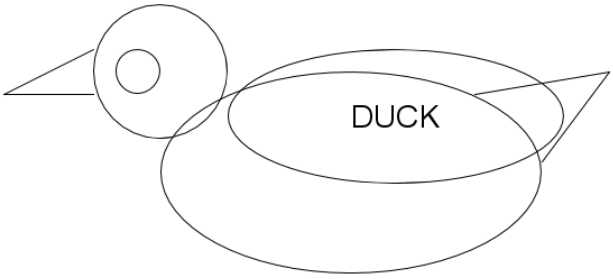
DUCK – A Use Case Assistant



Final Year Dissertation

Author: Artemiy Stepanov

Supervisor: Professor Andrew Ireland

BSc (Hons) Computer Science

Declaration

I, Artemiy Stepanov, confirm that this work submitted for assessment is my own and is expressed in my own words. Any uses made within it of the works of other authors in any form (e.g., ideas, equations, figures, text, tables, programs) are properly acknowledged at any point of their use. A list of the references employed is included.

Signed: 

Date: 15/04/2024

Abstract

Requirements analysis always lies at the start of the project lifecycle for any system. Completeness is vital but may be difficult to achieve as many nuances can go overlooked. In software engineering, UML use case diagrams and scenarios are frequently used to describe user-system interaction. However, there may be inconsistencies between the diagrams and scenarios, as well as within scenario flows.

While a standard exists for use case diagram drawing, there is not an officially recognised one for use case scenarios, so this is one of the causes for errors in models. Existing requirements engineering and use case modelling tools also have limited functionality when it comes to model validation. The aim of this project is to create a “Detailed Use Case Kit” (DUCK) – a methodology and an assistant application which would help to prevent or point out these inconsistencies.

Acknowledgements

I would like to thank Professor Andrew Ireland for guiding and encouraging me throughout the project.

Contents

Table of Abbreviations

Abbreviation	**Definition**
CMS	Credential Management System
DUCK	Detailed Use Case Kit
EARS	Easy Approach to Requirements Syntax
FR	Functional Requirement
GORE	Goal-Oriented Requirements Engineering
GUI	Graphical User Interface
IDE	Integrated Development Environment
KAOS	Keep All Objectives Satisfied
KB	Knowledge Base
MPL	Mars Polar Lander
NFR	Non-Functional Requirement
PIN	Personal Identification Number
SMS	Security Monitoring System
SSSS	Secure Self Storage System
SUS	System Usability Scale
UML	Unified Modelling Language

1 Introduction

1.1 Motivation

Any system is designed for a purpose. The process of software design begins with the requirements engineering – identifying why the system is being designed, how it will benefit the stakeholders, and how it will achieve the desired result. Failure to properly identify and define requirements before system design and implementation starts may severely impact the quality of the final product. Many things can be misunderstood or go overlooked, especially by inexperienced teams. If caught during the development process these mistakes will cause a delay and an overhead of redesigning the system, but if slipped unnoticed into a released product can lead to financial loss, legal trouble, safety violations and at worst loss of life. A prime example of such failure is NASA's loss of its Mars Polar Lander costing 165 million US dollars (McQuaid, P.A., 2012).

Launched 3 January 1999, the NASA Mars Polar Lander entered the Martian atmosphere on 3 December (Albee et al., 2000). Communication seized during the descent, by design the MPL would land autonomously. However, attempts to resume communication with the Lander after expected touchdown time failed, finally concluding on 17 January 2000 at which point the Lander was declared as a loss. According to the review board, the most probable cause for loss of mission was the software misinterpreting vibrations from deployment of its stowed legs as touchdown, so the landing engines were shut down prematurely and the MPL plummeted from 40 metres altitude.

Though the immediate reason for this failure was a single line of code, this poor software implementation was in turn a result of inadequate requirements analysis and elicitation in the beginning of the project (McQuaid, P.A., 2012). Funding constraints, tight deadline, lack of peer review and domain expert involvement lead to the key technical decisions in MPL's software design which proved to be wrong (Albee et al., 2000). Had the project team followed a better practice in the formulation phase, this error could have been caught before the implementation phase started.

Requirements can be either functional or non-functional – what the system should do and how it should do it. While many requirements engineering practices exist, there is no definitive standard. A popular way to represent functional requirements is UML use case modelling. Use case diagrams focus on how users interact with the system and what value it brings to the stakeholders (Jacobson et al.,2011). The diagrams are supported by a scenario which gives an interaction's goal, conditions, participants, and the steps of the process.

UML is a standard of the Object Management Group who has a detailed specification on drawing use case diagrams, but not for writing the scenarios (OMG, 2017). This project will explore the potential of standardised use case modelling as a foundation for requirements analysis, specification, and validation.

1.2Aims

This paper will document the creation of the Detailed Use Case Kit or DUCK – a methodology and an assistant application for it. The aims of this project are:

- To mechanise the process of requirements engineering by developing an application that would assist with formulating requirements, drawing use case diagrams, writing use case scenarios, and checking the consistency between and within the three.
- To create as standard for representing requirement, diagram-scenario, and scenario flow logic of use case models. This will be both a logical structure and a digital file format used to record it.
- To make the methodology intuitive and the supporting application's GUI convenient so that they can be utilised easily and effectively.

1.3 Objectives

Objective 1: To conduct a review of literature relating to requirements engineering and use case modelling.

Objective 2: To research existing software tools that assist with requirements engineering and use case modelling.

Objective 3: To identify techniques and features from the researched approaches and software tools that should be included.

Objective 4: To define a set of requirements for the assistant application.

Objective 5: To identify a suitable platform for developing the application.

Objective 6: To implement the use case assistant application.

Objective 7: Conduct thorough testing of the application with case scenarios.

Objective 8: Conduct user testing to see if the application's GUI is fit for its purpose.

1.4 Report Organisation

The second chapter will give a review of requirements engineering approaches, software tools, how they compare. This will help identify the principles and features necessary for the DUCK methodology and software. The third chapter will give the use case scenarios and a list of requirements for the assistant application, the choice of tools for building the application, how its performance will be evaluated, and its initial design considerations. Chapters four to seven will explain the application's design and implementation, then chapter eight will discuss the evaluation results. Finally, chapter nine will reflect on the accomplishments and outline future work.

2 Background

2.1 Literature Review

To make sure that the DUCK methodology is as effective and exhaustive as possible, existing approaches will be reviewed to see what principles can be borrowed. For elicitation, the concept of Goal-Oriented Requirements Engineering will be introduced, taking a close look at its derivative Keep All Objectives Satisfied. Problem Frames approach is concerned with the domain in question. Easy Approach to Requirements Syntax is a requirement formalisation method. Lastly, Use Case 2.0 is a project management approach introducing some new features to use case modelling.

As a running example for this paper, consider that a system is being designed to control a door and alarm in a self-storage facility – the Secure Self Storage System (SSSS).

2.1.1 Goal-Oriented Requirements Engineering

Goal-Oriented Requirements Engineering (GORE) is an approach that splits requirements analysis into two parts: acquisition and specification (Dardenne et al., 1993). The acquisition stage is concerned with building a preliminary “meta-model” from requirements and application domain knowledge written in natural language. The model is built by identifying abstractions like objectives, actions, relationships, and constraints and expressing them in an “acquisition language” which would eliminate some of the inadequacies and ambiguities that come with natural language but can still be understood by stakeholders involved in the elicitation process.

It is proposed to automate this process by building an acquisition assistant which would produce a database of requirements from a knowledge base of domain details and their relation to meta-model abstractions. Among the resulting requirements will be satisfaction of intermediate goals necessary for higher level objectives. Such goals may not necessarily be formulated by stakeholders from the start but must be considered by the development team when building the implementation. Goals

can be operational or non-operational, further differentiating into satisfaction, information, robustness, consistency, and safety goals.

Once enough information is gathered this way, a solution for the system to be designed can be found. It is further formalised through systematic refinement and structuring during the specification stage. A formal proof can then be carried out to check if the resulting model is valid and if so, the specification can be passed on to developers and implementation may begin.

2.1.2 Keep All Objectives Satisfied

Keep All Objectives Satisfied or Knowledge Acquisition in autOmated Specification (KAOS) is a goal-oriented methodology that works by building four models which in turn are represented by a series of diagrams. The models are overlapping and interconnected, together making up the overall requirements model which can then be used to produce a requirements document that would answer the questions about the system to be built (Respect-IT, 2007).

First, KAOS builds a goal model to answer the “Why and how?” question. It does that by taking high-level stakeholder objectives and systematically refining them to discover all the subgoals, eventually leading a set of low-level software requirements and external agent (actor) behaviour expectations.

Goals, requirements, and expectations may have conflicts and obstacles between them. A conflict is when one goal contradicts another, meaning they cannot be satisfied at the same time. For example, on high level there can be a conflict between the business objective of cheap implementation and safety requirements. On implementation level, a conflict might be the SSSS refusing to open the door due to an invalid PIN entry on the external keypad while having to open it for an exiting person who provided a valid PIN on the internal keypad. An obstacle is a possible condition that would obstruct the satisfaction of a goal. These are usually malfunctions and misuses found at lower level, but by chain reaction can make an impact on higher level objectives. Domain experts should then be consulted to decide what mitigations may need to be put in place.

Second, a responsibility model is built to answer the “Who?” question. It is linked to the goal model through requirements and expectations which can be thought of as systems and user requirements respectively. If necessary, new agents can be introduced, for example a member of staff could be assigned a responsibility within the system to satisfy an expectation.

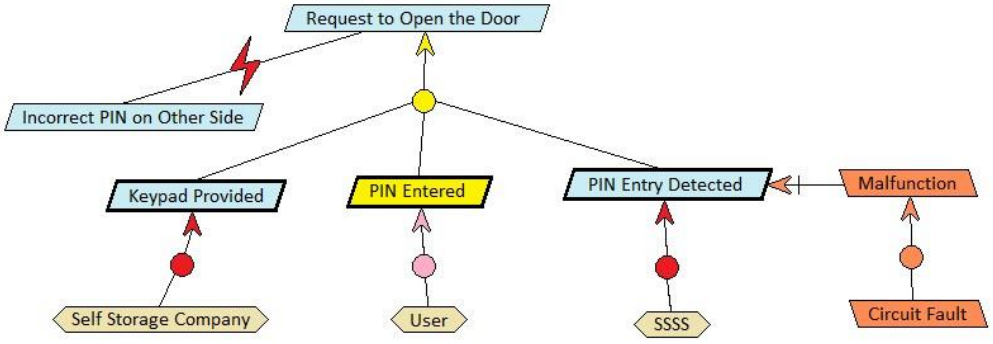


Figure 1. A KAOS diagram

Then, it is advised to build an object model – a consistent and complete glossary of terms within requirements that answers the “What?” question. This glossary would consist of active agents, passive entities, and the relationships between them. To do that, a formal notation of relationships is introduced. For example: ***enteredPIN(p, k)*** means that a person p entered a PIN on keypad k.

Lastly, to answer the “When and what to do?” question, an operation model is built. It is linked to the object model by defining pre and postconditions of actors’ actions in terms of relationships. For example, the “person” actor would have the “***OpenDoor***” action that would be defined as ***Closed(d)*** → ***Open(d)***, a transition of door d from closed to open state.

These four parts are put together to build a complete model of the system to be reviewed and validated. Overall, the scope of KAOS is very broad, it seeks to consider every detail and every possibility.

2.1.3 Problem Frames

Problem frames is a problem-oriented approach that focuses on interactions between a computer system and the “problem world”, aiming to eliminate the possible mismatch between the software’s

model and the actual, physical world (Jackson, 2009). It is thus essential to accurately represent the problem world in a machine-understandable way, so it is described in terms of domains and their phenomena – parts of the world and their states or properties. A domain could be an actor, human or system, or a physical object, including actuators and sensors. In the SSSS example, client and door, are domains, client's phenomena would be outdoors/indoors, and door's phenomena would be closed/open and locked/unlocked. In problem frames it is assumed that the problem world may not be changed.

A high-level objective is initially given to be refined into specific interaction requirements. Once identified, requirements will be modelled as domains interacting with the computer system. This can prove to be a formidable task for large systems, so it is necessary to decompose the complex program into a set of smaller, simpler subprograms.

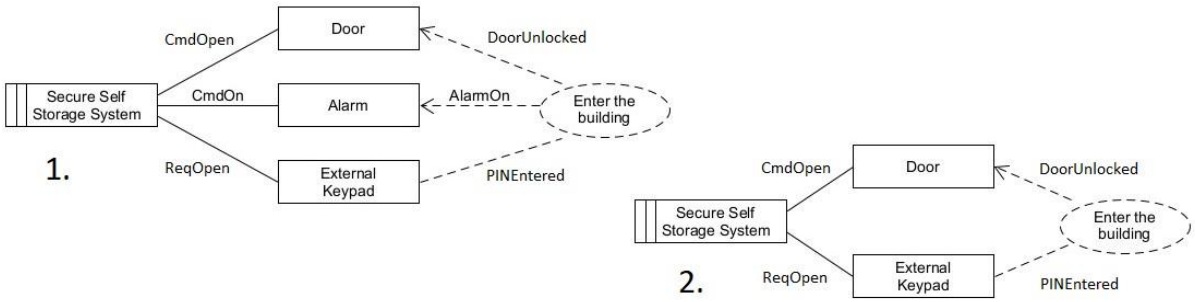


Figure 2. SSSS Problem Frames diagrams at two stages of decomposition

In these graphs the system is represented by the doubly striped rectangles, domains by the rectangles and requirements by dashed ellipses. Additionally, it is suggested that the phenomenonology can be formally defined and a notion of states and events is introduced:

DoorUnlocked = state: the door is unlocked ***LockDoor = event: the door is locked***

PINEntered(k) = event: PIN is entered on keypad k

These definitions can then be used to logically prove a list of steps for satisfaction of a goal. A proof failing to reach the expected result, is a sign that a domain may then need to be strengthened or weakened by adding or removing its influences of the phenomena (Ireland, 2021).

2.1.4 Easy Approach to Requirements Syntax

Easy Approach to Requirements Syntax (EARS) is a method to convert stakeholder requirements written in natural language to a formalised syntax for the sake of alleviating problems like ambiguity, complexity, and omission (Mavin et al., 2009). It suggests that most requirements will fall within one of five generic types: ubiquitous, event-driven, unwanted behaviour, state-driven, optional feature. C Complex (compound) requirements should be broken down can be broken down into multiple generic ones. For each type of requirement a template defined in terms of terms of actors, their responses, conditions, triggers, and if-then/while/where statements. For example, the template for an event-driven requirement is:

WHEN <optional preconditions> <trigger> the <system name> shall <system response>

So Functional Requirement 4 of the SSSS can be rewritten as:

"When the PIN of a person who is currently recorded as present is used in an attempt to enter the building, the SSSS shall trigger the alarm and send a message to the SMS"

The interpreted requirements show complete elimination of untestability, duplication, omission, and complexity. There is also a significant reduction in wordiness, vagueness, and ambiguity.

2.1.5 Use Case 2.0

Use Case 2.0 is an approach that is mostly concerned with project management, and scalability (Jacobson et al.,2011). It heavily emphasises that use cases can be used in an agile manner by splitting them into "slices" to be incrementally developed. This typically means that the main flow functionality from the narrative (scenario) would be implemented and whatever extension cases there would be left for later increments. The main flow itself can also be sliced further – certain steps

can be simplified or left out for the time of a sprint, for example if they are dependent on another use case's implementation being finished.

The Use case 2.0 approach consists of five "work products": the use case model, narrative, realisation, test cases and supporting information. These have several levels of completeness depending on the stage in the project's lifecycle.

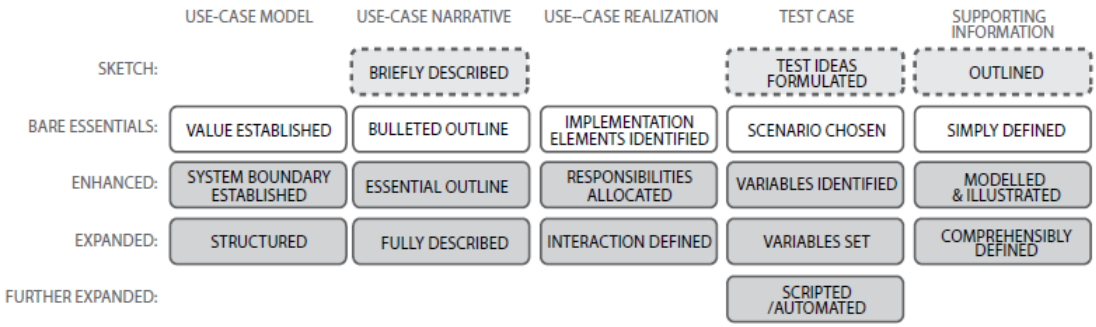


Figure 3. Products and their levels Use Case 2.0 (Jacobson et al.,2011)

2.1.6 Summary of Findings

The approaches discussed have many common concepts. KAOS, Problem Frames and use cases are goal-oriented. Refinement in KAOS, decomposition in Problem frames, compound requirements in EARS, and slicing in Use Case 2.0 all suggest breaking complex requirements into smaller parts. The methodologies also agree that formalism is needed to alleviate problems arising from writing requirements in natural language. Object model in KAOS, domain phenomenology in problem frames, and fully described narratives in Use Case 2.0 advise to build a detailed model of the problem world to understand the steps and constraints to achieve a goal. Obstacles in KAOS and unwanted behaviour template in EARS aim to identify potential issues that could arise during operation, the Problem Frames approach also addresses the possible difference between assumption made by the software model and the actual state of the problem world.

There are, however, a few minor differences in the approaches and their purposes. KAOS covers high-level requirements and business rules, while Problem Frames focuses more on the system's technical implementation and interaction with the problem world. Additionally, both KAOS and Problem

Frames prescribe to model the entire system rather than just the software, but KAOS allows exploring different possibilities for the domain while Problem Frames allow no changes to be made.

2.2 Existing Software

Many software tools exist to assist with requirements engineering and use case modelling. A handful will be reviewed before designing the DUCK assistant application to identify features that should be included, conventions that should be followed, and shortcomings that should be improved upon. Two popular tools are CaseComplete and Enterprise Architect. The two are similar, but Enterprise Architect's scope covers all of UML while CaseComplete is more lightweight. For both a 30-day free trial is available and will be used to investigate their capabilities. Objectiver is different, being the software implementation of KAOS. These three are proprietary software, but there are also some open-source tools like UMLet, although their functionality may be more limited.

