" actual-value expected-value)
    `(report-result
      (with-error-handler (:error-return-value 'nil)
        (equalp ,actual-value ,expected-value))
      "check equals"
      ',actual-value
      ,actual-value
      ,expected-value)))

;;;-----
;;; Macro    : check-equals-strict
;;; Purpose   : Compares two values
;;;           : and sends the result to report-result.
;;;           : int with same value and strings in
;;;           : different cases will return nil
;;; Arguments : actual-value
;;;           : expected-value (optional)
;;; Returns   : List of tested values and result bool.
;;; Author    : Lars Barlindhaug
;;;
(defmacro check-equals-strict (actual-value
                                &optional (expected-value-arg 'nil))
  (let ((expected-value
        (if expected-value-arg
            expected-value-arg
            (find-expected-value actual-value))))
    ;(format 't "check-equals ~a ~a~%" actual-value expected-value)
    `(report-result
      (with-error-handler (:error-return-value 'nil)
        (equal ,actual-value ,expected-value))
      "check equals strict"
      ',actual-value
      ,actual-value
      ,expected-value)))
;;;-----

```

```

;;; Macro      : check-not-nil
;;; Purpose    : Checks if an expression evaluates to
;;;            : something else than nil
;;;            : and sends the result to report-result.
;;; Arguments  : actual-value
;;; Returns    : List of tested values and result bool.
;;; Author     : Lars Barlundhaug
;;;
(defmacro check-not-nil (actual-value)
  `(format 't "check-not-nil ~a~%" actual-value)
  `(report-result
    (with-error-handler (:error-return-value 'nil)
      (not (equal ,actual-value nil)))
    "check not nil"
    ',actual-value
    ,actual-value
    "not nil"))

;;;-----
;;; Macro      : check-nil
;;; Purpose    : Checks if an expression evaluates to nil
;;;            : and sends the result to report-result.
;;; Arguments  : actual-value
;;; Returns    : List of tested values and result bool.
;;; Author     : Lars Barlundhaug
;;;
(defmacro check-nil (actual-value)
  `(format 't "check-nil ~a~%" actual-value)
  `(report-result
    (with-error-handler (:error-return-value 'nil)
      (equal ,actual-value nil))
    "check nil"
    ',actual-value
    ,actual-value
    "nil"))

;;;-----
;;; Macro      : check-delta
;;; Purpose    : Checks if two values are within a
;;;            : difference delta from each other
;;;            : and sends the result to report-result.
;;; Arguments  : actual-value
;;;            : expected-value
;;;            : delta
;;; Returns    : List of tested values and result bool.
;;; Author     : Lars Barlundhaug
;;;
(defmacro check-delta (actual-value expected-value delta)
  `(format 't "check-delta ~a ~a~%" actual-value expected-value)
  `(report-result
    (with-error-handler (:error-return-value 'nil)
      (roughly-equal ,actual-value ,expected-value ,delta))
    "check delta"
    ',actual-value
    ,actual-value
    (format 'nil "(~f +/- ~f)" ,expected-value ,delta)))

;;;-----
;;; Macro      : check-range
;;; Purpose    : check if a value is within a range
;;;            : and sends the result to report-result.
;;; Arguments  : actual-value, range-min, range-max
;;; Returns    : List of tested values and result bool.

```

```

;;; Author      : Kim Nguyen (based on LB's check-equal)
;;;
(defmacro check-range (actual-value range-min range-max)
  ;(format 't "check-range ~a ~a ~a ~%" actual-value range-min range-m
ax)
  `(report-result
    (with-error-handler (:error-return-value 'nil)
      (< ,range-min ,actual-value ,range-max))
    "check range"
    ',actual-value
    ,actual-value
    (format 'nil "(~a - ~a)" ,range-min ,range-max)))

(defun check-list-diff (list1 list2 diff)
  (check-equals (list-diff list1 list2) diff)
  )

(defun list-diff (list1 list2)
  (if (equal (length list1) (length list2))
    (loop for i from 0 to (- (length list1) 1)
      collect (- (nth i list1) (nth i list2)) into diff-list
      finally (return diff-list))
    'nil))

(defmacro check-list-delta (actual-value expected-value delta)
  `(report-result
    (with-error-handler (:error-return-value 'nil)
      (not (list-delta-is-incorrect? ,actual-value ,exp
ected-value ,delta)))
    ;(roughly-equal ,actual-value ,expected-value ,de
lta))
    "check list delta"
    ',actual-value
    ,actual-value
    (format 'nil "(~a +/- ~a)" ,expected-value ,delta)))

(defun list-delta-is-incorrect? (list1 list2 delta)
  (if (or (null list1) (null list2))
    't
    (loop for actual-value in list1
      for expected-value in list2
      when (not (roughly-equal actual-value expected-value delta))
      do
        (return 't))))

;;;-----
;;; Function    : check-image
;;; Purpose     : Calls export-image
;;;             : Checks if the exported image equals a
;;;             : reference image.
;;; Arguments   : image-description (string)
;;; Returns     : report-result return value
;;;             : kbe-test-results-class.
;;; Author      : Lars Barlindhaug
;;;
(defun check-image (image-description)
  (let ((current-image-file-path
        (format 'nil "~a\\images\\~a.tif" *output-folder-path* image-des
cription))
        (reference-image-file-path
        (format 'nil "~a\\images\\~a.tif" *expected-folder-path* image-d
escription)))
    (export-image image-description current-image-file-path)
    (print current-image-file-path)
    (print reference-image-file-path)
  )

```

```

(report-result
  (with-error-handler (:error-function 'write-error-string
                        :error-function-argument 'nil
                        :error-return-value 'nil)
    (kbe-identical-filestreams? reference-image-file-path current-image-file-path))

  "check image"
  image-description
  (format 'nil "<a href=\"images\\~a.tif.png\"><img src=\"images\\~a.tif-thumb.png\" width=\"384\" height=\"224\" /></a>" image-description image-description)
  (format 'nil "<a href=\"..\expected\\images\\~a.tif.png\"><img src=\"..\expected\\images\\~a.tif-thumb.png\" width=\"384\" height=\"224\" /></a>" image-description image-description))))

;;;-----
;;; Function   : export-image
;;; Purpose    : export an image to a file
;;; Arguments  : image-description (string)
;;; Returns    :
;;; Author     : Lars Barlindhaug
;;;
(defun export-image (image-description current-image-file-path)
  (output-display :filepath current-image-file-path
    :format ':tiff :line-width-scale 0.5 :min-line-width 1 :fore-color 'black)
  (let ((png-image-file-path (format 'nil "~a.png" current-image-file-path)
        (png-thumb-image-file-path (format 'nil "~a-thumb.png" current-image-file-path)))
    (output-display :filepath png-image-file-path
      :format ':png :line-width-scale 0.5 :min-line-width 1 :fore-color 'black)
    (output-display :filepath png-thumb-image-file-path
      :format ':png :image-width 384 :image-height 224 :line-width-scale 0.5 :min-line-width 1 :fore-color 'black)))

```

```

(in-package :aml)

;;;-----
;;; Class      : aunit-framework-class
;;; Inherit   : object
;;; Purpose   : Holds the path to save logfiles and images
;;;           : from each test. Default is C:\Data.
;;;           : results-list holds a list of all the
;;;           : results.
;;; Notes    :
;;; Author    : Lars Barlindhaug
(define-class aunit-framework-class
  :inherit-from (object)
  :properties (
    test-script-path (default "C:\\Data\\test-script.aml")
    folder-path (remove-filename-from-path ^test-script-path
^folder-path))
    expected-folder-path (default (format 'nil "~a\\expected" ^e
expected-folder-path))
    reference-image-path (default (format 'nil "~a\\images" ^e
xpected-folder-path))
    output-folder-path (default (format 'nil "~a\\output" ^fol
der-path))
    current-image-path (default (format 'nil "~a\\images" ^out
put-folder-path))

    name (remove-aml-file-extension (find-filename-in-path ^te
st-script-path))
  )
  :subobjects (
    (io :class 'aunit-file-io-class)
    (test-definitions :class 'aunit-test-definitions-class)
  )
)

(defvar *current-test* (the model-manager))
(defvar *expected-values* nil)
(defvar *current-image-path* nil)
(defvar *reference-image-path* nil)
(defvar *name-generator* nil)

;;;-----
;;; Method    : clean
;;; Purpose   : removes all loaded tests, incomplete.
;;; Arguments : -
;;; Returns   : -
;;; Author    : Lars Barlindhaug
;;;
(define-method clean aunit-framework-class ()
  (when (member (read-from-string !name) (subobjects (the superior result
s))))
    (delete-object (find-result-from-test (the)))
    (add-result)
  )
)

;;;-----
;;; Method    : set-up
;;; Purpose   : set the folder paths and get the
;;;           : expected values
;;; Arguments : -
;;; Returns   : -
;;; Author    : Lars Barlindhaug
;;;
(define-method set-up aunit-framework-class ()

```



```

    (setf *expected-values* (load-expected-values (the io)))
    (setf *expected-folder-path* (the expected-folder-path)) ;;used by check-
k-image
    (setf *output-folder-path* (the output-folder-path)) ;;used by check-im
age
    (setf *name-generator* (create-model 'name-generator :init-form (list '
auto-naming? t)))

    (add-result)

    ;;Core dependency to print
    (add-object (the) 'print 'aunit-print-class)

    (the)
    )

;;;-----
;;; Function   : delete-an-object
;;; Purpose    : Deletes an object from the tree,
;;;            : incomplete
;;; Arguments  : object-name (string)
;;;            : parent (aunit-result-*-class)
;;; Returns    : nil if the s-expression do not start
;;;            : with check-, t if it does.
;;; Author     : Lars Barlindhaug
;;;
(defun delete-an-object (object-name parent)
  (let ((children (children parent))
        (subobjects (subobjects parent))
        (position (position object-name subobjects)))
    (if position
        (progn
          (delete-object (nth position children))
          (format t "Deleted: ~a from ~a ~%" object-name (object-name par
ent))))))

(defun add-result ()
  (delete-an-object (read-from-string !name) (the superior results))

  (setf *current-test* (add-object (the superior results)
                                   (read-from-string !name)
                                   'aunit-result-test-class
                                   :init-form
                                   (list
                                    'result "result"
                                    'test-name "test-title"
                                    'successful-tests "successful-tests"
                                    'total-tests "total-tests"
                                    'tests nil
                                    'setup-cmds nil)))
  )

;;;-----
;;; Macro      : defmodtest
;;; Purpose    : Wraps around a model test.
;;;            : evaluates the s-expressions and combines
;;;            : the results for all check-* macros.
;;; Arguments  : test-title (string)
;;;            : class-instance ro set correct paths using check-image
;;;            : forms (s-expressions)
;;; Returns    : save-results return value
;;;            : List including test results,
;;;            : number of tests, successful tests and
;;;            : info from each test.
;;; Author     : Lars Barlindhaug

```

```

;;;           : based on combine-results by Peter Seibel,
;;;           : converted to AML by Geir Iversen
;;;
(defmacro defmodtest (test-title &key (class-instance 'nil) &rest forms)
  `(let ((total (list)) (setup-cmds (list)) (temp-list (list)))
      ,@(loop for f in forms
              if (starts-with? "(check-" f)
                collect `(push ,f total)
              else
                collect `(push ',f total)
              collect f)
      (format 't "total ~a~%" total)
      (save-results ,test-title (reverse total) (reverse setup-cmds))))

```

```

;;;-----
;;; Macro      : defaunit
;;; Purpose    : Wraps around all unit tests.
;;;           : Goes through all deftest unit testing
;;;           : functions and runs the set-up method
;;;           : (if any) before each test.
;;; Arguments  : test-title (string)
;;;           : class-instance ro set correct paths using check-image
;;;           : forms (s-expressions)
;;; Returns    : save-results return value
;;;           : List including test results,
;;;           : number of tests, successful tests and
;;;           : info from each test.
;;; Author     : Lars Barlindhaug
;;;           : based on combine-results by Peter Seibel,
;;;           : converted to AML by Geir Iversen
;;;

```

```

(defmacro defaunit (test-title &key (class-instance 'nil) &rest forms)
  (let ((total (list)) (set-up nil))
    (loop for func in forms
          if (not (eq set-up nil))
            collect (push set-up total)
          if (starts-with? "(deftest 'set-up" func)
            do (setf set-up func)
          if (not (starts-with? "(deftest 'set-up" func))
            collect (push func total)
          ;do (print set-up)
          )
    ;(format 't "all tests ~a~%" (reverse total))
    (save-overall-results
     test-title
     (loop for f in (reverse total)
           collect (eval f))
     'nil)))

```

```

;;;-----
;;; Macro      : deftest
;;; Purpose    : Creates a unit test.
;;;           : Evaluates the s-expressions and combines
;;;           : the results for all check-* macros.
;;; Arguments  : test-title (string)
;;;           : class-instance ro set correct paths using check-image
;;;           : forms (s-expressions)
;;; Returns    : save-results return value
;;;           : List including test results,
;;;           : number of tests, successful tests and
;;;           : info from each test.
;;; Author     : Lars Barlindhaug
;;;           : based on combine-results by Peter Seibel,
;;;           : converted to AML by Geir Iversen

```

```

;;;
(defmacro deftest (test-title &key (class-instance 'nil) &rest forms)
  `(let ((total (list)) (setup-cmds (list)) (temp-list (list)))
      ,@(loop for f in forms
              if (starts-with? "(check-" f)
                collect `(push ,f total)
              else
                collect `(push ',f total)
              collect f)
      ;(format 't "total ~a~%" total)
      (save-results (write-to-string ,test-title) (reverse total) (reverse
se setup-cmds))))

```

```

;;;-----
;;; Function   : starts-with?
;;; Purpose    : cheks if the s-expr. starts with the
;;;            : given string
;;; Arguments  : string (string)
;;;            : form (s-expression)
;;; Returns    : nil if the s-expression do not start
;;;            : with check-, t if it does.
;;; Author     : Lars Barlindhaug
;;
(defun starts-with? (string form)
  (let ((form-as-string (write-to-string form)) (string-length (length st
ring)))
    (when (> (length form-as-string) string-length)
      (if (string= string form-as-string :end1 string-length :end2 string
-length)
          't))))

```

```

;;;-----
;;; Function   : remove-filename-from-path
;;; Purpose    : Removes the file name from a file path
;;; Arguments  : path (string, file path)
;;; Returns    : string, file path
;;; Author     : Lars Barlindhaug
;;
(defun remove-filename-from-path (path)
  (first
   (string-to-delimited-token-list
    path
    :delimiter (format 'nil "\\~a" (find-filename-in-path path))))))

```

```

;;;
;;;-----
;;; Function   : find-filename-in-path
;;; Purpose    : Finds the filename in a file path string
;;; Arguments  : path (string, file path)
;;; Returns    : string, file name
;;; Author     : Lars Barlindhaug
;;
(defun find-filename-in-path (path)
  (first (last (string-to-delimited-token-list
               path
               :delimiter "\\"))))

```

```

;;;
;;;-----
;;; Function   : remove-aml-file-extension
;;; Purpose    : Removes the .aml file extension from a
;;;            : file path or file name.
;;; Arguments  : path (string, file path/file name)
;;; Returns    : string, file name/file path

```

```

;;; Author      : Lars Barlindhaug
;;;
(defun remove-aml-file-extension (filename)
  (first (string-to-delimited-token-list
         filename
         :delimiter ".aml")))
)

-----
;;; Function   : report-result
;;; Purpose    : write the result for a check-* macro
;;;            : to the log file.
;;; Arguments  : result (boolean),
;;;            : type-of-test (string),
;;;            : tested-object,
;;;            : actual-value,
;;;            : expected-value,
;;;            : test-name (string)
;;; Returns    : aunit-result-class
;;; Author     : Lars Barlindhaug
;;;
(defun report-result (result type-of-test tested-object
                    actual-value expected-value &key test-name)
  (add-object
   *current-test*
   (generate-name *name-generator* 'rep-result)
   'aunit-result-class
   :init-form (list
              'result result
              'type-of-test type-of-test
              'tested-object `',tested-object
              'actual-value `',actual-value
              'expected-value `',expected-value)))
)

-----
;;; Function   : save-results
;;; Purpose    : save the results from the tests
;;;            : to a list.
;;; Arguments  : test-title (string),
;;;            : total (list of test results)
;;;            : setup-cmds
;;; Returns    : Creates a aunit-result-collection-class
;;;            : that contains: test results,
;;;            : number of tests, successful tests and
;;;            : aunit-results-class from each test.
;;; Author     : Lars Barlindhaug
;;;
(defun save-results (test-title total setup-cmds)
  (when (not (equal test-title "set-up")))
    (let (
          (successful-tests (count-result-in-list t total))
          (total-tests (length (get-list-of-results total)))
          (result (if (= successful-tests total-tests)
                     't
                     'nil)))
      (let
        ((test-collection
          (add-object
           *current-test*
           (read-from-string test-title)
           'aunit-result-collection-class
           :init-form (list
                     'result result
                     'test-name test-title

```

```

        'successful-tests successful-tests
        'total-tests total-tests
        'tests 'nil
        'setup-cmds 'nil))))

    (add-test-name-to-tests total test-title)
    (change-value (the tests (:from test-collection)) total)
    (change-value (the setup-cmds (:from test-collection)) setup-cmds ↵
)

    (print test-collection))))))

;;;-----
;;; Function   : remove-aml-file-extension
;;; Purpose    : Adds the test name to each
;;;            : aunit-result-class test instance
;;; Arguments  : path (string, file path/file name)
;;; Returns    : string, file name/file path
;;; Author     : Lars Barlindhaug
;;;
(defun add-test-name-to-tests (tests test-name)
  (loop for test in tests
    do
      (if (typep test 'aunit-result-class)
          (change-value (the test-name (:from test)) test-name))))

;;;-----
;;; Function   : save-overall-results
;;; Purpose    : saves the results from each unit tests
;;;            : to a aunit-result-collection-class.
;;;            : Used by defaunit.
;;; Arguments  : test-title (string),
;;;            : total (list of test results)
;;;            : setup-cmds
;;; Returns    : Creates a aunit-result-collection-class
;;;            : that contains: test results,
;;;            : number of tests, successful tests and
;;;            : aunit-results-class from each test.
;;; Author     : Lars Barlindhaug
;;;
(defun save-overall-results (test-title total setup-cmds)
  (let ((tests-results (get-number-success total))
        (total-tests (nth 0 tests-results))
        (successful-tests (nth 1 tests-results))
        (result (get-result total-tests successful-tests)))

    (change-value (the result (:from *current-test*)) result)
    (change-value (the test-name (:from *current-test*)) test-title)
    (change-value (the successful-tests (:from *current-test*)) success ↵
ful-tests)
    (change-value (the total-tests (:from *current-test*)) total-tests)
    (change-value (the tests (:from *current-test*)) total)
    (change-value (the setup-cmds (:from *current-test*)) setup-cmds)

    *current-test*))

;;;-----
;;; Function   : find-expected-value
;;; Purpose    : Finds the expected value of a test
;;;            : from the expected-values file.
;;; Arguments  : object-to-check
;;; Returns    : Expected value
;;; Author     : Lars Barlindhaug
;;; History
;;; Created on : 2011-08-07

```

```

;;; Modified :
;;;
(defun find-expected-value (object-to-check)
  (let ((expected-values *expected-values*))
    (loop for key-value in expected-values
          until (when
                  (equal (nth 0 key-value) object-to-check)
                    (return (nth 1 key-value))))))

;;;-----
;;; Function : count-result-in-list
;;; Purpose  : Count the number of result instances
;;;           that equals element in a list containing
;;;           some aunit-result-class instances.
;;; Arguments: element, list
;;; Returns  : Number of elements in the list.
;;; Author   : Lars Barlindhaug
;;;
(defun count-result-in-list (element list)
  (let ((number-of-elements 0))
    (count-elements-in-list
     element
     (get-list-of-results list))))

;;;-----
;;; Function : get-list-of-results
;;; Purpose  : Gets a list of the result values from
;;;           : a list of aunit-result-classes
;;; Arguments: list
;;; Returns  : List of result values (t/nil)
;;; Author   : Lars Barlindhaug
;;;
(defun get-list-of-results (list)
  (loop for test-result in list
        when (typep test-result 'aunit-result-class)
        collect (the result (:from test-result))))

;;;-----
;;; Function : get-number-success
;;; Purpose  : Gets the number of successful tests
;;;           : from a list of results.
;;; Arguments: list
;;; Returns  : Number of successful tests
;;; Author   : Lars Barlindhaug
;;;
(defun get-number-success (list)
  (let ((number-of-tests 0) (successful-tests 0))
    (loop for test-result in list
          do
            (if (or
                 (typep test-result 'aunit-result-test-class)
                 (and
                  (typep test-result 'aunit-result-collection-class)
                  (not (equal (the test-name(:from test-result)) "set-up"))))
                (progn
                 (setf number-of-tests (+ 1 number-of-tests))
                 (if (the result (:from test-result))
                     (setf successful-tests (+ 1 successful-tests))))))
              (list number-of-tests successful-tests)))

  (defun get-result (total success)
    (if (equal total success)

```

```
t
nil))
```

```
;;;-----
;;; Function : count-elements-in-list
;;; Purpose  : Count the number of elements in a list.
;;; Arguments : element, list
;;; Returns  : Number of elements in the list.
;;; Author   : Lars Barlindhaug
;;;
(defun count-elements-in-list (element list)
  (let ((number-of-elements 0))
    (loop for instance in list
          when (equal instance element)
            do (setf number-of-elements (1+ number-of-elements)))
    number-of-elements))

;;;-----
;;; Function : get-window-resoltuion
;;; Purpose  : Gets the window resolution, can be used
;;;           : to ensure that check-image compares with
;;;           : an image in the correct resolution.
;;; Arguments : -
;;; Returns  : Window resolution (pixels).
;;; Author   : Lars Barlindhaug
;;;
(defun get-window-resolution ()
  (let ((window-coordinates (get-window-coordinates (the current-display) ↵
  )))
    (list (nth 2 window-coordinates)
          (nth 3 window-coordinates))))
```

```

(in-package :aml)

;;;-----
;;; Function   : run-test
;;; Purpose    : Main function of the aunit test framework
;;;            : It takes in a path to a folder containing
;;;            : a test-script.aml.
;;; Arguments  : folder-path (string)
;;; Returns    :
;;; Author     : Lars Barlindhaug
;;;
(defun run-test (aunit test-script-path)
  (print "run-test")
  (clean aunit)
  (let ((result (load (format 'nil "~a" test-script-path))))
    (format 't "Result: ~a" result)
    (print-results (the print (:from aunit)) result)
    result))