2.2.1 CaseComplete

As the world play implies, CaseComplete is a tool with a primary focus on use cases, however it has many other capabilities like generating traceability matrices and report documents. For use cases it aids with drawing the diagrams, writing the descriptions, conditions, scenarios, and constraints. A 30-day free trial was used to investigate the functionality.

CaseComplete takes the goal-oriented approach as roadmap suggests starting the project by build a context diagram, identifying actors and their goals which would then be used to generate use cases. After this you will have a list of actor and use case entities with which you will build the use case diagram. Upon dragging and dropping elements onto the canvas, connections will be generated based on how the goals were defined. Opening the properties of a use case would bring up a form which needs to be filled in with the description, supporting actors, main success scenario and its extensions. In the details tab, preconditions and success guarantees can be defined. Together this aligns well with the use case scenario template given in F28SD.

CaseComplete has a few design decisions to be considered. First, there is no support for actor abstraction, so the “person” from SSSS would either be treated as another concrete actor will have to be linked to client and staff member manually. Alternatively, “person” can be omitted, meaning the shared enter and exit goals will have to be given for both client and staff member actors. However, when this approach is taken, CaseComplete acknowledges the duplication of the goal and creates a single use case, linking it to both concrete actors.

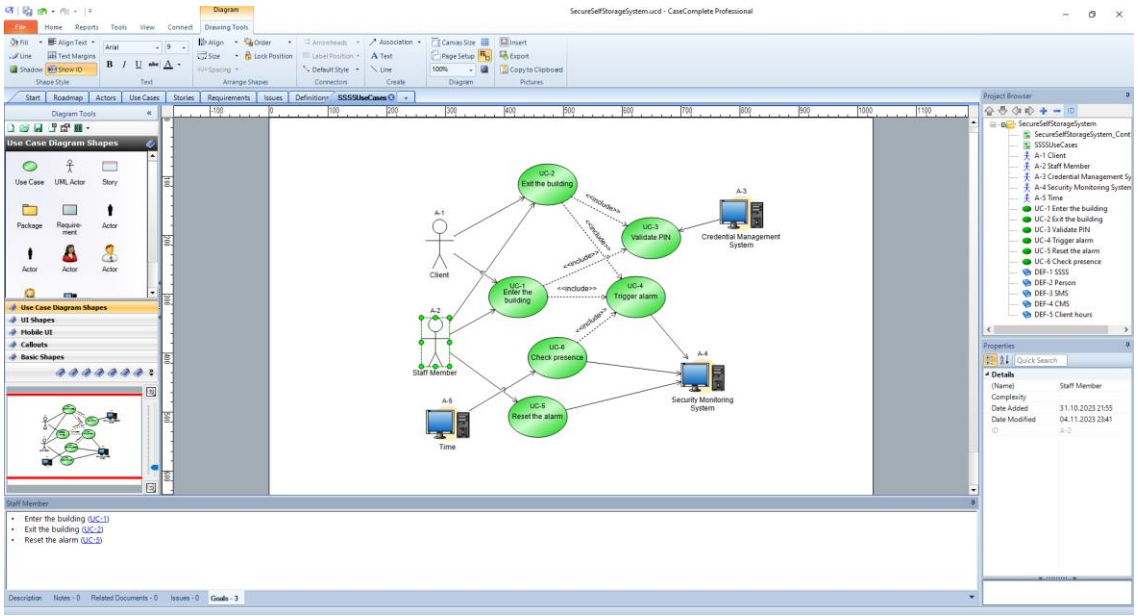


Figure 4. CaseComplete

CaseComplete has a dictionary where a term can be given a definition and a set of aliases. When writing anything, these terms or their aliases will be highlighted, hovering the mouse over them will bring up their respective information. The aliases feature would have been useful for actors, especially acronyms of systems like CMS or SMS, but actors can only go by one name. Attempting to resolve this by adding a definition with the same name as an actor or giving the actor’s name as an alias produces inconsistent results – the duplicated terms can appear in both forms, or all mentions of the actor can be overridden by the definition. This can cause mistakes in traceability as the

different IDs will be used in different places. Definitions can also have fields – pieces of data with a name, type, and definition. This is particularly convenient for complex inputs and data structures.

Apart from merging duplicated goals there is no consistency checking in CaseComplete – when generating the use cases, it assumes that any missing information will be completed later, so for actors without any goals it will simply not generate any use cases rather than question if they are redundant. Additionally, it is possible to have multiple actors with the same name – all entities go by their IDs, so CaseComplete does not see this as a problem.

2.2.2 Enterprise Architect

Enterprise Architect is an application with a very broad scope, from high-level business modelling to code generation from state machine and class diagrams. Though the array of supported standards is vast, UML is the prime focus, reaching a full completeness of the OMG specifications.

Use case models can thus be built with maximum precision. For example, when connecting elements of the diagram, the type of connection can be selected. For actors, abstraction and generalisation are supported, for use cases there are both the include and extend links. Use case “structured scenario” writing is detailed, allowing to give each step a result, associated state, and requirements, and differentiate between system and actor-performed steps. Steps can be linked to include and extend use cases and made a starting point for an alternative or exception path, which in turn can be specified to the same level of detail as the basic path (main flow). Interestingly, aliases can be given in actor properties, but are not highlighted and no traceable object link is made. Terms can be added to a glossary, but aliases are not supported for these.

While there is an option to validate a package, there are some limitations. The well-formedness check for use case diagrams only looks at the looks at the UML-compliance of relationship links between elements (e.g., element cannot be a generalisation of itself) and for unrealised requirements. Most relationship violations are prevented by restricting what type of connection the user can attach to an element of the graph. For example, if the user starts dragging an <include>

connection from a use case, it simply would not “stick” to an actor element, thus not registering as an illegal relationship.

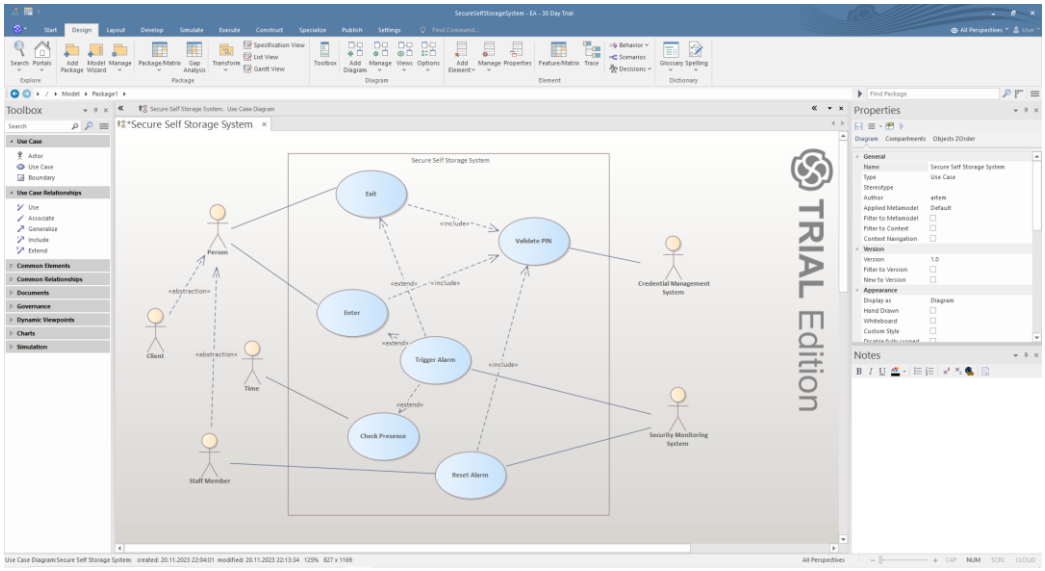


Figure 5. Enterprise Architect

Some other preventions are made, for example Enterprise architect does not permit copying and pasting diagram elements as to not duplicate an object. However later renaming elements to have the same name is possible, likely because an id system is being used. Glossary terms are stricter, neither creating nor renaming an entry to an existing name is permitted at all. There is no validation for use case scenarios, likely because the is no standard for them in OMG’s UML specification. However, a state machine diagram can be built instead to be executed and validated.

However, there are some third-party extensions for additional use case model validation Enterprise Architect. For example, UML Architecture Validation Extension (Cephas Consulting Group, [2]) would also check that there are no elements without a name or with the same name, or no connectivity. It also makes sure that all use cases have pre and postconditions and are linked to at least one requirement.

2.2.3 Objectiver

Objectiver is a software implementation of the acquisition assistant for KAOS. KAOS diagrams can be built using a palette of elements and relations. This would then be complemented by a domain concepts glossary, giving properties of agents and objects. When building the diagram, conformance to KAOS syntax is checked automatically, so for example certain elements can only be linked together using certain relations, otherwise an error will be raised. Completeness checks can point out goals with no refinement and requirements with no agent or operationalisation. Model quality checks ensure that each concept's definition is complete and that it appears at least one of the documents and diagrams. Though there is no built-in UML support, a note on the Respect-IT website suggests that Objectiver's goal-driven requirements elicitation approach can help identify use cases.

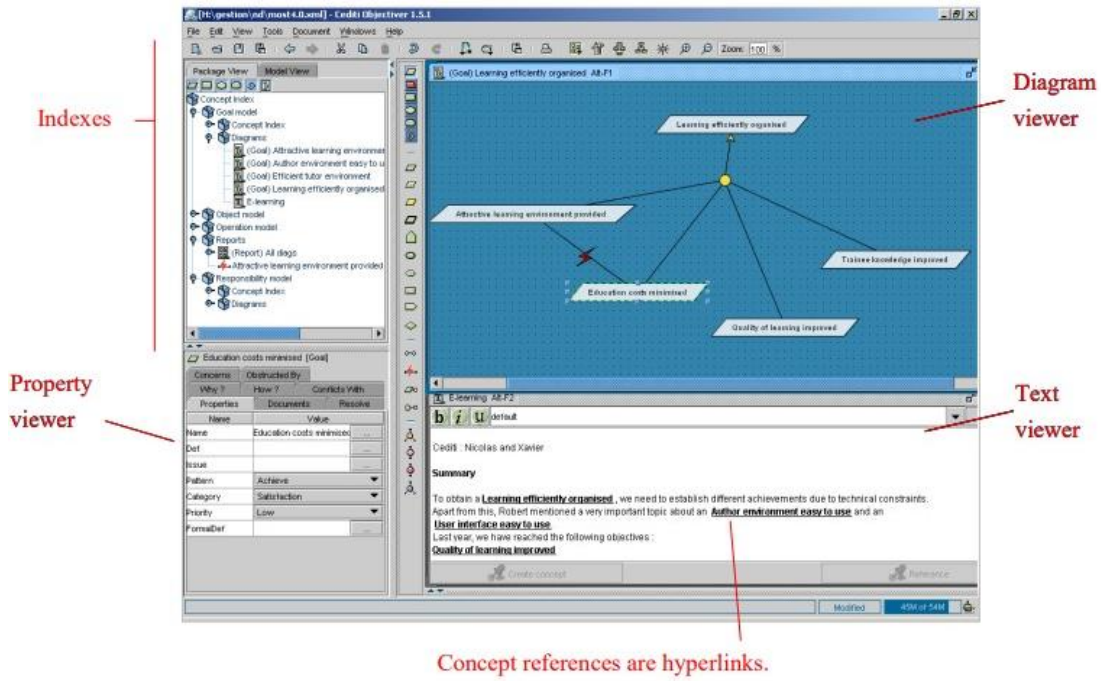


Figure 6. Objectiver (Respect-IT, 2008 [10])

2.2.4 UMLet

In contrast to CaseComplete and Objectiver, UMLet is a free, open-source program, however its functionality is limited to drawing UML diagrams and generating class diagrams from Java code files. UMLet comes as a standalone and web application, Eclipse, and Visual Studio Code plugins.

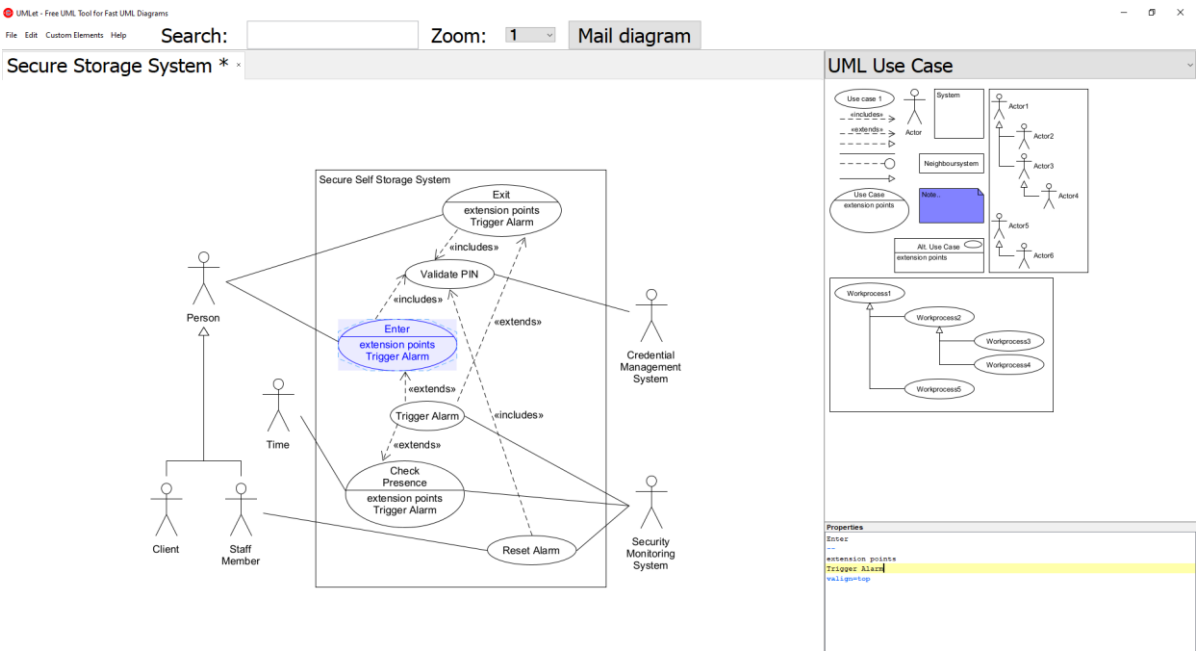


Figure 7. UMLet

2.2.5 Summary of Findings

	CaseComplete	**Enterprise Architect**	**Objectiver**	**UMLet**
Use cases diagrams	Yes	Yes	Can aid creation	Yes
Text writing	Yes	Yes	Yes	No
Traceability	Yes	Yes	Yes	No
Consistency checking	No	Third party	Yes	No
Open source	No	No	No	Yes

Table 1. Comparison of software tools

Though there are differences, certain conventions seem to exist for this type of application and so can be used as a guideline for the DUCK assistant application. For example, some cues can be taken from existing software when designing the GUI layout: CaseComplete and Enterprise Architect both have tabs under the toolbar, a palette of diagram elements on the on the left and a project browser on the right. While sometimes slightly differently named and designed, CaseComplete, Enterprise Architect, and Objectiver have a similar set of forms and tables for the user to fill out.

However, CaseComplete lacks many features like abstraction, extension connections and actor aliases that are present in Enterprise Architect which is conformant to the UML specification. Additionally,

each step in an Enterprise Architect use case scenario flow can have a result and an associated state, but CaseComplete does not have that. Despite these disadvantages, CaseComplete is more intuitive due to its simpler design, for example its use case forms closely resemble the templates given in F28SD. Use cases in Enterprise Architect have more detail, but it is scattered across multiple windows. While its capabilities are far greater, Enterprise Architect can be very overwhelming to a new user.

Model validation only goes as far as where an official standard exists – UML specification conformance for diagrams in Enterprise Architect or KAOS syntax conformance in Objectiver, consistency checking of use case scenarios is yet to be implemented by a software application. CaseComplete and UMLet have no validation at all and let the user draw diagrams as they please.

3 Requirements Analysis and Project Plan

As KAOS and Objectiver, the DUCK methodology and assistant application will also be “an approach supported by a tool”. Thus, a set of steps and rules will first be laid out allowing to build the knowledge base and use case models on paper. This set of rules will then give a set of requirements for the software intended to mechanise this process.

3.1 The DUCK Methodology

The user would start with requirements and problem world detail in plain text (natural language). The first step to building the DUCK knowledge base is to rewrite plain text requirements in terms of EARS templates. This formalisation will simplify elicitation of knowledge base entities: actors, actions, goals, objects, and states. The user will have to define them in accordance with the following rules:

- Actors have goals, actions, and states.
- Actions have preconditions and postconditions in terms of actors’ and objects’ states and may have an associated object.
- Goals may consist of actors’ states and objects’ states.
- Objects have states.
- States are Boolean true/false values.

The user would then build the DUCK use case model which consists of the diagram and scenarios. Actor and goal entities of the knowledge base would be used to give the actor and use case elements for the diagram. After the diagram is drawn, use case scenario forms will need to be filled out for each use case. This will be done using knowledge base entities so that:

- Actors and goals are actor and goal entities.
- Preconditions and postconditions must be defined in terms of actors’ and objects’ states.
- Each step of a flow must be an actor’s action entity.

Once this is finished, the user would check the consistency of the model.

The consistency rules are:

- The definition of all entities in the knowledge base is complete and none are duplicated.
- There are no use case diagram elements that have no valid links, and none are duplicated.
- All actor and use case elements on the diagram can be linked to an actor and use case entity in the knowledge base and vice versa.
- All actor, use case elements and their connections on the diagram correspond with the actors and goal entities in use case scenario forms and vice versa.
- All flows in use case scenarios have a valid "proof" in terms of knowledge base entities – starting from the preconditions, following the action steps must result in satisfaction of the postconditions.

If any of these rules are not satisfied, corrections must be made to knowledge base entities definition, use case diagram or scenarios.