;;;-----
;;; Function   : first-setup
;;; Purpose    : Setup done when aunit is started.
;;; Arguments  :
;;; Returns    :
;;; Author     : Lars Barlindhaug
;;;
(defun first-setup()
  (create-model 'aunit :class 'object)
  (add-object (the aunit) 'test-fws 'object)
  (add-object (the aunit) 'results 'object)
  )

;;;-----
;;; Function   : run-script
;;; Purpose    : Runs a testing script
;;; Arguments  : file-path (string)
;;; Returns    :
;;; Author     : Lars Barlindhaug
;;;
(defun run-script (file-path)
  (let ((name
        (read-from-string
         (remove-aml-file-extension (find-filename-in-path file-path))))
        )
    (delete-an-object name (the aunit test-fws))
    (let ((aunit-fw
          (add-object (the aunit test-fws) name 'aunit-framework-class
                     :init-form (list 'test-script-path file-path
                                       'output-file-name "results"))))
      (set-up aunit-fw)
      (run-test aunit-fw file-path)))

;;;-----
;;; Function   : run
;;; Purpose    : Runs a testing script
;;; Arguments  : file-path (string)
;;; Returns    :
;;; Author     : Lars Barlindhaug
;;;
(defun run (file-path)
  (let ((name (remove-aml-file-extension (find-filename-in-path file-path)
    )))
    (test (add-object (the) (read-from-string name) 'aunit-framework-
class
          :init-form (list 'test-script-path file-path

```



```
                                'output-file-name "results"'))))
  (set-up test)
  (run-test test file-path)
)

;;;-----
;;; Function   : aunit-setup
;;; Purpose    : Sets up aunit for each testing script.
;;; Arguments  : test-script-path (string)
;;; Returns    :
;;; Author     : Lars Barlindhaug
;;;
(defun aunit-setup (test-script-path)
  (create-model 'aunit :class 'aunit-framework-class
                :init-form (list 'test-script-path test-script-path
                                  'output-file-name "results"))
  (set-up (the aunit)))
```

```

;;;-----
;;; Function : identical-filestreams?
;;; Purpose  :
;;; Arguments:
;;; Returns  :
;;; Author   : Geir Iversen
;;; History
;;; Created on : 2011-03-01
;;; Modified  :

;;; files are not compared for last changed date etc. hence, filestream a
nd not file identical
(defun kbe-identical-filestreams? (filepath-1 filepath-2 &key debug?)
  (let ((file1 (or (probe-file filepath-1) (kbe-error-file-not-found filepath-1)))
        (file2 (or (probe-file filepath-2) (kbe-error-file-not-found filepath-2))))
    (difference? (/= (file-length file1) (file-length file2))));;avoid char-by-char comparison unless needed
  )
  (when debug? (format 't "File lengths filepath-1: ~a filepath-2: ~a~%"
    (file-length file1)
    (file-length file2)))
  (unless difference?
    (with-open-file (f1 file1 (:direction :input))
      (with-open-file (f2 file2 (:direction :input))
        (loop for read1 = (read-char f1 t :eof)
              for read2 = (read-char f2 t :eof)
              for i upfrom 1
              until (or difference? (equal :eof read1) (equal :eof read2))
              do
                (setf difference? (not (equal read1 read2)))
                (when debug? (format t "~a 1: ~a 2: ~a~%" i read1 read2))))))
    )))
  (not difference?)
  ))

;;;-----
;;; Function :
;;; Purpose  :
;;; Arguments:
;;; Returns  :
;;; Author   : Geir Iversen
;;; History
;;; Created on : 2011-03-02
;;; Modified  :
;;;
;;; TODO reimplement this method using xml-parser - now performs byte by
byte comparison, ignoring GUIDs and xml comments
;;;
(defun kbe-compare-akxml-exports (filepath-1 filepath-2)
  (let ((file1 (or (probe-file filepath-1) (kbe-error-file-not-found filepath-1)))
        (file2 (or (probe-file filepath-2) (kbe-error-file-not-found filepath-2))))
    (difference? nil)
    )
    (with-open-file (f1 file1 (:direction :input))
      (with-open-file (f2 file2 (:direction :input))
        (loop for read1 = (kbe-xml-read-ignore-comments-and-ids f1 t :eof)
              for read2 = (kbe-xml-read-ignore-comments-and-ids f2 t :eof)
              for i upfrom 1
              until (or (equal :eof read1) (equal :eof read2))
              if (not (equal read1 read2))
                (format t "Difference at line ~d: ~a vs ~a~%" i read1 read2))))))
  ))

```

```

collect (list read1 read2) into differences
finally (if (and (equal :eof read1) (equal :eof read2))
  (return differences)
  (return (append differences
    (list
      (let ((f (if (equal :eof read1)
        file1
        file2)))
        (format 'nil "~a bytes from ~a" (file-length f) f)))
    )))))))

;; helper function for kbe-compare-akxml-exports
(defun kbe-xml-read-ignore-comments-and-ids (stream &optional (eof-error-p 'nil) (eof-value :eof))
  (let ((r (read stream t :eof)))
    (if (equal r '<);; hotfix - identifies "start comment" because <!-- i
s read as 2 tokens: '< and (the --)
    (loop for rr = (read stream eof-error-p eof-value)
      until (or (equal rr '-->) (equal rr :eof))
      finally (return rr));; returning -->, because next token may be "sta
rt comment" or id
    (if (and (stringp r) (or (find "ID" r) (find "id" r)));; skip GUID
strings by performing an additional read
      (read stream t :eof)
      r)))));;return r unless comment or string containing ID

```

## E.2 GUI

```
;;;-----  
;;; System   : :aunit-gui  
;;; Purpose  : Graphical User Interface for  
;;;           AML Unit testing framework  
;;;  
;;; Author   : Lars Barlindhaug  
;;;  
(define-system :aunit-gui  
  :require-systems '(:aunit :aunit-print)  
  :files '(  
    "aunit-gui-browse.aml "  
    "aunit-gui-results-bar.aml "  
    "aunit-gui-progress-bar.aml "  
    "aunit-gui-result-text-field.aml "  
    "aunit-gui-tree.aml "  
    "aunit-gui.aml "  
  )  
)
```

```

(in-package :aml)

(define-class aunit-test-script-selection-class
  :inherit-from(file-selection-property-class)
  :properties(
    ;directory (default "D:\\workspace_win\\src\\test-cases\\au
nit")
    filter (default "*.aml")
  )
)

(define-class aunit-gui-browse-button-class
  :inherit-from(ui-action-button-class)
  :properties()
)

(define-class aunit-gui-browse-button-file-class
  :inherit-from(aunit-gui-browse-button-class)
  :properties(button1-action '(progn
    (let ((file-path (select-file-dialog
      :filter "*.aml")))
      (when (not (equal file-path nil))
        (add-to-test-tree (the superior superior
ior) file-path)
        (file-selected-update-gui file-path))
      )))
)

(define-class aunit-gui-browse-button-def-class
  :inherit-from(aunit-gui-browse-button-class)
  :properties(button1-action '(progn
    (let ((file-path (select-file-dialog
      :filter "*.def")))
      (when (not (equal file-path nil))
        (change-value ^^def-file file-path)

        ;reverse, starts with last item which
is placed at the bottom of the list.
        (loop for test-file-path in (reverse
(get-tests-from-def-file file-path))
          do
            (add-to-test-tree
              (the superior superior)
              (format 'nil "~a\\~a"
                (remove-filename-from-path
file-path)
                test-file-path))
            )
          (file-selected-update-gui file-path))
      )))
)

(defun get-tests-from-def-file (file)
  (with-open-file (file file
    :direction :input)
    (loop for line = (read-line file nil :eof)
      until (equal line :eof)
      collect line)))

(defun file-selected-update-gui (file-path)
  (replace-text (the superior chosen-test-field) file-path)

  (change-value (the superior action gray?) nil)

  (change-value (the superior run-selected-test gray?) nil)
  (change-value (the superior save-test-def gray?) nil)
  (change-value (the superior clear-tree gray?) nil)

```

```
(the superior superior update?)
)

(defun reset-update-gui ()
  (replace-text (the superior chosen-test-field) "")

  (change-value (the superior action gray?) t)

  (change-value (the superior run-selected-test gray?) t)
  (change-value (the superior save-test-def gray?) t)
  (change-value (the superior clear-tree gray?) t)
  (the superior superior update?)
)

(define-class aunit-choosen-test-field-class
  :inherit-from(ui-field-class)
  :properties(
    content "Please browse for a test."
    editable? nil
  )
)
```

```

(in-package :aml)

(define-class aunit-gui-results-form-class
  :inherit-from(ui-subform-class)
  :properties(
    frame? t
  )
  :subobjects(
    (result-label :class 'ui-label-class
      x-offset 0
      y-offset 10
      width 10
      height 80

      label "Result"
    )

    (result-field :class 'ui-field-class
      x-offset (+
        (the superior superior result-label x-offset)
        (the superior superior result-label width))
      y-offset (the superior superior result-label y-offset)
      width 5
      height 80

      content ""
      editable? nil
    )

    (successful-label :class 'ui-label-class
      x-offset (+
        (the superior superior result-field x-offset)
        (the superior superior result-field width)
        ^^element-x-margin)
      y-offset (the superior superior result-field y-offset)

      label "Successful tests"

      width 20
      height 80
    )

    (successful-field :class 'ui-field-class
      x-offset (+
        (the superior superior successful-label x-off↵
        (the superior superior successful-label width↵
      y-offset (the superior superior successful-label y-offs↵

      width 16
      height 80

      content "0/0 (0%)"
      editable? nil
    )
  )
)

```



```

(in-package :aml)

(define-class aunit-gui-bar-class
  :inherit-from(ui-subform-class)
  :properties(
    x-offset (default 1)
    y-offset 5
    width 2
    height 90

    background-color (default 'red)
  )
  :subobjects()
)

;;STATUS bar
;; either completely RED or completely GREEN
(define-class aunit-gui-status-bar-class
  :inherit-from(series-object ui-subform-class)
  :properties(
    ;;ui-subform properties
    frame? t

    ;;series-object properties
    quantity 33
    class-expression 'aunit-gui-bar-class
    init-form '(
      x-offset (+ (* (+ ^space !width) !index) ^space
        background-color (if ^success? 'green
' red)
      )

    space 1
    success? nil

  )
  :subobjects()
)

(define-method update-status-bar aunit-gui-status-bar-class (success?)
  (change-value !success? success?)
  !update?
)

;;PROGRESS BAR
;; shows progress, takes in an integer between 0 and 1.
(define-class aunit-gui-progress-bar-class
  :inherit-from(series-object ui-subform-class)
  :properties(
    ;;ui-subform properties
    frame? t

    ;;series-object properties
    quantity 33
    class-expression 'aunit-gui-bar-class
    init-form '(
      x-offset (+ (* (+ ^space !width) !index) ^space
        background-color (if (>= !index ^green
' red
' green))
      )

    space 1
    green-bars 0
  )
  :subobjects()
)

```

```
)  
  
(define-method update-progress-bar aunit-gui-progress-bar-class (success-↵  
decimal)  
  (change-value !green-bars (floor (* !quantity success-decimal)))  
  !update?  
)
```

```

(in-package :aml)

(define-class aunit-gui-result-text-field-class
  :inherit-from(ui-field-class)
  :properties(
    content ""
    input-rows 10)
  :subobjects()
)

(define-method clear-results aunit-gui-result-text-field-class ()
  (replace-text (the) ""))

(define-method add-results aunit-gui-result-text-field-class (test)
  ;; checking for test=nil, this happens because
  ;; when two list-boxes are in use, both button-actions are fired.
  (when (not (null test))
    (insert-text (the)
      (format 'nil "~a : ~a~%~a ~a~%"
        (format 'nil "~a"
          (the type-of-test (:from test)))
        (format 'nil "~a"
          (the tested-object (:from test)))
        (if (numberp (the expected-value (:from test)))
            (format 'nil "Expected: ~f"
              (the expected-value (:from test)))
            (format 'nil "Expected: ~a"
              (the expected-value (:from test))))
        (if (numberp (the actual-value (:from test)))
            (format 'nil "Actual: ~f"
              (the actual-value (:from test)))
            (format 'nil "Actual: ~a"
              (the actual-value (:from test))))
        (the actual-value (:from test))))
      -2)))

```

```

(in-package :aml)

(defvar *pass-img* "D:\\workspace_win\\src\\aunit\\gui\\v.bmp")
(defvar *fail-img* "D:\\workspace_win\\src\\aunit\\gui\\x.bmp")
(defvar *grey-img* "D:\\workspace_win\\src\\aunit\\gui\\grey.bmp")

(define-class aunit-gui-tree-class
  :inherit-from(ui-model-tree)
  :properties(
    root-object 'nil
    object-class 'aunit-display-class

    button1-action '(left-click !selected-item)
    button3-action '(execute-single-test
                     (the superior superior)
                     (the superior superior test-tree selected-
item))
  )
  :subobjects()
)

(define-method save-tree-to-file aunit-gui-tree-class ()
  (let
    ((def-file
      (if (equal (the def-file) "")
          (format nil "~a\\tests.def"
                  (remove-filename-from-path
                    (the content (:from !chosen-test-field))))
          (the def-file))))
      (format t "Writing tests to: ~a~%" def-file)
      (with-open-file (file def-file
                          :direction :output)
        (loop for result in (reverse
                             (tree-item-children
                              (the root-object (:from (the))))))
              do
                (progn
                  (format file "~a~%"
                          (find-filename-in-path
                           (the test-script-path
                            (:from (find-test-from-result result))))
                          (format t "~a~%"
                                  (find-filename-in-path
                                   (the test-script-path
                                    (:from (find-test-from-result result))))))))))
    )

  (define-method clear-tree aunit-gui-tree-class ()
    (progn
      (delete-tree-items (the) (tree-item-children (!root-object)))
      (reset-update-gui))
    )

  (define-method prepare-tree aunit-gui-tree-class ()
    (add-object (the superior) 'results 'aunit-display-class)
    (add-object (the superior) 'test-fws 'aunit-display-class)
    (change-value !root-object (the superior results))
  )

  (define-method set-all-images aunit-gui-tree-class (root img)
    (update-tree-item (the) (the superior (:from root)) :image img)
    (update-tree-item (the) root :image img)
    (let ((children (tree-item-children root)))

```

```
(loop for child in children
  do
    (if (tree-item-children child)
      (set-all-images (the) child img)
      (update-tree-item (the) child :image img))))

(define-method set-test-result-image aunit-gui-tree-class (test)
  (set-img (the) test (the result (:from test)))
  (loop for child in (tree-item-children test)
    do
      (if (typep child 'aunit-result-collection-class)
        (set-img (the) child (the result (:from child))))))

(define-method update-root-img aunit-gui-tree-class (result)
  (set-img (the) (the results) result))

(define-method set-img aunit-gui-tree-class (node result)
  (let ((img
        (if result
            *pass-img*
            *fail-img*)))
    (update-tree-item (the) node :image img)
  )
)

(defun left-click (selected-item)
  (when (typep selected-item 'aunit-result-collection-class)
    (evaluate-results-tree (the superior superior) selected-item)
  )
)

(defun test-already-in-tree? (test-name tree-root-children)
  (let ((name-list
        (loop for test in tree-root-children
          collect (object-name test))))
    (if (member test-name name-list)
        't
        'nil)))
)
```

```

(in-package :aml)

(define-class test-form-class
  :inherit-from(ui-form-class)
  :properties(
    label "AUnit"
    x-offset 300
    y-offset 200
    width 500
    height 500
    measurement 'percentage

    side-margin 4
    element-margin 10
    element-x-margin 3

    def-file ""
  )
  :subobjects(
    (chosen-test-field :class 'ui-field-class
      x-offset ^^element-x-margin
      y-offset ^^side-margin
      width 60
      height 4

      content "Please browse for a test."
      editable? nil
    )

    (browse-def :class 'aunit-gui-browse-button-def-class
      x-offset (+ (the superior superior chosen-test-field x-↵
offset)
                  (the superior superior chosen-test-field wi↵
dth)
                  2)
      y-offset ^^side-margin
      width 16
      height 4

      label "Load def"
    )

    (browse :class 'aunit-gui-browse-button-file-class
      x-offset (+ (the superior superior browse-def x-offset)
                  (the superior superior browse-def width)
                  2)
      y-offset ^^side-margin
      width 12
      height 4

      label "Browse"
    )

    (test-tree :class 'aunit-gui-tree-class
      x-offset ^^element-x-margin
      y-offset (the superior superior action y-offset)
      width 60
      height 50
    )

    (action :class 'ui-action-button-class
      x-offset (right-adjustment ^width ^^side-margin)
      y-offset (+ (the superior superior browse y-offset)
                  (the superior superior browse height)

```

```

        ^element-x-margin)
width 20
height 4

label "Run all tests"
gray? t

button1-action '(execute-all-tests (the superior superior
or))
)

(run-selected-test :class 'ui-action-button-class
x-offset (right-adjustment ^width ^side-margin)
y-offset (+ (the superior superior action y-offset)
(the superior superior action height)
^element-x-margin)
width 20
height 4

label "Run selected test"
gray? t

button1-action '(execute-single-test
(the superior superior)
(the superior superior test-tree selec
ted-item))
)

(save-test-def :class 'ui-action-button-class
x-offset (right-adjustment ^width ^side-margin)
y-offset (+ (the superior superior run-selected-test y
-offset)
(the superior superior run-selected-test h
eight)
^element-margin)

width 20
height 4

label "Save to def file"
gray? t

button1-action '(save-tree-to-file ^test-tree)
)

(clear-tree :class 'ui-action-button-class
x-offset (right-adjustment ^width ^side-margin)
y-offset (+ (the superior superior save-test-def y-offs
et)
(the superior superior save-test-def height
)
^element-x-margin)

width 20
height 4

label "Clear all tests"
gray? t

button1-action '(clear-tree ^test-tree)
)

(test-output :class 'aunit-gui-result-text-field-class
x-offset ^element-x-margin
y-offset (+ (the superior superior test-tree y-offset)
(the superior superior test-tree height)

```

```

        ^element-x-margin)
      width 94
      height 20
    )

    (results-bar :class 'aunit-gui-results-form-class
      x-offset 5
      y-offset 90
      width 90
      height 5
    )

    (status-bar :class 'aunit-gui-status-bar-class
      x-offset 5
      y-offset 85
      width 90
      height 5
    )
  )

  (defvar *gui-name-generator* 'nil)

  ;;-----
  ;; Method      : execute-all-tests
  ;; Purpose     : Done when pressing "run all tests" btn
  ;;            : Executes all the tests in the tree.
  ;; Arguments   : test-result (aunit-result-collection-class)
  ;; Returns    :
  ;; Author     : Lars Barlindhaug
  ;;
  (define-method execute-all-tests test-form-class ()
    (let ((sum-successful-tests 0)
          (sum-total-tests 0))

      (loop for test-result in (tree-item-children (the results))
        do
          (let ((test (find-test-from-result test-result))
                (if (typep test 'aunit-framework-class)
                    (let ((result-list (execute-test (the) test)))
                      (setf sum-successful-tests (+ sum-successful-tests (first
st result-list))))
                    (setf sum-total-tests (+ sum-total-tests (second result
-list)))))))

          (update-test-results (the) sum-successful-tests sum-total-tests)
        )
      )
  )

  ;;-----
  ;; Method      : execute-test
  ;; Purpose     : Done when pressing "run selected test" btn
  ;;            : Executes the test or tests.
  ;; Arguments   : test-result (aunit-result-collection-class
  ;;            : or aunit-result-class)
  ;; Returns    :
  ;; Author     : Lars Barlindhaug
  ;;
  (define-method execute-single-test test-form-class (test-result)
    (if (typep test-result 'aunit-result-test-class)
        (progn
          (let ((test (find-test-from-result test-result))
                (result-list (execute-test (the) test))
                (successful-tests (first result-list))
                (total-tests (second result-list))
  
```



```

        (success-decimal (/ successful-tests total-tests)))

        (select-tree-item (the test-tree) (find-result-from-test test)
          (update-test-results (the) successful-tests total-tests)))
      (if (typep test-result 'aunit-result-collection-class)
        (execute-single-test (the) (the superior (:from test-result)))
        (execute-all-tests (the))))))

;;;-----
;;; Method      : execute-test
;;; Purpose     : Runs a test
;;; Arguments   : test (aunit-framework-class)
;;; Returns    : list with # of suc. tests and tot. tests.
;;; Author     : Lars Barlindhaug
;;;
(define-method execute-test test-form-class (test)
  (let ((test-script-path (the test-script-path (:from test)))
        (test-results (run-test test test-script-path))
        (successful-tests (the successful-tests (:from test-results)))
        (total-tests (the total-tests (:from test-results)))
        (success-decimal (if (not (equal total-tests 0))
                              (/ successful-tests total-tests)
                              0)
          )))
    (progn
      (the update?)
      (open-branch (the test-tree) (find-result-from-test test) :all? t)
      (set-test-result-image (the test-tree) (find-result-from-test test) ↵
    )

    (list successful-tests total-tests))
  )

;;;-----
;;; Method      : update-test-results
;;; Purpose     : Updates the test results with the
;;;              : number and % of successful tests
;;;              : and the total number of tests.
;;; Arguments   : # of successful tests
;;;              : # of tests
;;; Returns    : -
;;; Author     : Lars Barlindhaug
;;;
(define-method update-test-results test-form-class (successful-tests total-
l-tests)
  (let ((success-decimal (if (not (equal total-tests 0))
                              (/ successful-tests total-tests)
                              (progn
                                (print "ERROR divide by zero")
                                0)))
        (success? (if (equal successful-tests total-tests) 't)))
    (replace-text (the results-bar successful-field)
      (format 'nil "~d/~d (~d%)"
        successful-tests
        total-tests
        (round (* 100 success-decimal))))
    (replace-text (the results-bar result-field) (princ-to-string success ↵
?))