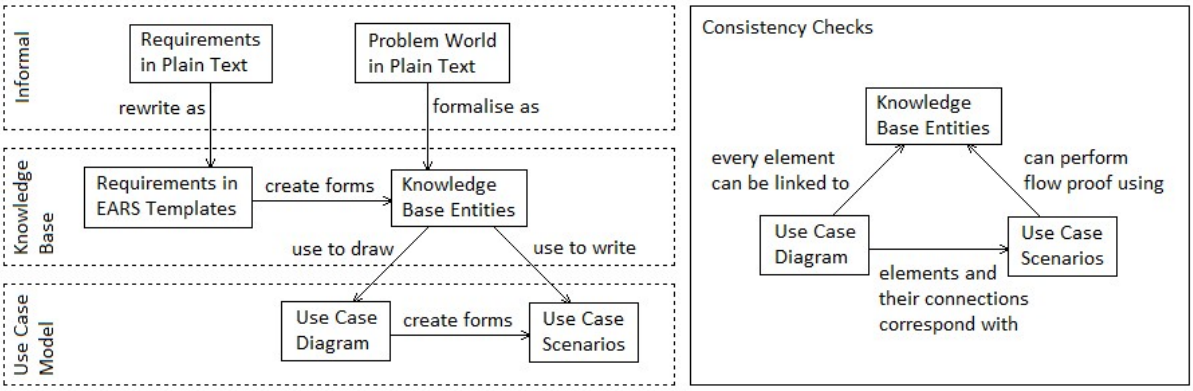


Figure 8. DUCK methodology model building (left) and validation (right)

While a specific order is given here, in practice the user may jump between the different stages. For example, it is not necessary to fully complete the knowledge base before drawing the use case diagram, the user may want to only model a slice at first. They may also reverse this order by drawing the diagram first and eliciting knowledge base entities from elements, the same applies for the scenarios.

3.2 The DUCK Assistant Application

The program must have means for both text and graphic and text input to build the knowledge base and use case model. This will be done in windows which the user would access by selecting a tab. For knowledge base input, a series of tables and forms will be provided, for the use case diagram – a drawing area and a palette of UML elements. Use case scenario template forms could be generated from the diagram to be filled out.

After the model is built, a button would be pressed to check the consistency. A popup window will appear listing any errors that were found. The user must also be able to save the DUCK model and load it later, so a new file format will be designed to record this information.

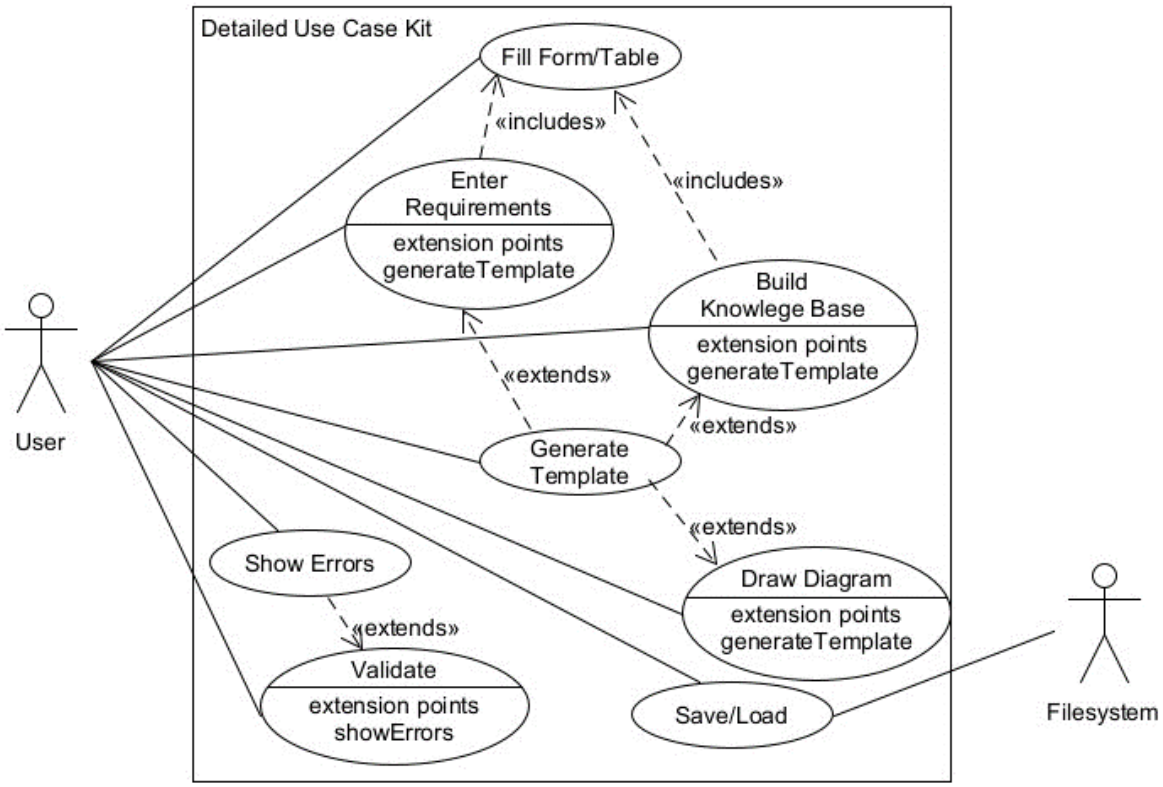


Figure 9. A use case diagram for the DUCK assistant application

3.2.1 Functional Requirements

ID	Description	Priority
FR1	The user shall be able to build the knowledge base.	M
FR1.1	The user shall be able to define a list of requirements using EARS templates.	M
FR1.2	The user shall be able to define a list of actors, their goals, actions, and states.	M
FR1.2.1	The user shall be able to define actor actions and goals in terms of objects, and states.	M
FR1.2.2	The program should support abstractions of actors.	S
FR1.2.3	The program should identify reusable actor goals.	S
FR1.3	The user shall be able to define non-actor objects, their states.	M
FR1.4	The user could be able to generate a checklist for drawing the use case diagram from actor and goal definitions.	C
FR1.5	The user would be able to generate a use case diagram from definitions.	W
FR2	The user shall be able to build a UML use case model.	
FR2.1	The user shall be able to draw a use case diagram.	M
FR2.2	The user shall be able to generate use case scenario template forms from the diagram.	M
FR2.2.1	The user shall be able to define preconditions and postconditions in terms of actors, objects, and states.	M
FR2.2.2	The user shall be able to define main and alternative flows in terms of actor actions.	M
FR3.1	The user shall be able to check the consistency between the use case diagram and scenario forms.	M
FR3.1.1	The program shall raise a consistency error when: i. The diagram has actors/use cases with duplicated names OR ii. The diagram has actors/use cases that are not connected to any use cases/actors OR iii. An actor/extension case/inclusion case is mentioned in a use case's scenario form but the actor/extension case/inclusion and use case are not connected on the diagram.	M
FR3.1.2	The program should raise an error when the graph does not conform to OMG's UML specifications.	S
FR3.2	The user shall be able to check the consistency within the scenario flows.	M
FR3.2.1	The program shall raise an error when: i. The scenario flow contains entities whose definition is incomplete OR ii. The flow proof fails.	M
FR3.3	The program should suggest corrections to fix inconsistencies.	S
FR4	The user could be able generate traceability matrices for the model.	C
FR5	The user shall be able to save, open and edit DUCK model files.	M

Table 2. Functional Requirements

3.2.2 Non-Functional Requirements

ID	Description	Priority
NFR1	The program shall be available as a standalone application.	M
NFR2	The program shall use a new file format to represent the model.	M
NFR3	The program shall use forms and tables and boxes for text input.	M
NFR6	The program could track and update changes and update the model in real time.	C
NFR7	The user could be able to use shortcuts to accomplish the tasks.	C
NFR8	The program could provide tips.	C

Table 3. Non-Functional Requirements

3.3 Tools

3.3.1 UMLet and Java

It was decided to use UMLet as the foundation for the DUCK assistant application. While UMLet's limited functionality can be seen a disadvantage, it is a blank slate which will allow greater freedom in designing the DUCK. Additionally, UMLet is the recommended software tool for F28SD, so it makes sense to base the assistant application on it as it provides an opportunity to recruit students who are familiar with it to participate in the usability study. This would accelerate the user testing process, allowing to

UMLet source code is available on GitHub and a comprehensive developer guide given with instructions on how to import it into Eclipse IDE and build the program. It is also very convenient that UMLet is written in Java and Eclipse is recommended as this is the most used language and IDE through the first three years of the Computer Science programme at Heriot-Watt.

3.3.2 UMLet Source Code in Eclipse IDE

To see if using Eclipse to extend UMLet code would be feasible, the developer guide was followed. The code was imported, prepared for development and a "Hello World"-type test modifications were made. The modified code compiled successfully, launching UMLet with a couple of changes to the GUI and console logs of updates to the diagram. Thus, this approach was proven to work without technical issues, development of the application can be started.

3.4 Initial Evaluation Strategy

3.4.1 Unit Testing

Unit tests will be written for functionality being designed to clearly outline the expected behaviour of the program. During development these tests will have to be passed before starting on work on any dependant functionality. This will serve as quality assurance for the application and thus can act as a metric for the success of the implementation.

3.4.2 Case Study Testing

Once individual parts of the program are complete, the application's functionality, from start to finish as prescribed in requirements analysis, will be assessed using case studies. The case studies will be examples taken from the "Writing Effective Use Cases" book (Cockburn, 2001) with a set of requirements, flows and a diagram to be used as input. These inputs will make up the test cases: either entered correctly or a mistake could be intentionally made to determine if the program will catch the inconsistencies.

3.4.3 User Testing

Once the application development is finished, user testing will be carried out to evaluate the GUI. The participants in this usability study will be second year students at Heriot-Watt University studying Computer Science or Computer Systems who have covered the requirements gathering and use case topics in F28SD and had some experience with UMLet. Like case study testing, the participants will be given a set of inputs to enter into the application and directions on how to use the functionality. As only the application GUI will be evaluated rather than the participants' performance, these tasks will be designed as a streamlined experience. This means that participants will be given precise enough directions so that they are able to interact with the application in full, not missing any functionality because they got stuck at any point.

After completing the tasks, the participants will be asked to complete a System Usability Scale questionnaire on MS Forms. These results can then be used to measure the quality of the GUI and tell if it was designed well for its purpose.

3.5 Design and Implementation Approach

The assistant application's design will be shaped by the DUCK methodology as outlined in section 3.1 as it provides a clear set of concepts, rules, and functions. Methodologies given in the literature review section may also be consulted, the functionality and interfaces of existing software tools could be used as reference. The DUCK methodology may be revised over the course of the project.

First, UMLet source code would have to be investigated to find what parts of it can be used and how the DUCK functionality can be integrated. The general philosophy behind this reverse-engineering process is that some code used to render the diagram correctly can be tapped into to build a correct logical model of the meaning behind that diagram (FR1-4). Additionally, any new GUI elements of the assistant functionality (NFR3) will have to be integrated with existing UMLet GUI elements. Overall, building the assistant application would involve both modifying some existing UMLet files and creating new files for which an appropriate place will have to be found.

Creation of Knowledge Base entity classes (FR1) can be done concurrently with UMlet reverse-engineering as they will be self-contained. Building the validation function (FR3), however, would require knowledge of UMLet's diagram backend. The GUI (NFR3) will be developed alongside the backend logic. The development of the assistant application will be incremental, functionality which is deemed to have the most value will be given prioritised. Over the course of the project, priorities given to requirements in sections 3.2.1 and 3.2.2 may be reconsidered and new requirements may be discovered.

4 Design and Implementation: UMLet Reverse-Engineering

4.1 Repository Contents and Structure

The code in UMLet GitHub repo is grouped into modules by purpose and platform; the "Information for Developers" wiki (Andreas, 2023) specifies which modules are required for each platform version. As per Non-Function Requirement 1, the standalone version of UMLet will be used as base for the assistant application. The required modules for that platform are umlet-elements, umlet-swing and umlet-standalone. Below is a summary of the modules' packages and their functionality relevant to the development of the Detailed Use Case Kit.

4.1.1 UMLet Elements Module

The umlet-elements module is the Java implementation of UMLet's diagram elements used by all platform versions of UMLet. Starting from the basics, its com.baselet.control and com.baselet.diagram series of packages contain the basic geometry rules such as dimensions and coordinates, shapes such as lines and rectangles, and the functions to draw these shapes. Overall, these are the core concepts and structures that the diagram elements are built from and rendered. Additionally, the com.baselet.control.constants package contains a file with the menu constants (e.g. "Save as…", "Edit Current Palette"). The reason for putting them in umlet-elements was likely to ensure the consistency in menus across platform versions that use different modules for their GUIs.

The com.baselet.element package series then contains the element-specific properties and behaviours. For example, it draws on control and diagram classes to define UML elements in terms of basic shapes, instantiate them as Java objects, and render them on the diagram. It gives the algorithms for interactions with and between elements like recalculating how an element should be rendered to fit new dimensions after it is resized or making relations stick to the frames of other diagram elements.

4.1.2 UMLet Swing Module

The umlet-swing module contains UMLet's Swing (Swing is a Java GUI widget toolkit) interface implementation which is shared by standalone UMLet and the Eclipse IDE plugin version. The com.baselet.gui packages are responsible for handling user's interactions with the diagram like adding, selecting, moving, editing, copying, deleting elements and some other interactions like changing the program's settings. The package series then also contains classes which give the actual Graphical User Interfaces, for example OptionPanel.java and MailPanel.java. They are built by using various Swing components, arranging their layout, and defining interactions.
The modules com.baselet.diagram package series contain classes that import the umlet-elements module packages of the same namespace to act as a bridge between them and the aforementioned umlet-swing's com.baselet.gui classes while handling diagram input/output. It also contains the classes that enable the saving and loading of UMLet's .uxf diagram files.

Lastly, the com.baselet.comtrol package (again, not to be confused with the umlet-elements package of the same namespace) contains the Main.java class which is responsible for initialising the GUI and diagram handler when the application is launched and is then used to return their instances whenever another class needs them during runtime.

4.1.3 UMLet Standalone Module

As the name implies, the umlet-standalone module contains the standalone UMLet endpoint. This is where the various umlet-swing GUI elements, the menu constants from umlet-elements and some native components and listeners of the module are arranged into the layout of UMLet's standalone version. For example, umlet-standalone's MenuBuilder.java imports MenuFactorySwing.java from umlet-swing and MenuConstants.java from umlet-elements to build the dropdown menu toolbar and context menus which appear when right-clicking in the diagram tab or on a diagram element. Similarly, MainStandalone.java calls on Main.java from umlet-swing to launch the application.

4.2 Diagram and Graphical User Interface

4.2.1 Grid Elements, Current Diagram and Current GUI Instances

Elements on a diagram are objects of GridElement class defined in umlet-elements. GridElements are stored by an instance of CurrentDiagram class defined in umlet-swing. In the same module, the CurrentGui class holds the current instance of the entire GUI, including diagram. The GUI is initialised by the Main function when the application is launched.

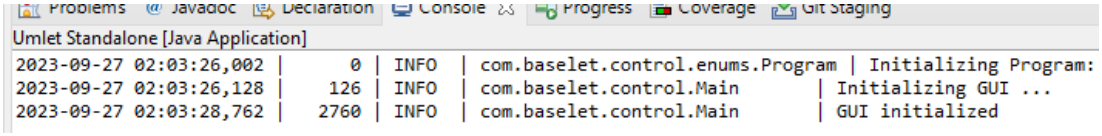


Figure 10. Console messages about GUI initialisation in Main when launching UMLet

At runtime, GridElement objects are stored by an instance of CurrentDiagram type object which in turn is held by an instance of CurrentGui type object. The function combination

CurrentGui.getInstance().getGui().getCurrentDiagram().getGridElements()

would return a collection of GridElement type objects.

4.2.2 JPanels and JSplitPanes

When looking for existing GUI elements to base the DUCK's interface off, the option and mail panels were found to be best picks and were thus reverse engineered to be used as reference. Both OptionPanel.java and MailPanel.java extend the JPanel class (the basic container) and implement the ActionListener class to detect and react to user input. An important difference between the two, however, is that the option panel pops up as an independent window while the mail panel is integrated with the rest of the GUI so it will share the main window when it slides out.

Swing's JSpitPanes are used to divide an area of the GUI between two components vertically or horizontally. That boundary can be dragged to resize the components. These JSplitPane objects can also be nested, so in BaseGUIBuilder.java rightSplit vertically splits the palette and Properties panel,

then mainSplit horizontally splits righSplit and the diagram while mailSplit vertically splits mainSplit and the mail panel.

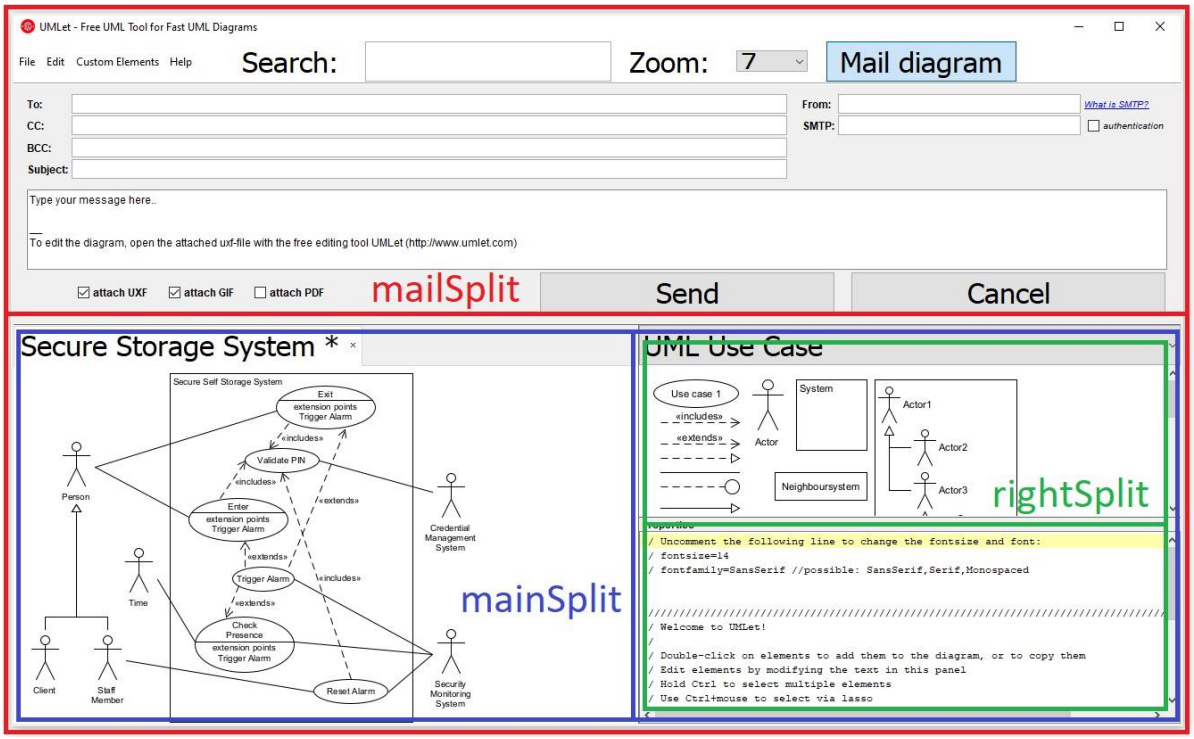


Figure 11. How UMLet GUI is split

4.3 Diagram Element Backend

4.3.1 Diagram Element Classes, IDs, and Panel Attributes

For a GridElement object, functions like getPanelAttributes() and getId() or Java's native functions like getClass() and toString() can be used to retrieve information about a diagram element:

```
Umlet Standalone [Java Application]
Output of toString(): com.baselet.element.elementnew.uml.UseCase@29b159b
Output of getClass(): class com.baselet.element.elementnew.uml.UseCase
Output of getId(): UMLUseCase
Output of getPanelAttributes(): Validate PIN

Output of toString(): com.baselet.element.elementnew.uml.Actor@1ffefbe
Output of getClass(): class com.baselet.element.elementnew.uml.Actor
Output of getId(): UMLActor
Output of getPanelAttributes(): Person
```

Figure 12. An experiment printing out results returned by different functions used on a GridElement

Class

In umlet-elements, GridElement is a parent class – all element classes inherit its common attributes and functions while having their own type-specific attributes and functions that are not shared.