    (update-root-img (the test-tree) success?)
    (update-status-bar (the status-bar) success?))

```

```

)

;;;-----
;;; Method      : evaluate-results-tree
;;; Purpose     : Method is called when clicking on a
;;;             : test in the tree.
;;;             : TODO Move to test tree, change class?
;;; Arguments   : results (aunit-result-collection-class
;;;             : or aunit-result-class)
;;; Returns     : -
;;; Author      : Lars Barlindhaug
;;;
(define-method evaluate-results-tree test-form-class (results)
  (clear-results (!test-output))
  (evaluate-tree (the) results))

;;;-----
;;; Method      : evaluate-tree
;;; Purpose     : Recursively evaluates all the results
;;;             : belonging to the test and adds them to
;;;             : the text-box.
;;; Arguments   : results (aunit-result-collection-class
;;;             : or aunit-result-class)
;;; Returns     : -
;;; Author      : Lars Barlindhaug
;;;
(define-method evaluate-tree test-form-class (results)
  (let ((test-results (the tests (:from results))))
    (loop for test in test-results
      do
        (progn
          (if (typep test 'aunit-result-collection-class)
              (progn
                (evaluate-tree (the) test)))
            (if (typep test 'aunit-result-class)
                (progn
                  (add-results (!test-output) test)))))))
  )

;;;-----
;;; Method      : add-to-test-tree
;;; Purpose     : adds a test to the test tree in the GUI
;;;             :
;;; Arguments   : file-path to the test
;;; Returns     : -
;;; Author      : Lars Barlindhaug
;;;
(define-method add-to-test-tree test-form-class (file-path)
  (let ((name (remove-aml-file-extension (find-filename-in-path file-path
))))
    (if (test-already-in-tree? (read-from-string name) (tree-item-children
n (the test-tree root-object)))
        (format t "Test: ~a is already added to the window~%" name)
        (let ((test (add-object (the test-fws) (read-from-string name) 'aunit-
it-framework-class
                               :init-form (list 'test-script-path file-path
h
                                               'output-file-name "results"))))
          (set-up test)
          (the update?)
          (set-all-images (the test-tree) (find-result-from-test test) *gre

```

```

ey-img*)
  (select-tree-item (the test-tree) (the results))
  (open-branch (the test-tree) (the results))))
)

;;;-----
;;; Function   : right-adjustment
;;; Purpose    : Calculates the placement for adjusting
;;;            : gui components to the right-side of
;;;            : the window.
;;; Author     : Lars Barlindhaug
;;;
(defun right-adjustment (width side-margin)
  (- 100 width side-margin))

;;;-----
;;; Function   : set-up-gui
;;; Purpose    : Creates a name generator for the gui.
;;; Author     : Lars Barlindhaug
;;;
(defun set-up-gui ()
  (setf *gui-name-generator* (create-model 'name-generator :init-form (list
t 'auto-naming? t))))

;;;-----
;;; Function   : aunit
;;; Purpose    : starts aunit
;;; Author     : Lars Barlindhaug
;;;
(defun aunit ()
  (set-up-gui)
  (let ((form
        (or
         (the interface forms test-form-class (:error nil))
         (add-object (the interface forms) 'test-form-class 'test-form-c
lass))))
    (display form)
    (prepare-tree (the test-tree (:from form))
    )
  )
)

;;;-----
;;; Function   : start-aunit-dev
;;; Purpose    : starts a development version of aunit
;;;            : This allows for modifying aunit and
;;;            : running the new version
;;; Author     : Lars Barlindhaug
;;;
(defun start-aunit-dev ()
  (let ((form
        (add-object
         (the interface forms)
         (generate-name *gui-name-generator* 'test-win)
         'test-form-class)))
    (display form)
    (prepare-tree (the test-tree (:from form))
    )
  )
)

```

## **E.3 Print**

```
;;;-----  
;;; System   : :aunit-print  
;;; Purpose  : Printing tool for  
;;;           AML Unit testing framework  
;;;  
;;; Author   : Lars Barlindhaug  
;;;  
(define-system :aunit-print  
  :require-systems '(:aunit)  
  :files '(  
    "aunit-print-html.aml"  
    "aunit-print.aml"  
  )  
)
```

```
(in-package :aml)
```

```
;;;-----
;;; Function   : write-html-header
;;; Purpose    : writes the HTML header tags,
;;;            : date and starts the BODY and TABLE.
;;; Arguments  :
;;; Returns    :
;;; Author     : Lars Barlindhaug
;;;
(defun write-html-header (html-file)
  (format html-file "
<HTML>~%
<HEAD>~%
<style type=\"text/css\">~%
td {text-align:left; font-size:100%;}~%
td.fail {color:red;}~%
td.pass {color:green;}~%
td.header {text-align:center; font-size:120%; font-weight:bold;}~%
tr.heading {text-align:left; font-size:100%; font-weight:bold;}~%
tr.result {font-weight:bold;}~%
</style>~%
</HEAD>~%
<BODY>~%"

    (let ((date-list (today :values? t)))
      (format html-file "<p>~%~a ~a ~%</p>~%" (nth 1 date-list) (nth 0 date-
-list)))

    (format html-file "<table border=\"0\">~% ")

    (defvar *test-collection-header?* t))

;;;-----
;;; Function   : write-html-footer
;;; Purpose    : writes the AML info and then the
;;;            : end tags for BODY and HTML header tags.
;;; Arguments  :
;;; Returns    :
;;; Author     : Lars Barlindhaug
;;;
(defun write-html-footer (html-file)
  (format html-file "</table>~% ")

  (format html-file "<pre>~%~a~%~%</pre>~%~%" (report-aml-component-updat
es))

  (format html-file "</BODY>~% </HTML>"))

(defun write-test-collection-name (html-file result-collection)
  (format html-file "<tr> <td class=\"header\" colspan=\"4\">~a</td></tr>~%
~%"
    (the test-name (:from result-collection))))

(defun write-test-collection-header (html-file)
  (format
  html-file
  "<tr class=\"heading\"> <td>Result:</td> <td>Tested object:</td> <td>A
ctual value:</td> <td>Expected value:</td></tr>~%"))

(defun write-test-collection-footer (html-file result-collection)
  (format html-file "<tr class=\"result\"> ~a <td>~a</td> <td colspan=\"2\"
~%>Successful: ~d/~d (~,2f%) </td></tr>~%"))
```

```

    (get-pass-or-fail (the result (:from result-collection))
      (the test-name (:from result-collection))
      (the successful-tests (:from result-collection))
      (the total-tests (:from result-collection))
      (calculate-percentage (the total-tests (:from result-collection)
        ))
        (the successful-tests (:from result-collection))))
    (format html-file "<tr> <td colspan=\"4\">&nbsp;</td> </tr>~%")
  (defun write-test-result-html (html-file test-result)
    (format html-file "<tr> ~a <td>~a: ~a</td> <td>~a</td> <td>~a</td> </tr>~%
  >~%"
      (get-pass-or-fail (the result (:from test-result))
        (the type-of-test (:from test-result))
        (the tested-object (:from test-result))
        (the actual-value (:from test-result))
        (the expected-value (:from test-result))))
  (defun write-test-result-txt (txt-file test-result)
    (format txt-file "(~a ~a)~%"
      (the tested-object (:from test-result))
      (the actual-value (:from test-result))))
  (defun write-setup (html-file test-result)
    (format html-file "<tr>~a <td colspan=\"3\">~a</td> </tr>~%"
      "<td class=\"pass\">DONE</td>"
      test-result))

;;;-----
;;; Function   : get-pass-or-fail
;;; Purpose    : depending on result,
;;;            : returns a string PASS or FAIL.
;;; Arguments  : result (boolean)
;;; Returns    : PASS or FAIL (string)
;;; Author     : Lars Barlindhaug
;;;
  (defun get-pass-or-fail (result)
    (if result "<td class=\"pass\">PASS</td>" "<td class=\"fail\">FAIL</td>"
  ))

```

```

(in-package :aml)

(define-class aunit-print-class
  :inherit-from (object)
  :properties (
    folder-path (default "C:\\Data")
    output-folder-path (default (format 'nil
                                         "~a\\output"
                                         ^folder-path))

    output-file-name (default "results")
    html-output-file-path (default
                              (format 'nil
                                       "~a\\~a.html"
                                       ^output-folder-path
                                       ^output-file-name))

    txt-output-file-path (default
                           (format 'nil
                                    "~a\\~a.txt"
                                    ^output-folder-path
                                    ^output-file-name))

    pass-file-path (default (format 'nil
                                     "~a\\pass"
                                     ^output-folder-path))
  )
)

;;;-----
;;; Method      : print-results
;;; Purpose     : Prints the results to output html and
;;;              : txt files.
;;; Arguments   : results
;;; Returns    : -
;;; Author      : Lars Barlindhaug
;;;
(define-method print-results aunit-print-class (test-result)
  (progn
    (when (not (directory? !output-folder-path))
      (create-directory !output-folder-path))

    (file-delete (!pass-file-path))
    (file-delete (!html-output-file-path))
    (file-delete (!txt-output-file-path))

    (with-open-file (html-output-file
                    (the html-output-file-path) :direction :output)
      (write-html-header html-output-file)

      (with-open-file (results-txt-file
                    (the txt-output-file-path) :direction :output)
        (print-result-collection html-output-file results-txt-file test-r
          ↵
          esult))

      (write-html-footer html-output-file))

    (if (the result (:from test-result))
      (with-open-file (pass-file
                    (!pass-file-path) :direction :output)
        (format pass-file "PASS"))))
  )

;;;-----
;;; Function    : print-result-collection
;;; Purpose     : Loops over each

```



```

;;;           : aunit-result-collection-class
;;;           : and writes the results to the
;;;           : log file with HTML markup.
;;;           : Called by print-results,
;;;           : calls print-test-result.
;;; Arguments : results
;;; Returns   :
;;; Author    : Lars Barlundhaug
;;;
(defun print-result-collection (html-file txt-file result-collection)
  (unless *test-collection-header?*
    (progn
      (setf *test-collection-header?* t)))
  (write-test-collection-name html-file result-collection)

  (let ((all-tests
        (the tests (:from result-collection))))

    (loop for test-results in all-tests
          do
            (if (typep test-results 'aunit-result-collection-class)
                (print-result-collection html-file txt-file test-results)
                (print-test-result html-file txt-file test-results))))

    (loop for setup-cmd in (the setup-commands (:from result-collection))
          do
            (print-setup-command setup-cmd))

    (write-test-collection-footer html-file result-collection))

;;;-----
;;; Function  : print-test-result
;;; Purpose   : Loops over each aunit-results-class
;;;           : and writes the results to the
;;;           : log file with HTML markup.
;;;           : Called by print-result-collection.
;;; Arguments : results
;;; Returns   :
;;; Author    : Lars Barlundhaug
;;;
(defun print-test-result (html-file txt-file test-result)
  (select-model test-result)

  (if (typep test-result 'aunit-result-class)
      (progn
        (print-test-collection-header html-file)
        (write-test-result-html html-file test-result)
        (write-test-result-txt txt-file test-result))
      (progn
        (if (equal (type-of test-result) 'cons)
            (progn
              (print-test-collection-header html-file)
              (write-setup html-file test-result))))))

(defun print-test-collection-header (html-file)
  (if *test-collection-header?*
      (progn
        (write-test-collection-header html-file)
        (setf *test-collection-header?* nil))))

;;;-----
;;; Function  : calculate-percentage

```

```
;;; Purpose   : Finds the percentage of successful
;;;           : tests, when there are no tests it
;;;           : returns 100%.
;;; Arguments : total-tests (integer)
;;;           : successful-tests (integer)
;;; Returns   : percentage (float)
;;; Author    : Lars Barlundhaug
;;;
(defun calculate-percentage (total-tests successful-tests)
  (if (not (= 0 total-tests))
      (* (/
          successful-tests
          total-tests)
         100)
      100))
```

# Appendix F

## Bottle KBE model

Source and test code for the bottle model from section 5.3.1.

```

(define-class bottle-class
  :inherit-from (object)
  :properties (
    diameter (default 2.0)
    end-diameter (default 1.0)

    body-height (default 5.0)
    bottom-height (default 0.5)
    top-height (default 1.0)
  )
  :subobjects (
    (bottle-coordinate-system
     :class 'coordinate-system-class
     origin (list 1.0 0.0 0.0))

    (body :class 'open-cylinder-object
     diameter ^^diameter
     height ^^body-height
     orientation (list
                  (translate (list 0 0 0)))
     reference-coordinate-system
     ^^bottle-coordinate-system
     )

    (bottom :class 'cylinder-object
     diameter ^^diameter
     height ^^bottom-height
     orientation (list
                  (translate
                   (list
                    0
                    0
                    (-(+ (half (the superior superior body ↵
height))
                        (half ^height))))))
     reference-coordinate-system ^^bottle-coordinate-system
     )

    (top :class 'open-truncated-cone-object
     start-diameter ^^diameter
     end-diameter ^^end-diameter
     height ^^top-height

     reference-coordinate-system ^^bottle-coordinate-system

     orientation (list
                  (translate
                   (list
                    0
                    0
                    (+ (half (the superior superior body he↵
height))
                        (half ^height))))))
     )
  )
)

```

```

(defaunit "bottle-test"
  (deftest 'set-up
    (clear)
    (load "D:\\workspace_win\\src\\bottle\\src\\bottle.aml")
  )
  (deftest 'bottom-location-default
    (create-model 'bottle
      :class 'bottle-class)
    (check-list-diff
      (convert-coords (the bottle bottom) '(0 0 0)
        :from :local :to :global)
      (convert-coords (the bottle body) '(0 0 0)
        :from :local :to :global)
      (list 0.0 0.0 -2.75))
    )
  (deftest 'bottom-location
    (create-model 'bottle
      :class 'bottle-class
      :init-form (list
        'body-height 10.0
        'bottom-height 2.0))
    (check-list-diff
      (convert-coords (the bottle bottom) '(0 0 0)
        :from :local :to :global)
      (convert-coords (the bottle body) '(0 0 0)
        :from :local :to :global)
      (list 0.0 0.0 -6.0))
    )
  (deftest 'top-location-default
    (create-model 'bottle
      :class 'bottle-class)
    (check-list-diff
      (convert-coords (the bottle top) '(0 0 0)
        :from :local :to :global)
      (convert-coords (the bottle body) '(0 0 0)
        :from :local :to :global)
      (list 0.0 0.0 3.0))
    )
  (deftest 'top-location
    (create-model 'bottle
      :class 'bottle-class
      :init-form (list
        'body-height 10.0
        'top-height 2.0))
    (check-list-diff
      (convert-coords (the bottle top) '(0 0 0)
        :from :local :to :global)
      (convert-coords (the bottle body) '(0 0 0)
        :from :local :to :global)
      (list 0.0 0.0 6.0))
    )
  )
)

```





# Appendix G

## Beam KBE model

Source and test code for the beam model from section 5.3.2.

```

(define-class stud-class
  :inherit-from(series-object)
  :properties(
    space-between-studs (default 1.0)

    height (default 1.0)
    width (default 0.5)
    depth (default 0.5)

    class-expression 'box-object

    quantity (ceiling
              (/
                (+ (^beam-width
                   ^space-between-studs)
                  (+ ^width
                     ^space-between-studs)))

    space (/
            (-
              ^^beam-width
              (* ^quantity ^width))
            (- ^quantity 1))

    init-form '(
                orientation (list
                             (translate
                              (list
                               (+
                                (half ^width)
                                (* !index ^space)
                                (* !index ^stud-width))
                               (+
                                (half ^height)
                                (^beam-height))
                               0))))
              )
  :subobjects()
)

(define-class beam-tdd-class
  :inherit-from(object)
  :properties(
    beam-height (default 0.1)
    beam-width (default 10)
    beam-depth (default 0.5)

    stud-height (default 1.0)
    stud-width (default 0.5)
    stud-depth (default 0.5)

    space-between-studs (default 1.0)
  )
  :subobjects(
    (main-beam :class 'box-object
               height ^beam-height
               width ^beam-width
               depth ^beam-depth)

    (stud :class 'stud-class
           height ^stud-height
           width ^stud-width
           depth ^stud-depth)
  )
)

```



```

(defaunit "beam-test"
  (deftest 'set-up
    (clear)
    (load "D:\\workspace_win\\src\\beam\\beam-tdd.aml")
    (create-model 'beam
      :class 'beam-tdd-class
      :init-form (list
        'beam-width 5
        'beam-height 0.1
        'space-between-studs 1
        'stud-width 1
        )
      )

    (create-model 'beam6
      :class 'beam-tdd-class
      :init-form (list
        'beam-width 6
        'beam-height 0.2
        'space-between-studs 1
        'stud-width 1
        )
      )
    )

  (deftest 'stud-quantity-len5
    (check-equals
      (the beam stud quantity)
      3)
    )

  (deftest 'stud-quantity-len6
    (check-equals
      (the beam6 stud quantity)
      4)
    )

  (deftest 'stud-space-len5
    (check-equals
      (the beam stud space)
      1)
    )

  (deftest 'stud-space-len6
    (check-equals
      (the beam6 stud space)
      (/ 2 3))
    )

  (deftest 'stud-position-len5-first-x
    (check-equals
      (first (convert-coords
        (the beam stud stud-0000)
        '(0 0 0) :from :local :to :global))
      0.5)
    )

  (deftest 'stud-position-len5-first-y
    (check-equals
      (second (convert-coords
        (the beam stud stud-0000)
        '(0 0 0) :from :local :to :global))
      0.6)
    )

  (deftest 'stud-position-len5-first-z
    (check-equals
      (third (convert-coords

```

```

        (the beam stud stud-0000)
        '(0 0 0) :from :local :to :global))
    0.0)
)

(deftest 'stud-position-len5-second-x
  (check-equals
    (first (convert-coords
            (the beam stud stud-0001)
            '(0 0 0) :from :local :to :global))
    2.5)
)

(deftest 'stud-position-len5-last-x
  (check-equals
    (first (convert-coords
            (the beam stud stud-0002)
            '(0 0 0) :from :local :to :global))
    4.5)
)

(deftest 'stud-position-len6-first-x
  (check-equals
    (first (convert-coords
            (the beam6 stud stud-0000)
            '(0 0 0) :from :local :to :global))
    0.5)
)

(deftest 'stud-position-len6-first-y
  (check-equals
    (second (convert-coords
            (the beam6 stud stud-0000)
            '(0 0 0) :from :local :to :global))
    0.7)
)

(deftest 'stud-position-len6-first-z
  (check-equals
    (third (convert-coords
            (the beam6 stud stud-0000)
            '(0 0 0) :from :local :to :global))
    0.0)
)

(deftest 'stud-position-len6-second-x
  (check-delta
    (first (convert-coords
            (the beam6 stud stud-0001)
            '(0 0 0) :from :local :to :global))
    ;Stud w, space, half stud w
    (+ 1 (/ 2 3) 0.5)
    0.0001)
)

(deftest 'stud-position-len6-last-x
  (check-equals
    (first (convert-coords
            (the beam6 stud stud-0003)
            '(0 0 0) :from :local :to :global))
    5.5)
)
)

```

# Appendix H

## Bookshelf KBE model

### H.1 Source code

Source code for the KBE bookshelf model [30]. Developed by Aker Solutions KBe-Design and later edited by the author.

```
;;;Filename: system.def

(in-package :AML)

(define-system :bookshelf-system
  :files '(
    "kbe-bookshelf-input-mixin.aml "
    "kbe-bookshelf-data-model-class.aml "
    "kbe-bookshelf-class.aml "
    "kbe-bookshelf-frame-class.aml "
    "kbe-bookshelf-wall-series-class.aml "
    "kbe-bookshelf-shelf-series-class.aml "
  )
)
```

```
;;;Filename: kbe-bookshelf-input-mixin.aml

(in-package :AML)

(define-class kbe-bookshelf-input-mixin
  :inherit-from (object)
  :properties (
    ;; parameters set in GUI
    height-input 5
    width-input 3
    max-hs-input 0.5
    vertical-spacing-shelves-input 0.5
    shelf-depth 0.7
    thickness-bottom-shelf-input 0.05
    thickness-top-shelf-input 0.05
    thickness-dividing-walls-input 0.05
    thickness-of-shelves-input 0.05
    thickness-side-walls-input 0.05
  )
  :subobjects (
  )
)
```

```

;;;Filename: kbe-bookshelf-data-model-class.aml
(in-package :AML)

(define-class kbe-bookshelf-data-model-class
  :inherit-from (kbe-bookshelf-input-mixin data-model-node-mixin)
  :properties (
    property-objects-list (list
      (list (the superior height-input self) '(
(automatic-apply? t))
      (list (the superior width-input self) '(
(automatic-apply? t))
      (list (the superior max-hs-input self) '(
(automatic-apply? t))
      (list (the superior vertical-spacing-shelves-input self) '(automatic-apply? t))
      (list (the superior thickness-bottom-shelf-input self) '(automatic-apply? t))
      (list (the superior thickness-top-shelf-input self) '(automatic-apply? t))
      (list (the superior thickness-dividing-walls-input self) '(automatic-apply? t))
      (list (the superior thickness-side-walls-input self) '(automatic-apply? t))
      (list (the superior thickness-of-shelves-input self) '(automatic-apply? t))
    )

;;;Properties:
  (height-input :class '(editable-data-property-class change-event)
    t)
    after-change (kbe-validate-bookshelf-height ^superior)
    formula :inherit-formula
    label "Bookshelf height"
  )

  (width-input :class '(editable-data-property-class change-event)
    )
    after-change (kbe-validate-bookshelf-width ^superior)
    formula :inherit-formula
    label "Bookshelf width"
  )

  (max-hs-input :class '(editable-data-property-class change-event)
    t)
    after-change (kbe-validate-bookshelf-width ^superior)
    formula :inherit-formula
    label "Maximum horizontal length of one shelf"
  )

  (vertical-spacing-shelves-input :class '(editable-data-property-class change-event)
    after-change (kbe-validate-bookshelf-height ^superior)
    formula :inherit-formula
    label "Vertical spacing between shelves"
  )

  (thickness-bottom-shelf-input :class 'editable-data-property-class
    ass
    formula :inherit-formula
    label "Thickness bottom shelf"
  )

  (thickness-top-shelf-input :class 'editable-data-property-class
    formula :inherit-formula
    label "Thickness top shelf"
  )

```

```

class      (thickness-dividing-walls-input :class 'editable-data-property-clas
           formula :inherit-formula
           label "Thickness of dividing walls"
           )

s          (thickness-side-walls-input :class 'editable-data-property-clas
           formula :inherit-formula
           label "Thickness side walls"
           )

s          (thickness-of-shelves-input :class 'editable-data-property-clas
           formula :inherit-formula
           label "Thickness of shelves"
           )
)