The actual Java object class of the diagram element (elementnew.uml.UseCase, elementnew.uml.Actor, elementnew.uml.Generic, relation.Relation) is retrieved with the getClass() function and can be used to derive the "type" of entity it represents (use case, actor, system, relation).

ID

It was discovered that there are two types of diagram element "IDs" in UMLet code. One is a random combination of digits and letters after a "@" character (e.g. @29b159b, @1ffefbe seen on the above figure as the toString() result). It is aways assigned at runtime and is only used to tell which element is which when the diagram is being edited, but not saved in the .uxf files and will thus be different each time the same diagram is loaded.

The second type of ID is given in the umlet-elements module by the ElementId.java class as a constant (e.g. UMLUseCase, UMLActor seen on the above figure as the getId() result). A comment in ElementId.java warns that "these IDs should NEVER be changed because they are stored in uxf files". The warning hints that this kind of ID is used to record an individual element's type (class) rather than to distinguish between individual elements which is not necessary to correctly represent the diagram.

Panel Attributes

This is the text string given in the "Properties" panel retrieved with the getPanelAttributes() function. Panel attributes are used to set the label or sometimes the style of an element through UMLet's facet syntax. They can thus be used to find out the "name" of a diagram element or the type of a relation.

Figure 13. Using the property panel specify an element's name (left) and relation type (right)

4.3.2 Diagram Element Rectangle and Location

All diagram elements have a rectangular boundary inside which they are displayed. This rectangle is used to select the element by clicking on it, moving the element by dragging it around and resize the element by pulling on its boundaries. A rectangle's properties thus include its distance from the top and left edges of the diagram (the X and Y coordinates of the rectangle's top-left corner) and its width and height. The location of a diagram element is therefore represented that of its rectangle. Placement of elements is not entirely free – elements snap to a grid, hence the class name GridElement.

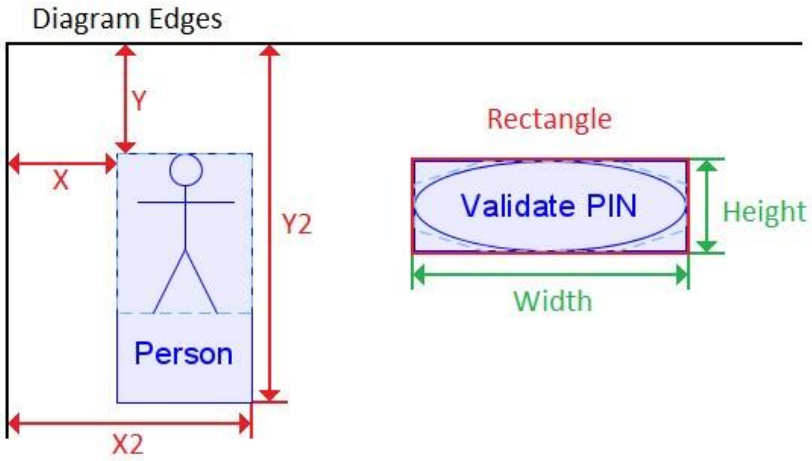


Figure 14. Diagram element coordinate values and rectangle

4.3.3 Relations and Sticking Polygons

Relations are an integral part of a UML diagram. While elements have an associated object, relation code is not concerned with them – it only needs to draw the diagram, so the code directly handles the coordinate points. A relation typically two "ends" or relation points with indexes [0] and [1].

However, a relation does not have to be a straight line, it can be shaped into a free path. The ends of intermediate lines would produce additional relation points which will be indexed accordingly.

The Relation.java class extends GridElement and thus relations have rectangles like any other diagram element. However, they are not rendered and are thus not visible to the user. The invisible rectangle is wrapped around the relation's outermost points to fit the whole shape within its boundaries. Coordinates of relation points are given relative to the edges of this rectangle rather than the diagram; however, a function exists to get their absolute position.

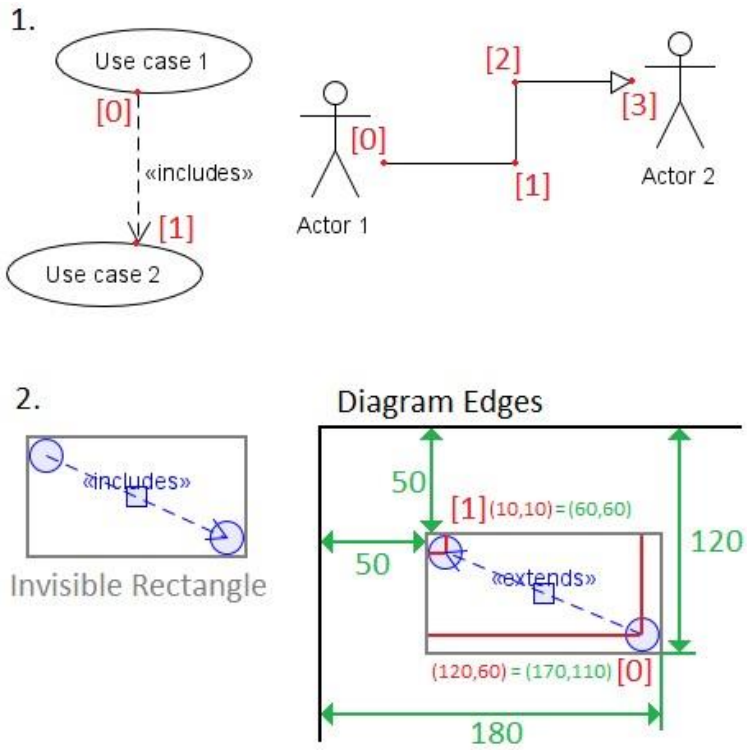


Figure 15. 1) Relation point indexes 2) Relation "invisible rectangle" and coordinates

The relation.facet package contains classes that handle a relation's appearance: arrow end, line type and description. These attributes are linked to the relation's panel attributes string. The line type is given with an **lt=** (meaning **l**-ine, **t**-ype) prefix followed by a dash for a solid line or a dot for a dashed line. The **lt=** prefix is also used to arrow end is given by **<** (less-than) and **>** (greater-than) characters for left and right end respectively. Depending on the number of these characters placed

consecutively, the appearance of the arrow is changed. Finally, any text given on a line different to the prefixes will be used as the arrow's description.

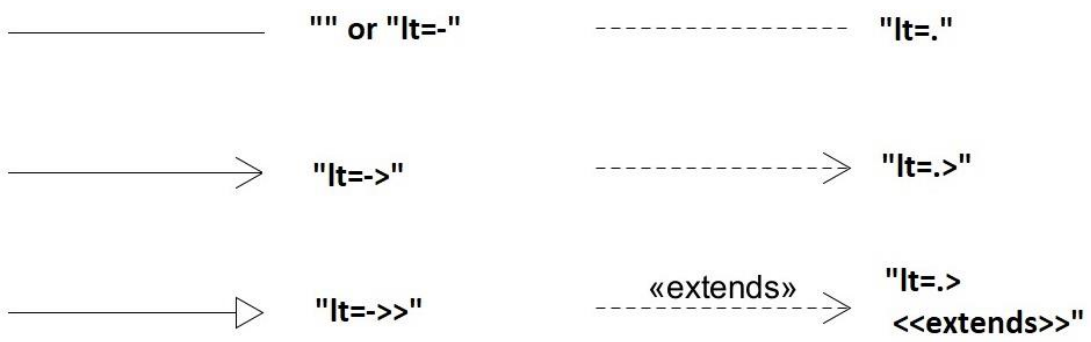


Figure 16. Relation appearance from panel attributes string

By using Java's contains() method, the line type and therefore relation type can be determined by checking for the respective substring, for example "extends" for an extends relation or "->>" for an abstraction.

UMLet relations "stick" to elements when they are moved around the diagram, which makes editing more convenient. To facilitate this, umlet-elements defines classes such as Stickable.java, StickLine.java, and StickingPolygon.java. The first class gives shared behaviour of things that stick, for example it is used to make relation point stickable. Since a grid is enforced, a given area only has a limited set of usable points, so a stickline is essentially a chain of stickable points. Finally, a set of sticklines can form a sticking polygon.

Unlike relations which only have two sticking points, other diagram elements have sticking polygons. A sticking polygon is contained within the element's rectangular bounds. The polygon's sticklines may often align with the rectangle's edges, but its overall shape can sometimes be different. For example, an actor has a rectangular sticking polygon, but a use case has an octagonal one.

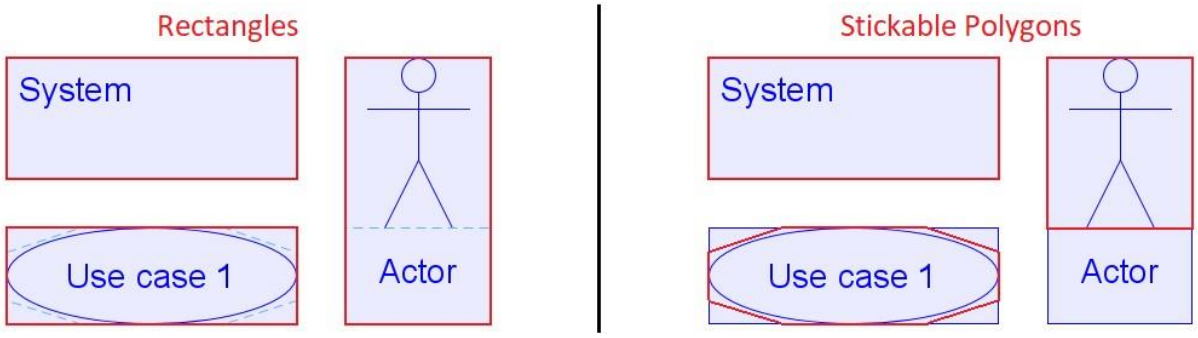


Figure 17. Rectangles and stickable polygons

In order to stick to an element, a relation's end must be placed exactly onto a stickline of the element's sticking polygon.

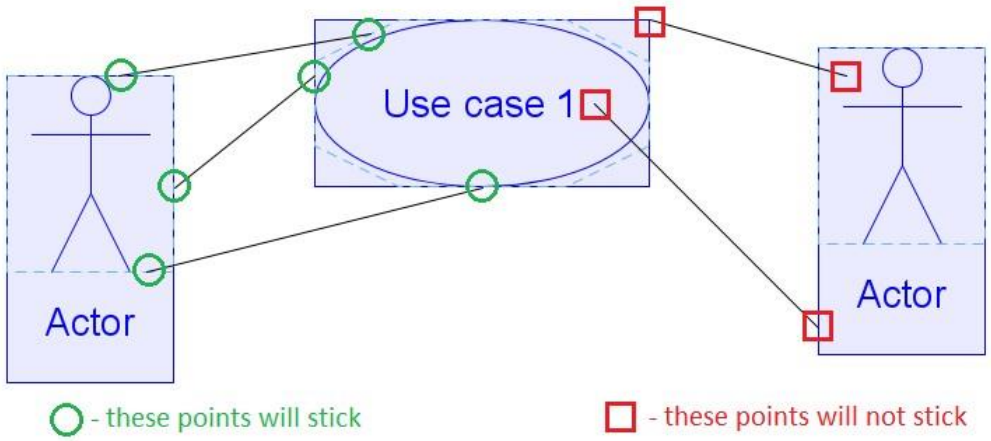


Figure 18. Where relations will and will not stick

5 Design and Implementation: The Knowledge Base

5.1 Assistant Application Package Structure

A new package was created in the umlet-swing module to hold the assistant application's files – com.baselet.assistant. As explained in section 4.1.2, this module was chosen to hold the new code because it allowed to easily integrate both the assistant backend (through umlet-swing's Main class in com.baselet.control) and the new frontend elements (which would be placed in the module's com.baselet.gui package) with existing UMLet functionality. The backend classes mostly correspond to entities and concepts outlined by the DUCK methodology in section 3.1. In total, the assistant package contains 11 new class files:

1. Action.java
2. Actor.java
3. DuckHandler.java
4. FlowStep.java
5. KnowledgeBase.java
6. LogStep.java
7. RelationTriple.java
8. Scenario.java
9. ScenarioLog.java
10. StateTriple.java
11. SystemBoundary.java

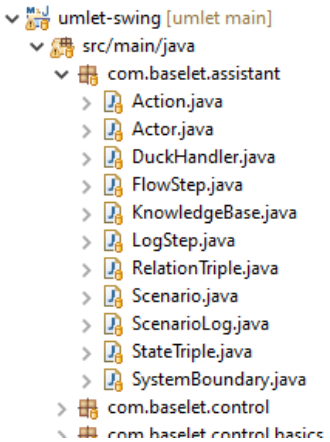


Figure 19. New assistant package as seen in Eclipse IDE

5.2 Knowledge Base Class

The Knowledge Base class stores the domain model entities as objects. As CurrentDiagram and CurrentGui, the Knowledge Base would in turn need to be initialised by some other class. It was decided to initialise it alongside the GUI in the Main function. Assistant application's classes could then use the function combination

Main.getInstance().getKnowledgeBase()

to access the Knowledge Base object instance.

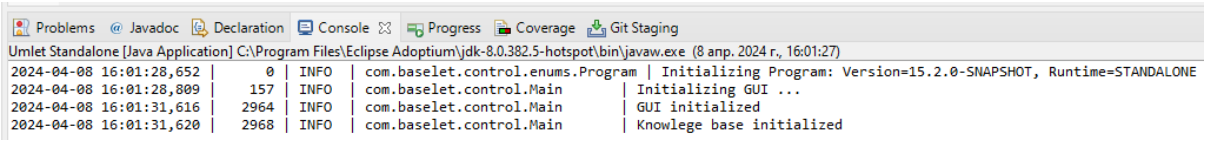


Figure 20. UMLet Knowledge Base initialiation seen in console logs

The knowledge base class holds the following as its attributes:

1. A single System Boundary object
2. A map of actors
3. A map of actions
4. A map of scenarios
5. A map of scenario logs
6. A list of domain objects
7. A list of states
8. An instance of the “DUCK Handler”

In line with the DUCK methodology, most of these are Knowledge Base entities. The choice of data structure for each will be explained in the respective sections. The rest are not part of the Knowledge Base on the methodological level but were placed here as on the technical level it was convenient to reuse the Knowledge Base data structure to store other objects as well. In the future, some shared “Assistant” object may be introduced to organise the objects more appropriately.

5.3 Knowledge Base Entity Classes

5.3.1 Objects and States

Objects and states are the most basic entities to exist in the Knowledge Base. Unlike other entities, the only property of an object or a state is its name, so both are stored as text strings:

Objects: “Door”, “PIN”, “Alarm” | States: “Locked”, “Closed”, “Entered”, “Validated”, “Triggered”

In terms of the Problem Frames approach looked at in section 2.1.3, objects and states would be considered domains and phenomena respectively. However, since Problem Frames' domains include actors, it may be more precise to describe objects as passive entities from KAOS (section 2.1.2).

In the original DUCK methodology, an object was supposed to have a set of associated states as its property, but for the sake of simplicity, it was chosen to forego explicit assignment of permitted states.

Lists are used to store objects and states as neither have an object behind their names, thus there is no value for a key-value pair that would warrant the use of a map. If eventually explicit associated state definitions were to be enforced for domain objects, a map would have to be used.

5.3.2 Actor and System Boundary Classes

The actor class is a custom data structure containing an actor's name the list of their actions. As with domain objects, it was decided not to enforce explicit associated state assignment for actors. The reason why action assignment is required is because during scenario flow proofs actions change the state of the domain, thus mistakenly having the wrong actor perform an action may impair the model's correctness. This would have a much more significant impact than assigning an irrelevant state to some entity. Since actors have a list of actions which warranted the creation of the new object type, a map is used to store them as there are key-value pairs of actors' names and objects.

The initial version of the DUCK methodology did not consider the system boundary as a separate entity type, it was assumed that it can be modelled as an actor entity. However, a problem with that approach was discovered later in the assistant application's development when the validation function was being implemented. The issue was that if the system was given as an actor entity, the validation function would expect to see a corresponding actor diagram element. The system boundary class was thus created so that an element of type Generic would be used instead. The two classes are functionally similar – both have a list of actions that they can

perform; however, the system cannot have associated states. In hindsight, this may not have been a good design decision as (e.g. if the system is operational or the mode of its operation). Additionally, though a use case diagram may have multiple system boundaries to represent interactions between different systems (Cockburn, 2001), this is beyond the F28SD course's scope, and so it was decided to restrict the amount of system boundaries to one. Because of this only a single instance of the system boundary object is stored in the knowledge base.

5.3.3 State Triple Class

In accordance with the DUCK methodology, the conditions of the domain are formalised as composite entity-state-value triples known as "state triples". The entity can be either an actor or an object, the state is some "phenomena", and the value is either True or False.

Entity	State	Value
Person	Inside	False
Door	Locked	True

Table 4. State triples

For example, "the door is locked" domain condition can be formalised as **[Door, Locked, True]** and the state triple for "the person is not inside" would then be **[Person, Inside, False]**. While methodologies like KAOS and Problem Frames suggest using entity-state pairs like ***locked(door)*** or ***outside(person)***, it was decided to include the Boolean value as it may reduce the number of "states" or "phenomena" that must be given. For example, instead of having ***locked*** and ***unlocked*** or ***inside*** and ***outside***, only one is required and the other can be represented using a state triple combination with the opposite Boolean value.

5.3.4 Action Class

The action class attributes are the action's name and two state triple lists: one for the preconditions and one for the postconditions. One thing that was dropped between the original methodology and the implementation is the associated object. This was both to simplify the validation and because it was seen as somewhat redundant. To elaborate, while KAOS may

include an object in an action notation e.g. ***enterPIN(person, keypad)***, DUCK could represent the fact that a keypad is required to perform an action as a precondition. If a person must be able to use it, it can be said that two preconditions for the action are **[Person, Inside, False]** and **[Keypad, Inside, False]**. This would mean that both the person and the keypad are outside, thus the person should have access to the keypad.

5.4 Scenario Class

Though not a Knowledge Base entity, the Knowledge Base contains the use case model's scenarios as at least in the initial stages of development creating a separate "ScenarioBase" structure was seen as adding unnecessary complexity. Since a scenario is a composite data structures with a unique key given by its name, a map was used to store them in the Knowledge Base.

5.4.1 Scenario Attributes

Apart from its name, a scenario stores a single primary actor and a list of its secondary actors as. The actors are referenced by their name text strings. As for an action, a scenarios preconditions and postconditions are stored as state triples lists. Finally, a scenario stores its main flow as list of flow step objects (explained below).

Attribute	**Value**		
Name	Enter		
Primary actor	Person		
Secondary actors	SSSS, CMS		
Preconditions	**Entity**	**State**	**Value**
	Person	Inside	False
	Door	Locked	True
Postconditions	**Entity**	**State**	**Value**
	Person	Inside	True
Main flow	**Actor**	**Action**	
	Person	EnterPIN	
	SSSS	RequestCredentials	
	CMS	ReceiveRequest	
	CMS	ReturnResponse	
	SSSS	ProcessResponse	
	SSSS	OpenDoor	
	Person	WalkInside	

Table 5. A scenario as stored in the knowledge base

5.4.2 Flow Step Class

Like state triples, the FlowStep.java is another auxiliary class of "nameless" objects which is used by the scenario class to formalise its main flow. Flow steps are also known as actor-action tuples because they are primarily used to formalise the sequence in which actors perform actions, for example "the person enters the PIN" would be represented as **[Person, EnterPIN]**. However, they were also used to implement the sub-scenarios "included" in the main flow, in which case the actor value would be "Include" and the action value would be the name of the name of the scenario being included e.g. **[Include, Validate PIN]**.

Actor	Action
Person	EnterPIN
Include	Validate PIN
SSSS	UnlockDoor
Person	WalkInside

Table 6. Actor-action tuples

5.5 Scenario Log and Log Step Classes

5.5.1 Scenario Log Class

A functionality that was not in the original set of requirements but was later seen as necessary was ability to actually see the flow proof and not just the resulting warnings or lack thereof. The "logs" of a flow proof would then need to be recorded and stored, for which again, the Knowledge Base was used. After flow proofs are performed as part of model validation, a scenario log would be produced for each scenario. A scenario log data structure then consists of its respective scenario's name and a list of log steps.

5.5.2 Log Step Class

A log step consists of the following:

1. The flow step's index
2. The flow step's actor name
3. The flow step's action name
4. The current state as a list of state tripes
5. The flow step action's preconditions as a list of state tripes
6. The flow step action's postconditions as a list of state tripes

A log step is conceptually similar to a flow step, which it effectively contains as attributes 2 and 3, with additional detail being the action's preconditions and postconditions as attributes 5 and 6. On top of that, it contains a record of the domain conditions before the aforementioned action is performed.

5.6 DUCK Handler Class

Though still stored in the Knowledge Base despite not being a KB entity, the DuckHandler class was created to have at least some degree of compartmentalisation. Named in convention with existing UMLet handler classes (e.g. DiagramHandler, RelationPointHandler etc.) the DuckHandler contains the validation function which will be covered in the chapter 6. The validation function is also referred to as the "Check Consistency" function (called that in the code and on the button used to trigger it) and can be accessed by through the

Main.getInstance().getKnowledgeBase().getDuckHandler().checkConsistency()

command combination.

The class also contains the function that handles a new type of user interaction with the diagram elements. Based on the selected element's class, the function chooses the appropriate window to display and passes the element's panel attributes as an argument to load an existing entity's details. The new assistant GUI element design and interactions will be fully explained in chapter 7.

6 Design and Implementation: The Validation Function

6.1 Warning Report and Scenario Logs Compilation

The validation function serves two purposes: it checks for any errors in the model and while doing so puts together the explanatory scenario logs.

6.1.1 Warnings

Any problems identified by the DUCK after model validation are communicated to the user through the warning report. These warnings are formatted as follows:

"*Warning (*<Problem Location>*):* <Problem Description>*."*

If no warnings were added to the list, the warning report will instead read:

"No problems found!"

When this message is seen, the user will know that the model is valid.

6.1.2 Problem Location and Flow Step Indexing

Depending on where a problem was discovered, an appropriate pointer would be given to guide the user. This may be the diagram, the name of a scenario and potentially the step of its flow. To clearly identify the step, its index in the main flow is given. If this happens to be an included use case, the step's index in that use case is given too:

Step X of main flow, [step Y of* <Included Use Case Name>*]

These indexes are given as strings and are assigned in advance. These indexes are then used both for locating problems and identifying steps in scenario logs.

6.1.3 Scenario Logging

Apart from the warning list, the function creates a set of scenario logs as flow proofs are performed. A scenario log contains an entry for each step of the flow, with the corresponding

index, plus one “final state” step. Unlike the warning list, which is returned directly by the function, scenario logs are added to the Knowledge Base to be retrieved when the user views them.

6.2 Scenario-Knowledge Base Checks

The validation function checks that any actors mentioned in scenarios have an actor entity in the knowledge base.

“Warning (*<Scenario Name>*): Actor* <Actor Name> *does not exist in the Knowledge Base.”

Later, the function goes through the steps of the scenario’s flow when the proof is performed. After retrieving the actor’s name from a FlowStep tuple, failure to match it with a corresponding Knowledge base entity would additionally point to the index of the offending flow step:

“Warning (*<Scenario Name>*, step* <Flow Step Index>*): Actor* <Actor Name> *does not exist in the Knowledge Base.”

Similarly, an unknown action would be reported:

“Warning (*<Scenario Name>*, step* <Flow Step Index>*): Action <Action Name> does not exist in the Knowledge Base.”

If the actor exists, their action list is retrieved to check that that the given action appears in it. If it is found that this actor should not be able to perform the action, the DUCK would say:

“Warning (*<Scenario Name>*, step* <Step Index>*): Actor* <Actor Name> *does not have action <Action Name>.”

6.3 Flow Proofs

6.3.1 State of the Domain

Throughout the flow proof, the “current state” of the domain is represented as a list of state triples. In the beginning of a scenario flow, the current state is formed by the preconditions. As the function iterates through the flow steps, the current state is updated as it is changed by actors performing actions. The current state before each step is recorded as part of scenario logs.

The open-world assumption implies that if the truth value of a statement is not given, the truthfulness of the statement is unknown. The closed-world assumption rules that unless the truth value of a statement is known to be true, the statement is considered false. The validation function operates on the middle ground between the two. That is, if an entity-state combination is not given in the current state, the program will not assume a state triple to hold false. In short, truthfulness of the entire state triple is considered during validation.

For example, consider that a precondition state triple for a person to walk through a door is **[Door, Locked, False]**. The fact that **[Door, Locked, True]** is not part of the current state would not imply that the door is not locked, therefore the person may not walk through the door unless it is explicitly stated to not be locked.

It is then up to the user to choose the level of completeness in domain model definition. They may initialise all state triple combinations in a scenario’s preconditions or only introduce state triples as actions are performed.

6.3.2 Flow Step Walkthrough

If a flow step's actor and action both exist in the Knowledge Base, the function would then retrieve the action's preconditions state triple list. It then attempts to find an exact match for each in the current state to see if the preconditions for that action are satisfied at that point. If that is not the case, the user will be informed:

***"Warning (*<Scenario Name>, <Flow Step Index>*): Precondition (*<State Triple>*) not satisfied for action* '<Action Name>'."**

If all action's preconditions are satisfied, however, the function will retrieve its postconditions and update the current state accordingly to reflect the action's impact on the domain. This process is repeated until the end of the main flow is reached. Finally, the function compares the current state against the scenario's postconditions to determine if the main flow leads to their satisfaction. If that was not the case, the report will say:

***"Warning (*<Scenario Name>*): Postcondition (*<State Triple>*) not satisfied*."**

Since the current state is not updated with an action's postconditions if its preconditions are not satisfied, the user is likely to see a chain reaction that in the end leads to the scenario's postconditions not being satisfied either. If they see this last warning on its own, the cause must be either a missing flow step or a missing postcondition in one of its actions. After the flow proof, the current state is added to the scenario log under the "Final State" index. The scenario log is added to the Knowledge Base.

6.4 Diagram Validation

6.4.1 Diagram-Knowledge Base

For each scenario, the validation function first makes a list of actor-usecase "relation triples" (not to be confused with state triples) for the given primary actor and any secondary actors to create a list of relations that the diagram should show. For convenience, an auxiliary "relation triple" class was created. Objects of this class are only used by the validation function and are thus not stored in the Knowledge Base.

The function then iterates through the list of GridElement objects – elements present on the diagram. The getClass function is used to sort them into a list of entities ("Actor" or "UseCase") and relations ("Relation") or note the presence of a system boundary ("Generic"). The function also checks if the panel attributes correspond to an entity in the Knowledge Base. If not, this will be reported:

"*Warning (Diagram): Actor '*<Actor Name>*' does not exist in the Knowledge Base.*"

Note that unlike the Scenario-Knowledge Base checks which points to the offending scenario, this warning points to the diagram. The scenario warning has no reference to Knowledge Base:

"*Warning (Diagram): No scenario exists for the use case '*<Use Case Name>*'.*"

The function then iterates through the two lists to get the coordinates of entities' rectangles and relations' stickable points. For each relation, it takes these coordinates and checks if they fall within the bound of any element rectangles, in which case it records their panel attributes and class. Finally, it gets the relation element's panel attributes, determining the type of relation.

6.4.2 Relation UML Conformance and Consistency with Scenarios

Knowing the classes of the two elements and the relation type, DUCK checks whether this relation is UML-compliant. Since only actors and use cases were included in the entity list, they

are the only two entity types for which relations are identified. The system boundary is not included in the "entity" list despite being a Knowledge Base entity and having a "relation" to use cases. This is to avoid the cases where a stickable point's coordinates fall within the borders of a system box, meaning it could be picked up the system as one of the that relation's elements instead of whichever element the relation is meant to connect.

The following combinations would be identified as not UML-compliant:

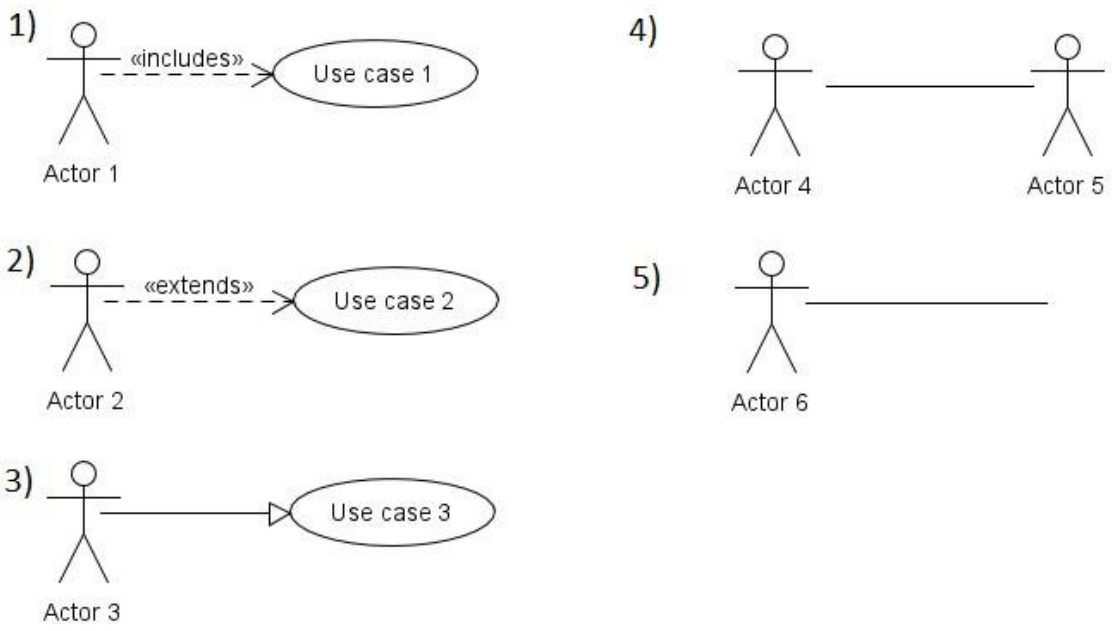


Figure 21. UML non-compliant relations

1. The relation is "includes", one of the elements is an actor.
2. The relation is "extends", one of the elements is an actor.
3. The relation is "abstraction", the elements are of different classes.
4. The relation is "actor-usecase", the elements are of the same class.
5. For any relation, one or both elements are "nothing".

This will be communicated to the user as:

"*Warning (Diagram): Cannot connect* <Element 1 Name> *(*<Element 1 Class>*) and*

<Element 2 Name> *(*<Element 2 Class>*) with an '*<Relation Type>*' relation.*"

However, if an "actor-usecase" relation is UML-compliant, DUCK checks it against the list compiled earlier when going through scenarios. If the DUCK doesn't find a match, it will say:

"Warning (Diagram): Actor '*<Actor Name>*' is not mentioned in use case '*<Use Case Name>*', but a connection was drawn."

If it does find a match, the relation triple is removed from the list. Once the iteration through diagram relation elements is complete, if there are any relation triples left in the list, DUCK will say:

"*Warning (Diagram): Use case '*<Use Case Name>*' has a reference to actor '*<Actor Name>*', but no connection was drawn.*"

6.4.3 System Boundary UML Conformance

The next UML compliance check is the presence of a system boundary. If during the initial iteration none of the elements were of the "Generic" class, which would be reported as:

"Warning (Diagram): System boundary was not placed."

If the system boundary was placed, however, DUCK will check that all use case elements are placed withing its borders and all actor elements are placed outside. As with relations, this is done by getting their rectangles and comparing the coordinate ranges. The DUCK can then say:

"Warning (Diagram): Actor '*<Actor Name>*' was placed within the system boundary."

and

"Warning (Diagram): Use caser '*<Use Case Name>*' was placed outside the system boundary."

This is the last check of the validation function after which the warning list is returned.

7 Design and Implementation: The User Interface

7.1 User Interface Classes

All new GUI classes were created in the com.baselet.gui package of the umlet-swing module and are listed as follows:

1. ActionPanel.java
2. ActorPanel.java
3. DUCKHelpPanel.java (unfinished NFR8)
4. EARSPanel.java (unfinished FR1.1)
5. KnwledgeBasePanel.java
6. RequirementsPanel.java (unfinished FR1.1)
7. ReportPanel.java
8. ScenarioListPanel.java
9. ScenarioLogPanel.java
10. ScenarioPanel.java
11. SystemBoundaryPanel.java

The package contains a dedicated class for all integrated panels (Knowledge Base, scenario list validation) and all “named” entity windows (also called panels due to being an extension of JPanel, as is UMlet’s naming convention). Dialog popups are used to populate tables with the “nameless” structures (state triples, flow step actor-action tuples).

The interfaces were built and integrated using Java Swing’s components such as JTextField, JButton, JComboBox and JOptionPane. Tables were build using a combination of Swing’s JTable, DefaultTableModel and the setViewportView() function. Panels 3, 5, 6, 7 and 8 were integrated with existing UMLet UI using a chain of nested JSplitPanes (section 4.2.2).

7.2 Menu Modifications

First additions to the GUI were the menu modifications which enabled access to the DUCK functionality. Integrated panels are accessed through the new “Assistant” dropdown menu found in the upper toolbar. Windows of scenarios, actors and the system can be accessed by selecting the element on the diagram and choosing “Properties” or using the Ctrl+I shortcut (the only DUCK shortcut). This makes a call to the showProperties() function from an instance of the DUCK Handler

(section 5.6). A new function had to be added to BaseGUI.java in umlet-swing to get the selected element. Addition of these new menu constants was the only time a file from the umlet-elements module was modified.

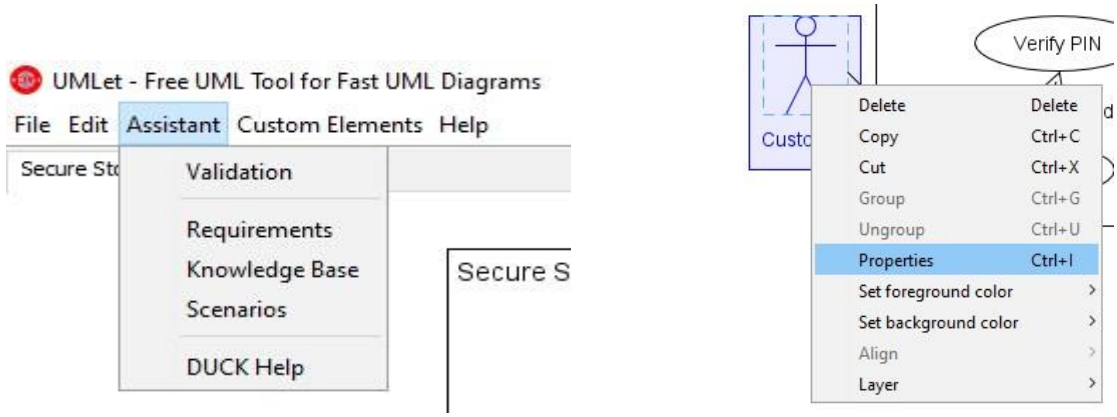


Figure 13. The new assistant drop-down menu (left) and the new Properties option in a context menu on after right-clicking on a diagram element (right).