;;;-----
;;;Method for verification of width-input and max-hs-input
;;;-----

(define-method kbe-validate-bookshelf-width kbe-bookshelf-data-model-clas
s ()
  (if (< !width-input (* 0.5 (!max-hs-input)))
      (pop-up-message "WRONG INPUT PARAMETERS: The bookshelf is too narrow. Adjust bookshelf width or maximum horizontal length of one shelf. ")
      nil
  )
)

;;;-----
;;;Method for verification of height-input and vs-input
;;;-----

(define-method kbe-validate-bookshelf-height kbe-bookshelf-data-model-cl
ass ()
  (if (> !vertical-spacing-shelves-input !height-input)
      (pop-up-message "WRONG INPUT PARAMETERS: The bookshelf is too low for even one vertical space in the bookshelf. Adjust bookshelf height or vertical spacing between shelves. ")
      nil
  )
)

```

```

;;;Filename: kbe-bookshelf-class.aml
(in-package :AML)

(define-class kbe-bookshelf-class
  :inherit-from (kbe-bookshelf-data-model-class)
  :properties (
    height-for-shelves (- ^height-input
                          (+ ^thickness-bottom-shelf-input
                              ^thickness-top-shelf-input))

    number-of-dividing-walls (if
                              (> ^width-input ^max-hs-input)
                              (ceiling (/ ^width-input ^max-hs-inp
ut))
                              0)

    number-of-shelves (if
                       (>
                        (* 2 ^vertical-spacing-shelves-inpu
t)
                        ^height-for-shelves)
                       0
                       (- (floor
shelves-input))
                          (/ ^height-for-shelves ^vertical-spacing-
                              1))

    shelf-length (/
                  (- ^width-input
                    (+ (* 2 ^thickness-side-walls-input)
                      (*
                       ^thickness-dividing-walls-input
                       ^number-of-dividing-walls)))
                  (+ ^number-of-dividing-walls 1))

  )

  :subobjects (
    (bookshelf-frame :class 'kbe-bookshelf-frame-class

    )

    (bookshelf-walls :class (if (equal !number-of-dividing-walls 0)
                                'null-object
                                'kbe-bookshelf-wall-series-class)

    )

    (bookshelf-shelves :class (if (equal !number-of-shelves 0)
                                   'null-object
                                   'kbe-bookshelf-shelf-series-class)

    )
  )
)

```



```

;;;Filename: kbe-bookshelf-frame-class.aml
(in-package :AML)

(define-class kbe-bookshelf-frame-class
  :inherit-from (object)
  :properties (
    vertical-offset (* 0.5
      (-
        (* 0.5 ^^thickness-bottom-shelf-input)
        (* 0.5 ^^thickness-top-shelf-input)))
  )

  :subobjects (
    (side-1 :class 'box-object
      depth ^^shelf-depth
      width ^^thickness-side-walls-input
      height ^^height-for-shelves
      orientation (list
        (translate (list
          (+ (* -0.5 ^^width-input) (* 0.5 ^width))
          (-
            ^^thickness-bottom-shelf-input
            ^^thickness-top-shelf-input)
          0))
        )
      reference-coordinate-system nil
    )

    (side-2 :class 'box-object
      depth ^^shelf-depth
      width ^^thickness-side-walls-input
      height ^^height-for-shelves
      orientation (list
        (translate (list
          (- (* 0.5 ^^width-input) (* 0.5 ^width))
          ^^vertical-offset
          0))
        )
      reference-coordinate-system nil
    )

    (top-shelf :class 'box-object
      depth ^^shelf-depth
      width ^^thickness-top-shelf-input
      height ^^width-input
      orientation (list
        (translate (list
          (- (* 0.5 ^^height-input)
            (* 0.5 ^^thickness-top-shelf-input))
          0
          0))
        (rotate 90 '(0 0 1))
        )
      reference-coordinate-system nil
    )

    (bottom-shelf :class 'box-object
      depth ^^shelf-depth
      width ^^thickness-bottom-shelf-input
      height ^^width-input
      orientation (list
        (translate (list
          (+
            (* -0.5 ^^height-input)
            (* 0.5 ^^thickness-bottom-shelf-input))
          0
          0))
        )
      reference-coordinate-system nil
    )
  )
)

```

```
(rotate 90 '(0 0 1))  
)  
reference-coordinate-system nil  
) )
```

```
;;;Filename: kbe-bookshelf-wall-series-class.aml
```

```
(in-package :AML)
```

```
(define-class kbe-bookshelf-wall-series-class
  :inherit-from (series-object)
  :properties (
    quantity ^^number-of-dividing-walls
    reference-coordinate-system (the superior superior bookshelf-fr
ame side-1)
    wall-direction (list 1 0 0)
    shelf-length-list-1 (let (
      (a (list (+
        (* 0.5 ^^thickness-side-walls-input)
        ^^shelf-length
        (* 0.5 ^^thickness-dividing-walls-input))))
      (var
        (+ ^^shelf-length ^^thickness-dividing-walls-input))
      )
      (loop for i from 1 to (- ^quantity 1)
        do
          (push var a)) a
      )
    shelf-length-list (reverse ^shelf-length-list-1)

    wall-coords (loop for i from 0 to (- ^quantity 1)
      sum (nth i ^shelf-length-list) into dist-sum
      collect (multiply-vector-by-scalar ^wall-direction dist-su
m)
      )

    class-expression 'box-object

    init-form '(
      depth ^^shelf-depth
      width ^^thickness-dividing-walls-input
      height (the height
        (:from
          (the superior superior superior bookshelf-frame sid
e-1)))

      orientation (list
        (translate (nth !index ^wall-coords))
        )
      reference-object (the superior superior superior bookshelf-fr
ame side-1)
      parent-coordinate-system ^^reference-coordinate-system
    )
    :subobjects (
    )
  )
```

```

;;;Filename: kbe-bookshelf-shelf-series-class.aml
(in-package :AML)

(define-class kbe-bookshelf-shelf-series-class
  :inherit-from (series-object)
  :properties (
    reference-coordinate-system (the superior superior bookshelf-fr
ame top-shelf)
    wall-direction (list 1 0 0)
    shelf-height-list-1 (let (
      (a
        (list
          (* -1 (+
            (- ^^vertical-spacing-shelves-input
              (half ^^thickness-of-shelves-input))
            (half ^^thickness-top-shelf-input))))))
      (var (* -1
        ^^vertical-spacing-shelves-input))
      )
      (loop for i from 2 to ^^number-of-shelves do (push var
        a))
      shelf-height-list (reverse ^shelf-height-list-1)

shelf-coords (loop for i from 0 to (- (length ^shelf-height-li
st) 1)
  sum (nth i ^shelf-height-list) into dist-sum
  collect (multiply-vector-by-scalar ^wall-direction dist-su
m)
  )
  quantity (if
    (<
      (- ^^height-input
        ^^thickness-bottom-shelf-input
        (* -1 (nth 0 (nth (- ^^number-of-shelves 1) ^
shelf-coords)))
        (* 0.5 ^^thickness-of-shelves-input))
      ^^vertical-spacing-shelves-input)
      (- ^^number-of-shelves 1)
      ^^number-of-shelves)

class-expression 'box-object

init-form '(
  depth ^^shelf-depth
  width ^^thickness-dividing-walls-input
  height (the height (:from (the superior superior superior boo
kshelf-frame top-shelf)))
  orientation (list
    (translate (nth !index ^shelf-coords))
    )
  reference-object (the superior superior superior bookshelf-fr
ame top-shelf)
  parent-coordinate-system ^^reference-coordinate-system
  )
  )
  :subobjects (
  ))

```

## H.2 Test code

Test code for the KBE bookshelf model.

05/09/12

tests.de ↵

kbe-bookshelf-test.aml  
kbe-bookshelf-wall-series-class-test.aml  
kbe-bookshelf-shelf-series-class-test.aml  
kbe-bookshelf-frame-class-test.aml  
kbe-bookshelf-class-test.aml

```
(defaunit "bookshelf-test"
  (deftest 'set-up
    (clear)
    (load-system :bookshelf-system)
  )

  (deftest 'height
    (create-model 'bookshelf
      :class 'kbe-bookshelf-class
      :init-form '(height-input 5
                    max-hs-input 3))

    (check-equals
      (+
        (the bookshelf bookshelf-frame side-1 height)
        (the bookshelf bookshelf-frame top-shelf width)
        (the bookshelf bookshelf-frame bottom-shelf width))
      (the bookshelf height-input)))

    (deftest 'height-big-top-bottom
      (create-model 'bookshelf
        :class 'kbe-bookshelf-class
        :init-form '(height-input 5
                      thickness-bottom-shelf-input 1
                      thickness-top-shelf-input 0.5
                      max-hs-input 3))

      (check-equals
        (+
          (the bookshelf bookshelf-frame side-1 height)
          (the bookshelf bookshelf-frame top-shelf width)
          (the bookshelf bookshelf-frame bottom-shelf width))
        (the bookshelf height-input)))

    (deftest 'width
      (create-model 'bookshelf
        :class 'kbe-bookshelf-class
        :init-form '(width-input 5
                      max-hs-input 3))

      (check-equals
        (the bookshelf bookshelf-frame top-shelf height)
        (the bookshelf width-input)
      )

      (check-equals
        (the bookshelf bookshelf-frame bottom-shelf height)
        (the bookshelf width-input)))

  )
```

```

(defaunit "bookshelf-class-test"
  (deftest 'set-up
    (clear)
    (load-system :bookshelf-system)
  )

  (deftest 'height-for-shelves
    (create-model 'bookshelf
      :class 'kbe-bookshelf-class
      :init-form '(height-input 5
                    thickness-bottom-shelf-input 1
                    thickness-top-shelf-input 1))
    (check-equals (the bookshelf height-for-shelves) 3))

  (deftest 'number-of-dividing-walls-width-less-than-max-hs
    (create-model 'bookshelf
      :class 'kbe-bookshelf-class
      :init-form '(width-input 2 max-hs-input 3))
    (check-equals (the bookshelf number-of-dividing-walls) 0)
  )

  (deftest 'number-of-dividing-walls
    (create-model 'bookshelf
      :class 'kbe-bookshelf-class
      :init-form '(width-input 5 max-hs-input 2))
    (check-equals (the bookshelf number-of-dividing-walls) 3)
  )

  ;;2*2 > 3.99999
  (deftest 'number-of-shelves-no-shelf
    (create-model 'bookshelf
      :class 'kbe-bookshelf-class
      :init-form '(vertical-spacing-shelves-input 2
                    height-input 3.9999))
    (check-equals (the bookshelf number-of-shelves) 0)
  )

  ;;2*1 + 0.5 + 0.5 > 2.9
  (deftest 'number-of-shelves-no-shelf-big-top-bottom
    (create-model 'bookshelf
      :class 'kbe-bookshelf-class
      :init-form '(vertical-spacing-shelves-input 1
                    height-input 2.9
                    thickness-bottom-shelf-input 0.5
                    thickness-top-shelf-input 0.5))
    (check-equals (the bookshelf number-of-shelves) 0)
  )

  (deftest 'number-of-shelves
    (create-model 'bookshelf
      :class 'kbe-bookshelf-class
      :init-form '(vertical-spacing-shelves-input 2
                    height-input 9))
    (check-equals (the bookshelf number-of-shelves) 3)
  )

  (deftest 'number-of-shelves-2
    (create-model 'bookshelf
      :class 'kbe-bookshelf-class
      :init-form '(thickness-top-shelf-input 1
                    thickness-bottom-shelf-input 1
                    height-input 9
                    vertical-spacing-shelves-input 2
                    thickness-of-shelves-input 0.5))
    (check-equals (the bookshelf number-of-shelves) 2))

```



```

(deftest 'number-of-shelves-big-top-bottom
  (create-model 'bookshelf
    :class 'kbe-bookshelf-class
    :init-form '(vertical-spacing-shelves-input 1
                  height-input 5
                  thickness-bottom-shelf-input 0.5
                  thickness-top-shelf-input 0.5))
  (check-equals (the bookshelf number-of-shelves) 3)
  )

(deftest 'number-of-shelves-big-lower-shelf
  (create-model 'bookshelf
    :class 'kbe-bookshelf-class
    :init-form '(vertical-spacing-shelves-input 1
                  height-input 5.9
                  thickness-bottom-shelf-input 0.5
                  thickness-top-shelf-input 0.5))
  (check-equals (the bookshelf number-of-shelves) 3)
  )

(deftest 'shelf-length
  (create-model 'bookshelf
    :class 'kbe-bookshelf-class
    :init-form '(width-input 10
                  thickness-side-walls-input 1
                  thickness-dividing-walls-input 0.2
                  max-hs-input 2))
  (check-equals (the bookshelf number-of-dividing-walls) 5)
  (check-equals (the bookshelf shelf-length) (/ 7 6))
  )

(deftest 'bookshelf-walls-null-object
  (create-model 'bookshelf
    :class 'kbe-bookshelf-class
    :init-form '(width-input 2 max-hs-input 3))
  (check-equals (type-of (the bookshelf bookshelf-walls)) 'null-object)
  )

(deftest 'bookshelf-walls-kbe-object
  (create-model 'bookshelf
    :class 'kbe-bookshelf-class
    :init-form '(width-input 7 max-hs-input 3))
  (check-equals (type-of (the bookshelf bookshelf-walls)) 'kbe-bookshelf-wall-series-class)
  )

(deftest 'bookshelf-shelves-null-object
  (create-model 'bookshelf
    :class 'kbe-bookshelf-class
    :init-form '(vertical-spacing-shelves-input 2
                  height-input 3.9))
  (check-equals (the bookshelf number-of-shelves) 0)
  (check-equals (type-of (the bookshelf bookshelf-shelves)) 'null-object)
  )

(deftest 'number-of-shelves-kbe-object
  (create-model 'bookshelf
    :class 'kbe-bookshelf-class
    :init-form '(vertical-spacing-shelves-input 2
                  height-input 9))
  (check-equals (type-of (the bookshelf bookshelf-shelves))
    'kbe-bookshelf-shelf-series-class)
  )

```

```

(defaunit "bookshelf-frame-class-test"
  (deftest 'set-up
    (clear)
    (load-system :bookshelf-system)
  )

  (deftest 'validate-bookshelf-width
    (create-model 'bookshelf
      :class 'kbe-bookshelf-class
      :init-form '(thickness-bottom-shelf-input 2
                    thickness-top-shelf-input 1))

    (check-equals
      (the bookshelf bookshelf-frame vertical-offset)
      0.25)
  )

  (deftest 'side-1-height
    (create-model 'bookshelf
      :class 'kbe-bookshelf-class
      :init-form '(height-input 10
                    thickness-top-shelf-input 1
                    thickness-bottom-shelf-input 2))

    (check-equals (the bookshelf bookshelf-frame side-1 height) 7)
  )

  (deftest 'side-1-orientation
    (create-model 'bookshelf
      :class 'kbe-bookshelf-class
      :init-form '(width-input 10
                    thickness-side-walls-input 1
                    thickness-top-shelf-input 1
                    thickness-bottom-shelf-input 2))

    (check-equals
      (first (the bookshelf bookshelf-frame side-1 position))
      (list -4.5 1.0 0.0))
  )

  (deftest 'side-2-height
    (create-model 'bookshelf
      :class 'kbe-bookshelf-class
      :init-form '(height-input 10
                    thickness-top-shelf-input 1
                    thickness-bottom-shelf-input 2))

    (check-equals (the bookshelf bookshelf-frame side-2 height) 7)
  )

  (deftest 'side-2-orientation
    (create-model 'bookshelf
      :class 'kbe-bookshelf-class
      :init-form '(width-input 10
                    thickness-side-walls-input 1
                    thickness-top-shelf-input 1
                    thickness-bottom-shelf-input 2))

    (check-equals
      (first (the bookshelf bookshelf-frame side-2 position))
      (list 4.5 0.25 0.0))
  )

  (deftest 'top-shelf-orientation
    (create-model 'bookshelf
      :class 'kbe-bookshelf-class
      :init-form '(height-input 10
                    thickness-top-shelf-input 1))

    (check-list-delta
      '(0.0 4.5 0.0)
      (first (the bookshelf bookshelf-frame top-shelf position))
      0.000001)
  )

```

```
)  
(deftest 'bottom-shelf-orientation  
  (create-model 'bookshelf  
    :class 'kbe-bookshelf-class  
    :init-form '(height-input 10  
                thickness-bottom-shelf-input 1))  
  (check-list-delta  
    '(0.0 -4.5 0.0)  
    (first (the bookshelf bookshelf-frame bottom-shelf position))  
    0.000001)  
  )  
)
```

```
(defaunit "bookshelf-shelf-series-class-test"
  (deftest 'set-up
    (clear)
    (load-system :bookshelf-system)
  )

  (deftest 'shelf-length-list
    (create-model 'bookshelf
      :class 'kbe-bookshelf-class
      :init-form '(thickness-side-walls-input 1
                    thickness-dividing-walls-input 0.1
                    width-input 4
                    max-hs-input 1
                    vertical-spacing-shelves-input 2
                    thickness-of-shelves-input 0.5))

    (check-equals (the bookshelf shelf-length) 0.32)
    (check-equals (the bookshelf number-of-dividing-walls) 4)

    (check-list-delta
      (the bookshelf bookshelf-walls shelf-length-list)
      '(0.87 0.42 0.42 0.42)
      0.000001)
    )

  (deftest 'wall-coords
    (create-model 'bookshelf
      :class 'kbe-bookshelf-class
      :init-form '(thickness-side-walls-input 1
                    thickness-dividing-walls-input 0.1
                    width-input 4
                    max-hs-input 1
                    vertical-spacing-shelves-input 2
                    thickness-of-shelves-input 0.5))

    (check-list-delta
      (first (the bookshelf bookshelf-walls wall-coords))
      '(0.87 0 0)
      0.000001)

    (check-list-delta
      (second (the bookshelf bookshelf-walls wall-coords))
      '(1.29 0 0)
      0.000001)

    (check-list-delta
      (third (the bookshelf bookshelf-walls wall-coords))
      '(1.71 0 0)
      0.000001)

    (check-list-delta
      (fourth (the bookshelf bookshelf-walls wall-coords))
      '(2.13 0 0)
      0.000001)
    )
  )
)
```

```

(defaunit "bookshelf-shelf-series-class-test"
  (deftest 'set-up
    (clear)
    (compile-system :bookshelf-system)
  )

  (deftest 'shelf-height-list
    (create-model 'bookshelf
      :class 'kbe-bookshelf-class
      :init-form '(thickness-top-shelf-input 1
                    thickness-bottom-shelf-input 1
                    height-input 9
                    vertical-spacing-shelves-input 2
                    thickness-of-shelves-input 0.5))

    (check-equals (the bookshelf number-of-shelves) 2)

    (check-equals (the bookshelf bookshelf-shelves shelf-height-list)
      '(-2.25 -2))
  )

  (deftest 'shelf-coords
    (create-model 'bookshelf
      :class 'kbe-bookshelf-class
      :init-form '(thickness-top-shelf-input 1
                    thickness-bottom-shelf-input 1
                    height-input 9
                    vertical-spacing-shelves-input 2
                    thickness-of-shelves-input 0.5))

    (check-list-delta (first (the bookshelf bookshelf-shelves shelf-coord
s))
      '(-2.25 0 0)
      0.0000001)
    (check-list-delta (second (the bookshelf bookshelf-shelves shelf-coord
ds))
      '(-4.25 0 0)
      0.0000001)
  )
)

```



# Appendix I

## Luva Spar test code

Test code for the Luva spar model.

```

(defaunit "kbe-spar-space-manager-test"

  (deftest 'set-up
    (create-model 'kbe-spar-space-manager-class))

  (deftest 'label-manager-test
    (check-equals
      (get-plane-by-label
        (the kbe-spar-space-manager-class)
        "not-existing")
      'nil))

  (deftest 'label-manager-x-min
    (check-equals
      (the label (:from (get-plane-by-label
                          (the kbe-spar-space-manager-class)
                          "x-min"))))
      "x-min"))

  (deftest 'label-manager-x-min-type
    (check-equals
      (type-of (get-plane-by-label
                 (the kbe-spar-space-manager-class)
                 "x-min"))
      'oda-datum-plane-class))

  (deftest 'label-manager-test-x-min-origin
    (check-equals
      (third (the origin (:from (get-plane-by-label
                                  (the kbe-spar-space-manager-class)
                                  "x-min"))))
      -50))

  (deftest 'label-manager-x-min-origin-convert-to-global
    (check-list-delta
      (convert-coords
        (the reference-coordinate-system
          (:from (get-plane-by-label
                  (the kbe-spar-space-manager-class)
                  "x-min"))))
        (the origin (:from (get-plane-by-label
                            (the kbe-spar-space-manager-class)
                            "x-min"))))
      '(-50 0 50)
      0))

  (deftest 'label-manager-x-min-origin-convert-to-global-change-value
    (change-value (the origin (:from (the kbe-spar-space-manager-class))
                              '(10 10 10))
      (check-list-delta
        (convert-coords
          (the reference-coordinate-system
            (:from (get-plane-by-label
                    (the kbe-spar-space-manager-class)
                    "x-min"))))
          (the origin (:from (get-plane-by-label
                              (the kbe-spar-space-manager-class)
                              "x-min"))))
          '(-40 10 60)
          0))

  (deftest 'label-manager-x-min-origin-convert-to-global-change-value
    (change-value (the origin (:from (the kbe-spar-space-manager-class))
                              '(10 10 10))
      (check-list-delta
        (convert-coords
          (the reference-coordinate-system
            (:from (get-plane-by-label
                    (the kbe-spar-space-manager-class)
                    "x-min"))))
          (the origin (:from (get-plane-by-label
                              (the kbe-spar-space-manager-class)
                              "x-min"))))
          '(-40 10 60)
          0))

```



```
        (the kbe-spar-space-manager-class)
        "x-min"))))
  (the origin (:from (get-plane-by-label
                      (the kbe-spar-space-manager-class)
                      "x-min"))))
  '(-40 10 60)
  0))

(deftest 'origin-default
  (check-equals
   (the origin (:from (the kbe-spar-space-manager-class)))
   '(0 0 0)))

(deftest 'origin-change
  (change-value (the origin
                  (:from (the kbe-spar-space-manager-class)))
                '(10 10 10))
  (check-equals
   (the origin (:from (the kbe-spar-space-manager-class)))
   '(10 10 10)))
  )
```