7.3 Entity Windows

7.3.1 Actor and System Windows

As explained in section 5.3.2, actor and system classes are similar, thus the GUIs are similar too. The windows consist of a text field for the entity's name and a table for its list of actions with buttons to add and delete them. Clicking the "Add" brings up a popup where the action to be added is chosen from a JComboBox dropdown menu. Pressing the "Save" button adds the entity to the Knowledge Base, retrieving the name from the JTextField and the actions from the JTable.

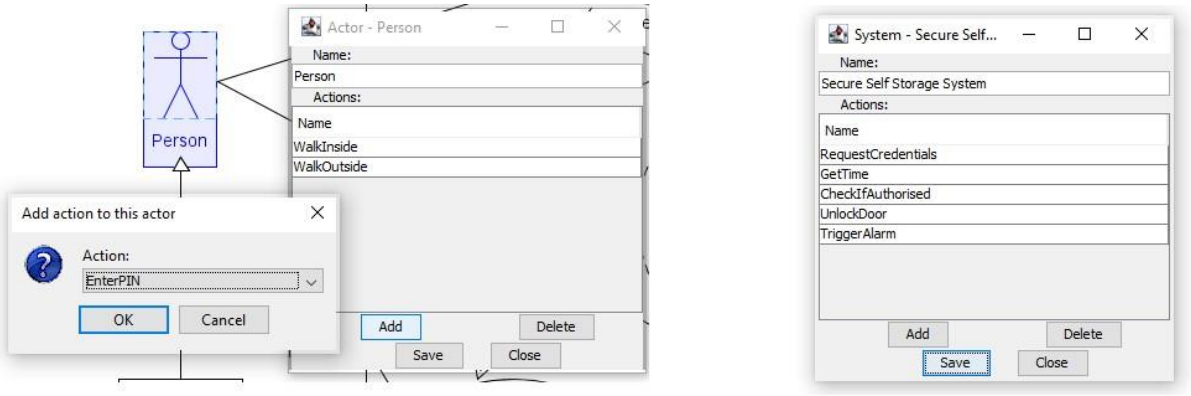


Figure 22. Actor window with the "Add action to this actor" dialog (left) and a system window (right).

7.3.2 Action Window

Apart from a JTextField for the action's name, the action window has two tables. The entity and state JComboBoxes allow the user to choose from existing entities and states or enter them manually; if the given entity or state does not already exist in the knowledge base, a new domain object and a state respectively would be added to the KB. The last value dropdown is "disabled" as only two options are permitted – True or False.

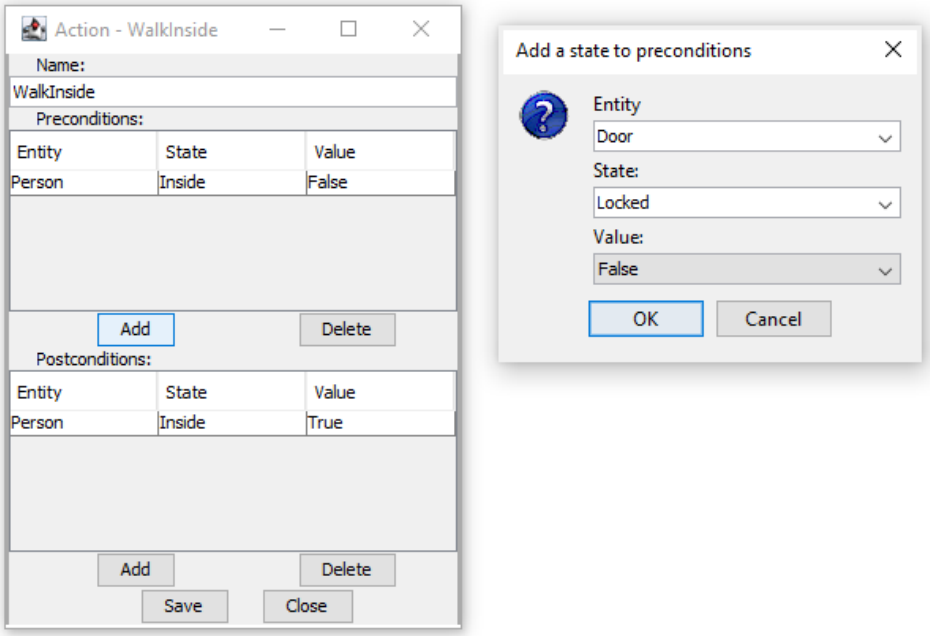


Figure 23. Action window (left) and its "Add a state to preconditions" dialog (right)

7.3.3 Scenario Windows

The. Resembles … The primary actor's name is entered in a JTextField and the secondary actors are added to the table below it. This is followed by the precondition and postcondition tables as in the action window. Lastly, a table is used to give the steps of the main flow which are automatically labelled with the step numbers. In the flow step popup, the actor and action combination is specified using the JComboBox dropdown menus. They are disabled, so only existing entities can be used. For reasons mentioned in section 5.4.2, the "Includes" option and existing scenario names are available to represent an included scenario as a main flow step.

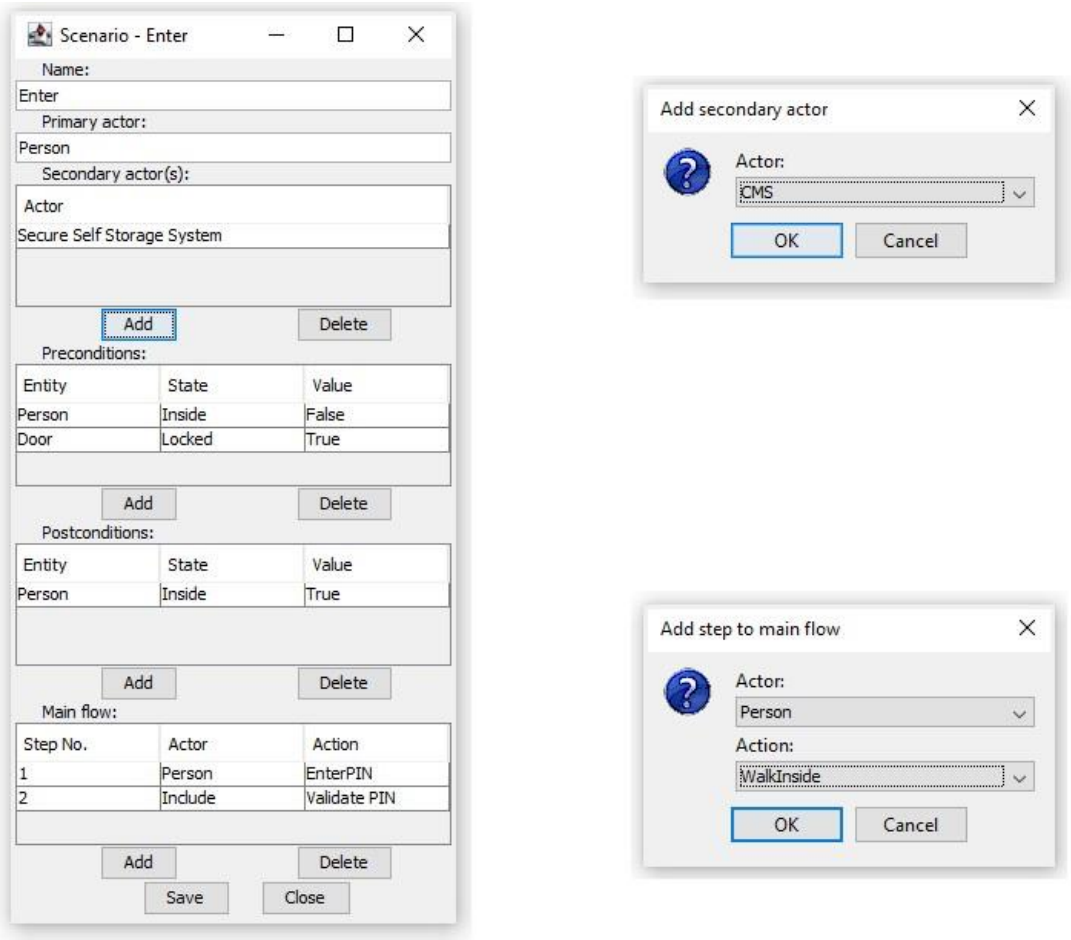


Figure 24. Scenario window (left) with its secondary actor and main flow step dialogs (left)

7.4 Knowledge Base and Scenario List Panels

The Knowledge Base and scenario list panels are similar in design – both consist of a table and a set of buttons to manipulate its contents. The table of Knowledge Base entities has the entity name and entity type as its columns, for scenarios the scenario name and primary actor are given. As future work, these tables could also be made to display the number of warnings for each entity or scenario by linking them to the validation function.

Clicking the “New Entity” button on the KB panel would bring up a dialog where the new entity’s name would be given and entity type selected from the 4 available options: Actor, Action, Object and State. Its “New Scenario” counter in the scenario list panel would only ask for a name.

As an alternative to accessing entity windows from diagram elements, the user may can select an entity in the Knowledge Base panel and press “Edit”. Scenario windows can be accessed in a similar fashion through the scenario list panel. Finally, the Knowledge Base panel gives the user the option to delete the entity from the KB by selecting it and pressing “Delete”. Same applies for scenarios in the scenario list panel. The “close” button can be used to retract the panel.

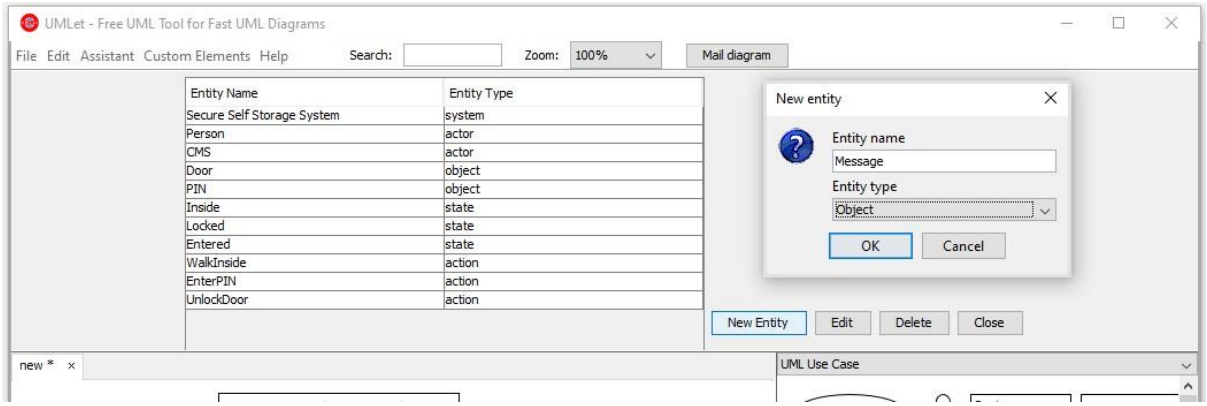


Figure 25. Adding a new object entity to the Knowledge Base

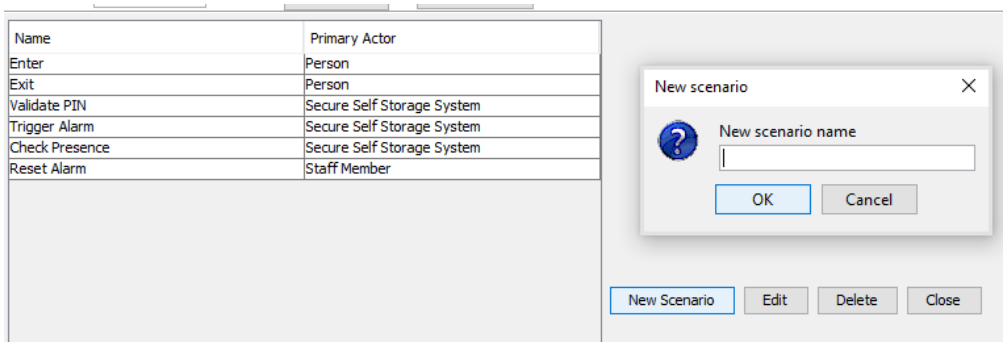


Figure 26. Scenario table of the scenario list panel7.4 Validation and Scenario Logs Interface

7.5 Validation and Scenario Logs Interface

7.5.1 Validation (Report) Panel

The validation panel contains the warning report to display which a Swing JTextArea component was used. Below is a button used to validate the model – on press it triggers the checkConsistency() function. The “View Scenario Logs” is used to access the scenario logs.

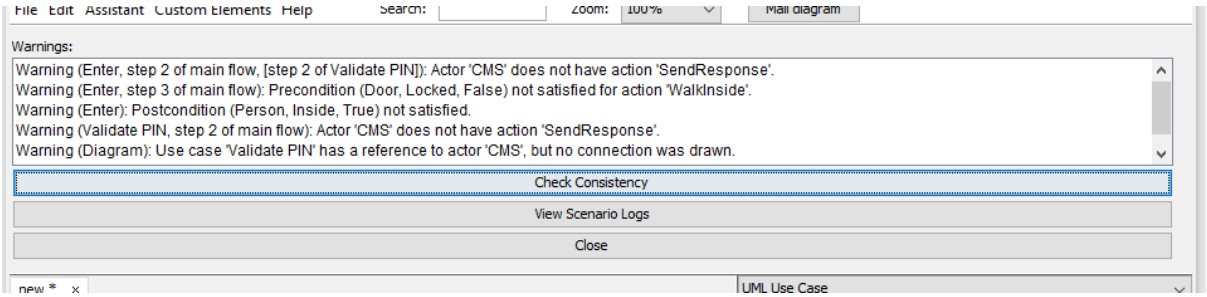


Figure 27. The validation (report) panel

7.5.2 Scenario Log Window

A scenario log window is used to view the step-by-step walkthrough of a scenario. The JComboBox at the top is used to select the scenario whose logs are to be displayed, then when the "Load" button is pressed, the respective Scenario Log object will be fetched from the Knowledge Base. The Log Step class (section 5.5.2) stores the actor and actions name as well as the action's pre and postconditions separately instead of using flow step objects because all this information was readily available in the validation function when the log step was compiled. Having it in this decomposed format allows to populate the fields and tables of the scenario log window faster as nothing has to be retrieved from the Knowledge Base. The "< Previous Step" and "Next Step >" buttons are used to navigate the flow.

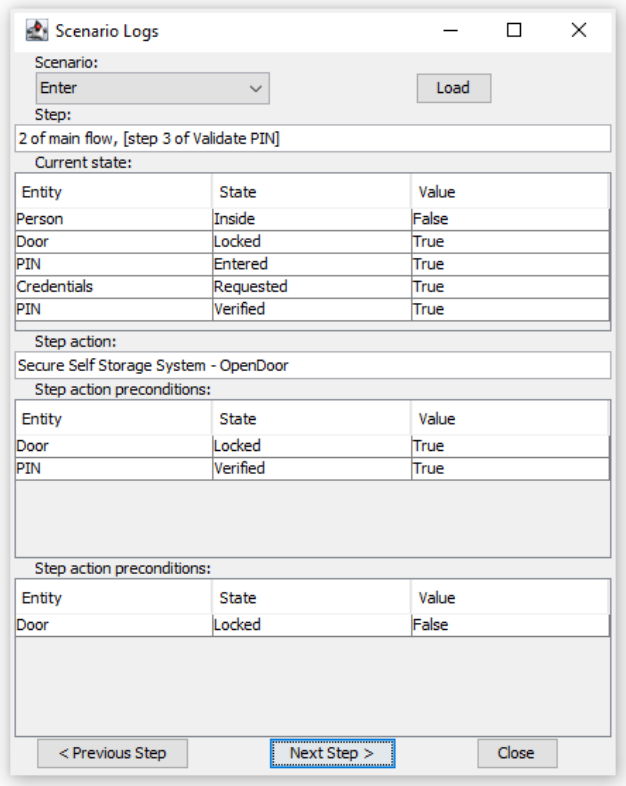


Figure 28. Scenario log window

8 Evaluation

8.1 Unit Testing

Though unit testing was supposed to be done throughout development, program architecture made it complicated. Because most functions also interact with the GUI (e.g. the validation function updates the warning report of the validation panel), executing them without initialising the GUI will result in an error. Unit testing could be accommodated for by reformatting the functions, decomposing them into separate smaller ones responsible exclusively for handling the backend or frontend. The application still underwent thorough manual testing examples of which can be seen below:

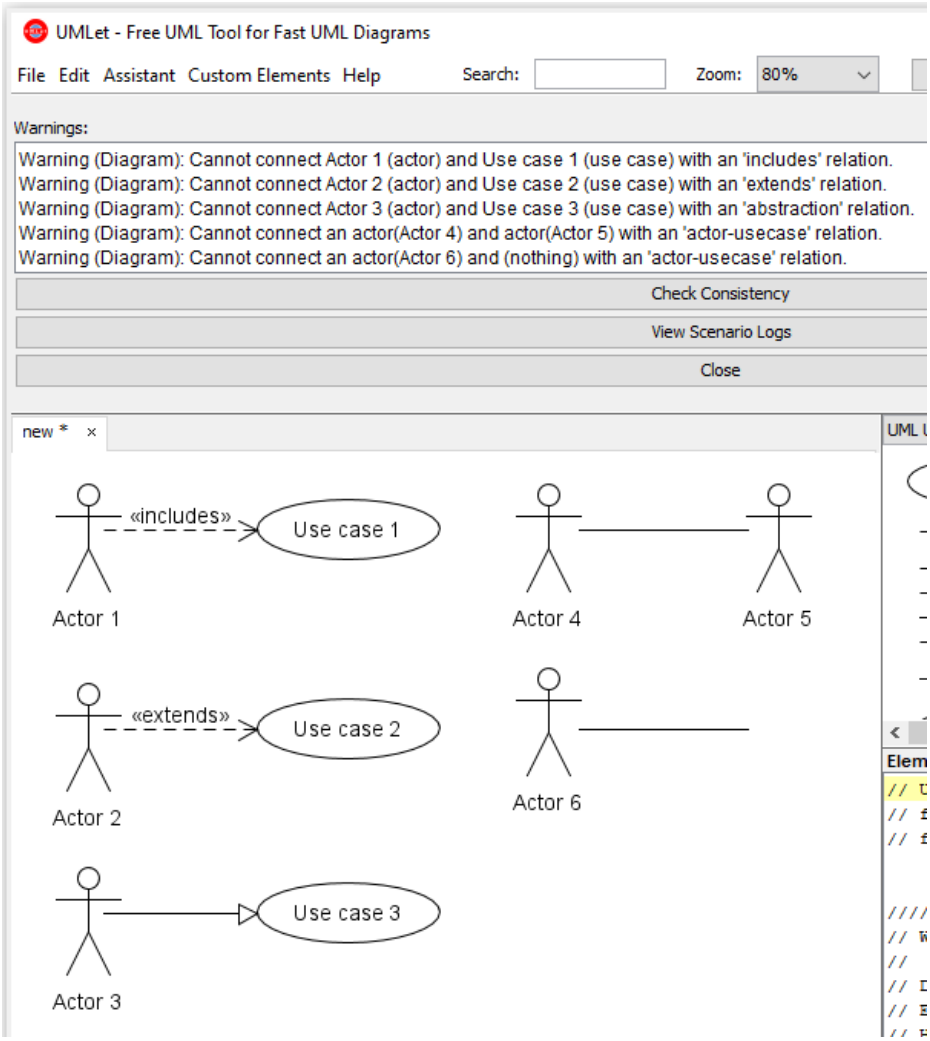


Figure 29. Warnings for UML non-conformant relations on the diagram

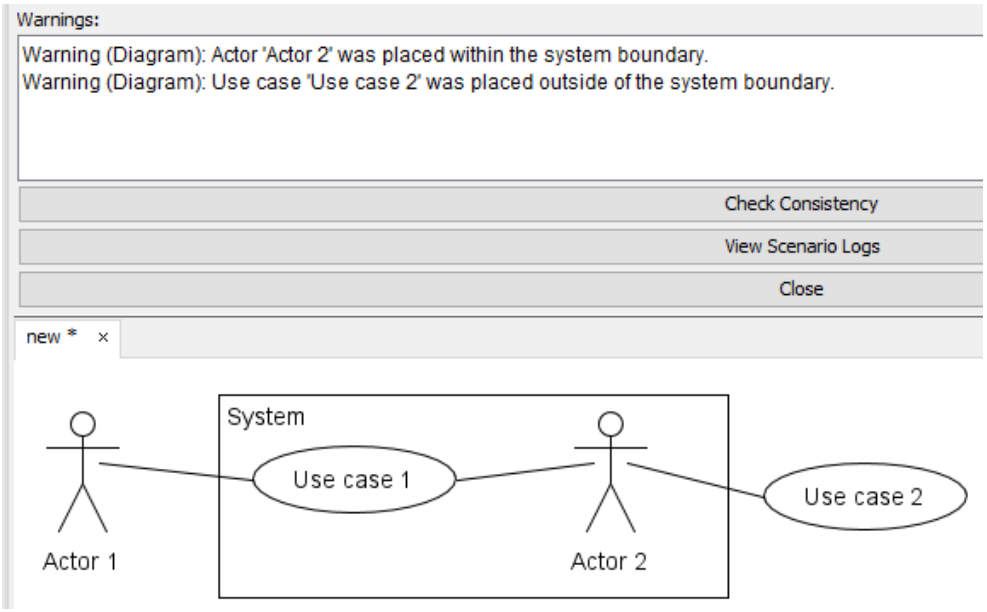


Figure 30. Warnings for incorrect element placement in relation to the system boundary

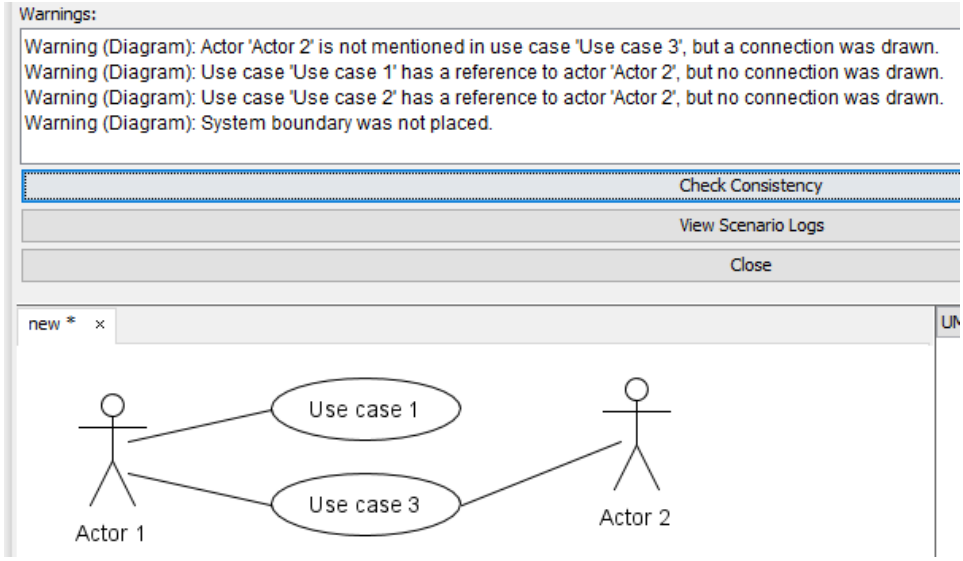


Figure 31. Warnings for not having a system boundary on the diagram and inconsistency between scenarios and the diagrams

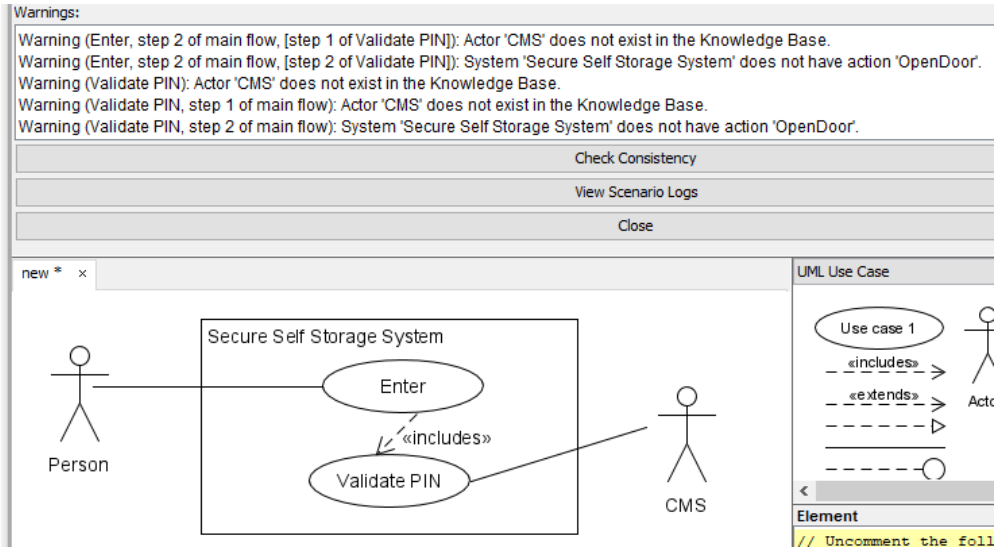


Figure 32. Knowledge base checks

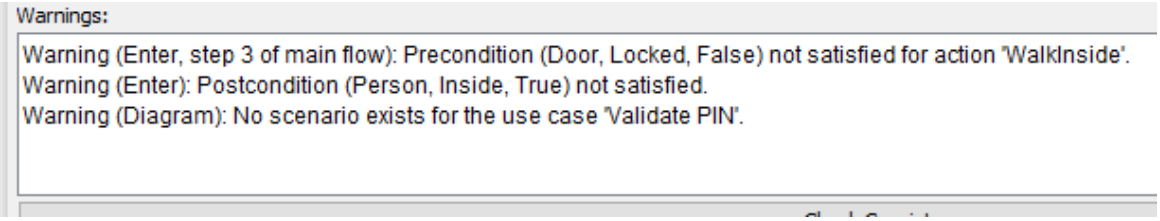


Figure 33. Warning about action's preconditions/ scenario's postconditions not being satisfied with a followed by a "no scenario exists" warning

8.2 Case Study Testing

While unit testing is concerned with the correct operation of program, the aim of case study testing was to see how well the DUCK methodology accommodates for the different of use case modelling. Examples and exercises from the "Writing Effective Use Cases" book by Alistair Cockburn were used for case study testing (see appendix for full details).

8.2.1 Correctly Written Scenario Example

In Use Case 39 (p. 251) "Buy Stocks Over the Web", the system under consideration is PAF (no backronym given) used to maintain a stock portfolio.

Dropping a couple of details that are beyond the assistant application's scope, the use case scenario is as follows:

Primary Actor: Purchaser/User

Scope: PAF

Precondition: User already has PAF open.

Success Guarantees: Remote site has acknowledged the purchase; PAF logs and the user's portfolio are updated.

Main Success Scenario:

1. User selects to buy stock over the web.
2. PAF gets name of website to use (E*Trade, Schwab, etc.)
3. PAF opens web connection to the site, retaining control.
4. User browses and buys stock from the website.
5. PAF intercepts the responses from the website and updates the user's portfolio.
6. PAF shows the user the new portfolio standing.

Using DUCK methodology entities, this can be represented in the Knowledge Base as follows:

The system:

Attribute	**Value**
Name	PAF
Actions	GetWebsiteName, OpenConnection, InterceptResponse, UpdateLogs, UpdatePortfolio, ShowPortfolio

Table 7. Tabular representation of the PAF system boundary object in the "Buy Stock Over the Web" example

The actor map:

Key	**Value**	
Purchaser	**Attribute**	**Value**
	Name	Purchaser
	Actions	SelectWebsite, BuyStock
Website	**Attribute**	**Value**
	Name	Website
	Actions	SendResponse

Table 8. Tabular representation of the actor map for the "Buy Stock Over the Web" example

Object List: Connection, Logs, WebsiteName, Portfolio, Purchase, Response, Stock

State List: Bought, Opened, Received, Selected, Sent, Shown, Updated

The action map:

Key	**Value**			
SelectWebsite	**Attribute**	**Value**		
	Name	SelectWebsite		
	Preconditions	**Entity**	**State**	**Value**
		Website	Selected	False
	Postconditions	**Entity**	**State**	**Value**
		Website	Selected	True
GetWebsiteName	**Attribute**	**Value**		
	Name	GetWebsiteName		
	Preconditions	**Entity**	**State**	**Value**
		PAF	Open	True
		Website	Selected	True
	Postconditions	**Entity**	**State**	**Value**
		WebsiteName	Received	True
OpenConnection	**Attribute**	**Value**		
	Name	OpenConnection		
	Preconditions	**Entity**	**State**	**Value**
		WebsiteName	Received	True
	Postconditions	**Entity**	**State**	**Value**
		Connection	Open	True
BuyStock	**Attribute**	**Value**		
	Name	BuyStock		
	Preconditions	**Entity**	**State**	**Value**
		Website	Selected	True
	Postconditions	**Entity**	**State**	**Value**
		Stock	Bought	True
SendResponse	**Attribute**	**Value**		
	Name	SendResponse		
	Preconditions	**Entity**	**State**	**Value**
		Stock	Bought	True
	Postconditions	**Entity**	**State**	**Value**
		Response	Sent	True
InterceptResponse	**Attribute**	**Value**		
	Name	InterceptResponse		
	Preconditions	**Entity**	**State**	**Value**
		Connection	Open	True
		Response	Sent	True
	Postconditions	**Entity**	**State**	**Value**
		Response	Received	True
UpdateLogs	**Attribute**	**Value**		
	Name	UpdateLogs		
	Preconditions	**Entity**	**State**	**Value**
		Response	Received	True
	Postconditions	**Entity**	**State**	**Value**
		Logs	Updated	True

UpdatePortfolio	**Attribute**	**Value**		
	Name	UpdatePortfolio		
	Preconditions	**Entity**	**State**	**Value**
		Response	Received	True
	Postconditions	**Entity**	**State**	**Value**
		Portfolio	Updated	True
ShowPortfolio	**Attribute**	**Value**		
	Name	ShowPortfolio		
	Preconditions	**Entity**	**State**	**Value**
		Portfolio	Updated	True
	Postconditions	**Entity**	**State**	**Value**
		Portfolio	Shown	True

Table 9. Tabular representation of the action map for the "Buy Stock Over the Web" example

A problem that became apparent very early on was the inability to use the PAF system object in a state triple because associated states are not supported for system boundary entities as mentioned in section 5.3.2 and 7.2.1. To remedy this, a PAF domain object had to be created, however such a duplication is not an ideal solution as it will create confusion between the two entities.

Finally, these objects are used to put together the scenario:

Property	**Value**		
Class	Scenario		
Name	Buy Stock Over the Web		
Primary actor	Purchaser		
Secondary actors	PAF, Website		
Preconditions	**Entity**	**State**	**Value**
	PAF	Open	True
	Website	Selected	False
Postconditions	**Entity**	**State**	**Value**
	Response	Received	True
	Logs	Updated	True
	Portfolio	Updated	True
Main flow	**Actor**	**Action**	
	Purchaser	SelectWebsite	
	PAF	GetWebsiteName	
	PAF	OpenConnection	
	Purchaser	BuyStock	
	Website	SendResponse	
	PAF	InterceptResponse	
	PAF	UpdateLogs	
	PAF	UpdatePortfolio	
	PAF	ShowPortfolio	

Table 10. Tabular representation of the scenario object in the "Buy Stock Over the Web" example

Scenario logs steps for as shown by the scenario log window correct run (omitting action pre/postconditions):

Log Step	**Field/Table**	**Value**		
1	Step index	Step 1 of main flow		
	Current state	**Entity**	**State**	**Value**
		PAF	Open	True
		Website	Selected	False
	Action	Purchaser - SelectWebsite		
2	Step index	Step 2 of main flow		
	Current state	**Entity**	**State**	**Value**
		PAF	Open	True
		Website	Selected	True
	Action	PAF - GetWebsiteName		
3	Step index	Step 3 of main flow		
	Current state	**Entity**	**State**	**Value**
		PAF	Open	True
		Website	Selected	True
		WebsiteName	Received	True
	Action	PAF - OpenConnection		
4	Step index	Step 4 of main flow		
	Current state	**Entity**	**State**	**Value**
		PAF	Open	True
		Website	Selected	True
		WebsiteName	Received	True
		Connection	Open	True
	Action	Purchaser - BuyStock		
5	Step index	Step 5 of main flow		
	Current state	**Entity**	**State**	**Value**
		PAF	Open	True
		Website	Selected	True
		WebsiteName	Received	True
		Connection	Open	True
		Stock	Bought	True
	Action	Website - SendResponse		
6	Step index	Step 6 of main flow		
	Current state	**Entity**	**State**	**Value**
		PAF	Open	True
		Website	Selected	True
		WebsiteName	Received	True
		Connection	Open	True
		Stock	Bought	True
		Response	Sent	True
	Action	PAF - InterceptResponse		

7	Step index	Step 7 of main flow		
	Current state	**Entity**	**State**	**Value**
		PAF	Open	True
		Website	Selected	True
		WebsiteName	Received	True
		Connection	Open	True
		Stock	Bought	True
		Response	Sent	True
		Response	Received	True
	Action	PAF - UpdateLogs		
8	Step index	Step 8 of main flow		
	Current state	**Entity**	**State**	**Value**
		PAF	Open	True
		Website	Selected	True
		WebsiteName	Received	True
		Connection	Open	True
		Stock	Bought	True
		Response	Sent	True
		Response	Received	True
		Logs	Updated	True
	Action	PAF - UpdatePortfolio		
9	Step index	Step 9 of main flow		
	Current state	**Entity**	**State**	**Value**
		PAF	Open	True
		Website	Selected	True
		WebsiteName	Received	True
		Connection	Open	True
		Stock	Bought	True
		Response	Sent	True
		Response	Received	True
		Logs	Updated	True
		Portfolio	Updated	True
	Action	PAF - ShowPortfolio		
10	Step index	Final State		
	Current state	**Entity**	**State**	**Value**
		PAF	Open	True
		Website	Selected	True
		WebsiteName	Received	True
		Connection	Open	True
		Stock	Bought	True
		Response	Sent	True
		Response	Received	True
		Logs	Updated	True
		Portfolio	Updated	True
		Portfolio	Shown	True
	Action	none - none		

Table 5. Tabular representation of the scenario log for the "Buy Stock Over the Web" example

It can be seen that aside from the **[Website, Selected, False]** state triple given as precondition for the scenario and SelectWebsite action (which in fact is not really necessary), no other state triple ever had a value of False. However, as justified with the **[Door, Locked, True] → [Door, Locked False]** example, the Boolean values should be kept.

In log step 5 the **[Response, Sent, True]** state triple is added to the current state followed by **[Response, Received, True]** in step 6. The fact that "the response was sent" is carried through rest of the scenario logs even though it loses some relevance once it served its purpose as a precondition to step 6 where "the response was received" domain condition is established. The same thing can be said about the **[Portfolio, Updated, True]** and **[Portfolio, Shown, True]** state triples. The application could possibly allow to remove state triples or update their state as well as the value.

Cockburn describes such an approach to defining actions as "a bit too small" but having "the advantage of steps that are separably testable units, possibly suited for a more formal development situation" (p. 94). A more moderate approach can be tried by combining actions to closer resemble the steps given in scenatio flow:

Key	**Value**			
SelectWebsite	**Attribute**	**Value**		
	Name	SelectWebsite		
	Preconditions	**Entity**	**State**	**Value**
		Website	Selected	False
	Postconditions	**Entity**	**State**	**Value**
		Website	Selected	True
ConnectToWebsite	**Attribute**	**Value**		
	Name	ConnectToWebsite		
	Preconditions	**Entity**	**State**	**Value**
		PAF	Open	True
		Website	Selected	True
	Postconditions	**Entity**	**State**	**Value**
		Connection	Open	True
BuyStock	**Attribute**	**Value**		
	Name	BuyStock		
	Preconditions	**Entity**	**State**	**Value**
		Website	Selected	True
	Postconditions	**Entity**	**State**	**Value**
		Stock	Bought	True

SendResponse	**Attribute**	**Value**		
	Name	SendResponse		
	Preconditions	**Entity**	**State**	**Value**
		Stock	Bought	True
	Postconditions	**Entity**	**State**	**Value**
		Response	Sent	True
HandleResponse	**Attribute**	**Value**		
	Name	UpdateLogs		
	Preconditions	**Entity**	**State**	**Value**
		Response	Sent	True
	Postconditions	**Entity**	**State**	**Value**
		Logs	Updated	True
		Portfolio	Updated	True
		Portfolio	Shown	True

Table 6. Composite action definition

Consecutive PAF actions were merged to reduce the total number of actions which allows for a shorter scenario flow. A shorter scenario log would be produced. An assumption was made for the HandleResponse action that a response being sent by the website is guaranteed to be intercepted by PAF, which may not always be the case. This can compromise the model's quality, so a separate InterceptResponce action may still be necessary. However, action combinations which are less likely to fail and are like updating the logs and portfolio then showing the latter can be safely combined.

8.2.2 "Common Pitfalls" Examples

In his book, Cockburn also talks about how a use case scenario should NOT be written. Two cases of that are not giving the actions of system under consideration and not giving the actions of external actors. As part of the validation function, the DUCK checks that a system boundary element is placed on the diagram. While it does not then check whether any actions are assigned to it or that the system is included in the scenario flows, this may serve as a reminder to the user that the system is part of the use case model.

8.3 Usability Testing

8.3.1 Revised Method

Originally, user testing was meant to have the participants complete a set of tasks using the application, then give a judgement of the application's usability based on their experience interacting with it. This assumed that an executable of the assistant application will be made and shared with the testers digitally to allow for asynchronous participation. An attempt was made to build the DUCK executable using Launch4j tool as prescribed by the UMLet developer guide (GitHub, 2023), however it has been unsuccessful. A .jar (Java ARchive) file of the code was created but attempts to process it int an executable failed due to exception "Class Not Found" exception being thrown for two existing UMLet classes. The two classes in question were not modified during the assistant application's creation, so the error was likely caused by some mismatch in UMLet's build version or the Java version used.

Regardless, this meant that the only option for participants to try the DUCK would be to use it on the facilitator's laptop. The logistics of this approach were deemed to be too inconvenient for everyone involved, and so it was decided to change the format of the usability study. Instead of interacting with the system, the participants were asked to watch a video demonstration of the application, then give their impressions about the perceived usability of the system. The consent form and questionnaire were changed accordingly.

Another deviation from the original usability testing plan was that only a single Year 2 student volunteered to participate. As a mitigation outlined in the risk analysis, two additional participants were recruited elsewhere: one was a MEng student and the other in Year 4 of the BSc Artificial Intelligence programme. This is thought to not have significantly impaired the results of the study as a sample of only a few people is still mostly sufficient to point out the most significant issues with the design. The broader range of participants' levels of study and thus experience might have actually given a more balanced view of the application's usability.

8.3.2 Usability Study Results

Instead of the standard 10-question System Usability Scale, a custom Lickert scale questionnaire was used to gather feedback about the DUCK's individual parts. An optional "additional comment or suggestions" space was also included so that the testers could give more specific feedback. The participants' answers are given below:

1. I found the user interface of the …

	Poorly Designed	**Adequately Designed**	**Well Designed**	**Very Well Designed**
Knowledge base panel	0	1	2	0
Knowledge base entity forms	1	0	1	1
Use case scenario forms	1	1	0	1
Validation panel	0	0	3	0
Scenario log window	0	1	1	1

Table 7. Question 1 results as a table

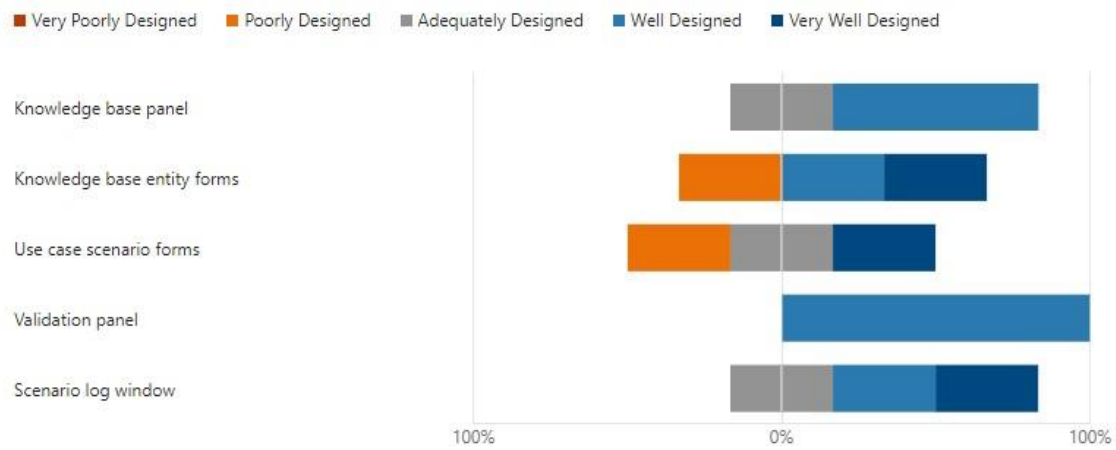


Figure 34. Question 1 results visualised

2. I think that most users would find …

	Difficult	**Easy**	**Very Easy**
Adding entities to the knowledge base	0	1	2
Accessing an existing entity's details	1	1	1
Editing a use case scenario form	1	1	1
Understanding the warning report	0	2	1
Understanding a scenario flow	0	1	2

Table 8. Question 2 results as a table

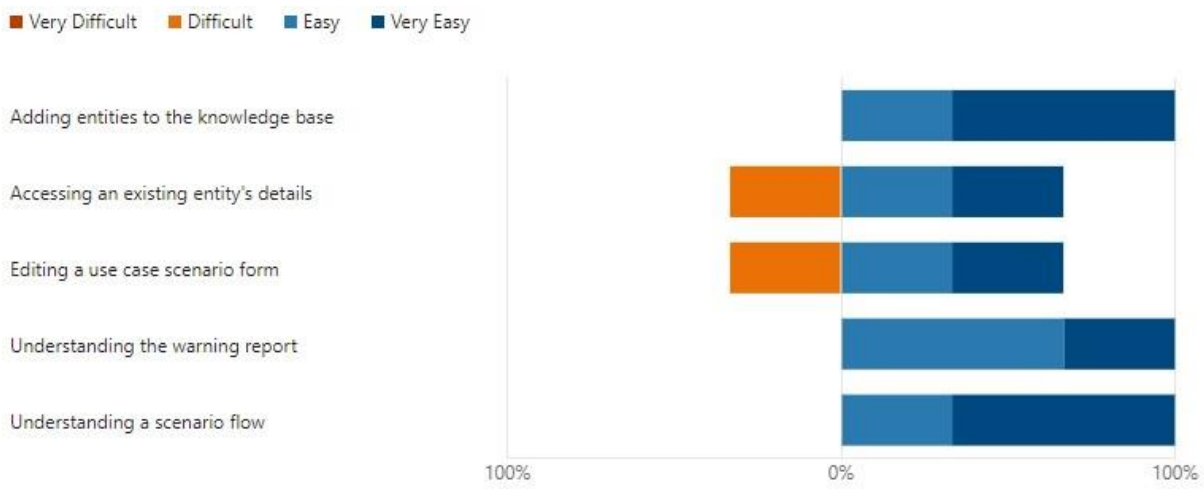


Figure 35. Question 2 results visualised

3. Overall, I found the …

	Adequate	Good	Very Good
Integration of DUCK's interface into UMLet	1	0	2
Consistency between DUCK's interface elements	1	1	1

Table 95. Question 3 results as a table

Figure 356. Question 3 results visualised

4. Additional comments or suggestions

One user commented on the boundaries of text fields and tables going right to the edges of windows. Function was prioritised over form for this project, but indeed, the user interface would need polish if the application were going to be released. Another user commented that it is not obvious that changing a scenario's name then saving it would create a new scenario rather than renaming the existing one. The application should follow general conventions like making a

distinction between “Save” and “Save As” or having a “Are you sure you want to delete X?” dialog to avoid accidental deletion.

8.3.3 Usability Study Conclusions

Overall, the application received positive feedback, responses for all questions skew towards the “good” ratings. The participants’ opinions were divided on some questions, however certain parts of the application were criticised more than others. Knowledge Base entity and scenario forms received a “Very Poorly Designed” rating from one participant and were then judged as “Difficult” to use. This corresponded with the written feedback of question 4 where two participants commented on the scenario form’s subpar quality and behaviour which does not conform to the general conventions of software application interaction design (this applies to many DUCK interface elements). Another problem of the scenario forms is that stacking many text fields, tables and button panels resulted in a design that is very tall, a window of such a height may not fit well on smaller screens. The user would then have to scroll through the secondary actor, precondition/postcondition and flow tables, which is not ideal. This could be remedied by making parts of the form retractable or redesigning the form layout to have a more horizontal placement of the GUI components.

The layouts of other GUI elements need work too. The warning list text area should be made taller as in cases where many warnings is displayed the user has to scroll through them, which is inconvenient. Consistency between the GUI elements can also be improved, for example the placement and width of buttons is different between the Knowledge Base/scenario list panel, validation panel and the entity/scenario forms.

9 Conclusions

9.1 Achievements

The Detailed Use Case Kit methodology and assistant application has proven to be a successful effort in formalising and validating use case models. Most importantly, it proposed a reasonable standard for formalisation of use case scenario flows. Case study testing has shown that the simplicity of the DUCK methodology provides a great degree of flexibility, enabling different levels of completeness and complexity in definition of the domain and use case scenarios. Most concepts could be represented with creative use of the available entities. Aside from a few oversights and unfinished parts, diagram-scenario validation and UML conformance checks have also been successfully implemented.

Though it still very much at the "proof of concept" stage and does not compare in the range of its functionality, it can be argued that the DUCK provides better model validation capabilities than CaseComplete or Enterprise Architect (section 2.2). At any rate, the assistant functionality has been a massive addition to UMLet and so is a significant contribution on open-source software level.

Usability testing has shown that while at this stage the assistant application may not yet be ready to compete with commercial software in terms of GUI quality, however the interface is overall of an acceptable quality. With some polish, it can easily be brought up to a standard of the tools mentioned above.

Below is a summary of the degree to which each of the original requirements was implemented:

Key
Fully or mostly implemented
Partially implemented
Not yet implemented, future work
Additional, not originally planned functionality implemented

Functional Requirements

ID	Description	Priority
FR1	The user shall be able to build the knowledge base.	M
FR1.1	The user shall be able to define a list of requirements using EARS templates.	M
FR1.2	The user shall be able to define a list of actors, their goals, actions, and states.	M
FR1.2.1	The user shall be able to define actor actions and goals in terms of objects, and states.	M
FR1.2.2	The program should support abstractions of actors.	S
FR1.2.3	The program should identify reusable actor goals.	S
FR1.3	The user shall be able to define non-actor objects, their states.	M
FR1.4	The user could be able to generate a checklist for drawing the use case diagram from actor and goal definitions.	C
FR1.5	The user would be able to generate a use case diagram from definitions.	W
FR2	The user shall be able to build a UML use case model.	M
FR2.1	The user shall be able to draw a use case diagram.	M
FR2.2	The user shall be able to generate use case scenario template forms from the diagram.	M
FR2.2.1	The user shall be able to define preconditions and postconditions in terms of actors, objects, and states.	M
FR2.2.2	The user shall be able to define main and alternative flows in terms of actor actions.	M
FR3.1	The user shall be able to check the consistency between the use case diagram and scenario forms.	M
FR3.1.1	The program shall raise a consistency error when: iv. The diagram has actors/use cases with duplicated names OR v. The diagram has actors/use cases that are not connected to any use cases/actors OR vi. An actor/extension case/inclusion case is mentioned in a use case's scenario form but the actor/extension case/inclusion and use case are not connected on the diagram.	M
FR3.1.2	The program should raise an error when the graph does not conform to OMG's UML specifications.	S
FR3.2	The user shall be able to check the consistency within the scenario flows.	M
FR3.2.1	The program shall raise an error when: iii. The scenario flow contains entities whose definition is incomplete OR iv. The flow proof fails.	M
FR3.3	The program should suggest corrections to fix inconsistencies.	S
FR4	The user could be able generate traceability matrices for the model.	C
FR5	The user shall be able to save, open and edit DUCK model files.	M
FR6	The user should be able to view the logs of a scenario flow proof.	S

Table 106. Functional requirements with achievement status

Non-Functional Requirements

ID	Description	Priority
NFR1	The program shall be available as a standalone application.	M
NFR2	The program shall use a new file format to represent the model.	M
NFR3	The program shall use forms and tables and boxes for text input.	M
NFR6	The program could track and update changes and update the model in real time.	C
NFR7	The user could be able to use shortcuts to accomplish the tasks.	C
NFR8	The program could provide tips.	C

Table 117. Non-functional requirements with achievement status

9.2 Limitations

1. For preconditions and postconditions of actions and scenarios, the program does not prevent giving two logically contradictory state triples i.e. with same entity and state but opposite Boolean values. For example, the program permits having both **[Door, Locked, True]** and **[Door, Locked, False]** at the same time. During validation, both state triples would be considered, which will result in unpredictable behaviour.
2. During a flow proof, the validation function does not check that for a given flow step the specified actor is mentioned as the scenario's primary or secondary actor or connected to it on the diagram.
3. There is no checking of consistency between the "included" scenarios in the main flow and their connection on the diagram.
4. Only one level of "included" scenarios is supported, that is, nesting multiple "included" scenarios would produce incorrect results.
5. The validation function does not warn about the diagram having duplicated elements or elements that have no connections. It also does not warn about entities with incomplete definitions or unused entities.
6. In a scenario's main flow table, flow step numbers are not updated when an element is removed. However, this does not impact the logic as a flow step's index in the actual list data structure is used by the backend.

9.3 Future Work

Aside from the obvious fixes, refactoring, and GUI polishing, future work will include implementing the rest of the planned functionality.

9.3.1 Requirements Engineering Functionality

As for this documented period the use case functionality became the primary focus, the priority of requirements engineering functionality was reconsidered. As a result, requirements and goals are not included in the present version of the assistant application.

However, much of the groundwork has been laid when other functionality was being implemented. For example, as in the DUCK methodology actor goals were told to consist of what eventually became known as the state triples. Implementation of the goal functionality would thus not take long; parts of the validation function could also be consulted for checking goal satisfaction or finding reusable goals.

Additionally, some GUI classes were created for requirements: a “requirements panel” with a layout similar to the Knowledge Base and scenario list ones was made, a new EARS template window used to populate that requirements table.

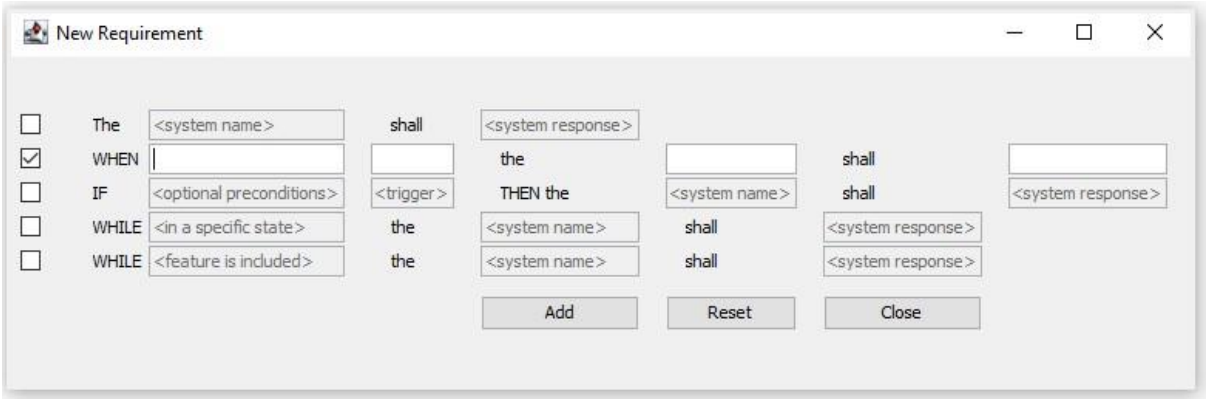


Figure 367. EARS template window concept

9.3.2 Extension Scenarios

The last use case related functionality to be implemented are the extension scenarios. Unlike "included" scenarios, there is less consensus on extension cases (Cockburn, 2001); the lack of a definitive standard could complicate development since the guidelines are not as clear, but on the other hand this provides more freedom for the implementation.

Extension scenarios would require more planning. For example, the action class could be modified to determine if an action is deterministic, that is if performing it would always result in its postconditions. Extension cases would then be conditional, different alternative flows specified depending on state triple values at a given step. The concept of triggers would need to be introduced to scenario flows. A decision must also be made on how additional scenario logs would be produced for extension cases.

9.3.3 Actor Abstraction

With a few modifications, the actor class can be made to support abstraction (generalisation). Similar to how actor entities can be created from the Knowledge Base panel, actor window may include the option to create new actor entity with inherited actions. Since the validation function is already capable of recognising abstraction relations on the diagram, adding abstraction checks should be straightforward. As with actor-usecase relations, a list of abstraction relations could be compiled to be checked against the diagram and vice-versa.

9.3.4 Saving the Knowledge Base as a File

The last functional requirement was the ability to save and load a DUCK model into the assistant application. At the simplest level, this could just involve writing the contents of a Knowledge Base to a text file. Since the model is already formalised by the application, only a few additional syntax rules would need to be created for specifying the types of entities and separating them between each other. The process of saving the DUCK model would thus involve iterating through its Knowledge

Base entity and scenario maps and writing them to the file using these labels and delimiters. To load the model, the assistant application would then create entities and fill scenarios as it parses the file.

Initially, the new file format was planned to include both the diagram and the DUCK model. To do this, a way to append the model to UMLet's .uxf file would have to be found. After some consideration, it was decided that the model should be stored separately, one reason being to avoid the technical complexity of combining the two. Additionally, the user may still want to use the baseline .uxf files either because they want to create DUCK models for existing diagrams or share their diagrams as .uxf files with someone who is not using the DUCK. A possible extension for the new file format is **.drks** which would stand for "**D**-etailed Use Case Kit **R**-equirements, **K**-nowledge Base entities and **S**-cenarios" or "drakes" to keep with the "duck" theme.

9.4 Final Thoughts

This project was largely shaped by the nature of working with existing software code, using an open-source program like UMLet as base for the assistant application allowed to take advantage of its infrastructure. Coding the diagram functionality and GUI from the ground up would no doubt have taken a huge amount of time. However, it also took time to reverse-engineer UMLet code and integrating the assistant functionality into it was not exactly straightforward. Finally, the attempt to build an executable has been unsuccessful due to an issue with existing classes which would require additional knowledge to fix.

Nonetheless, this project was successful at exploring the potential of the Detailed Use Case Kit methodology and the assistant application which automates it. This experiment provided insights which will be used to refine the DUCK vision and carry it forward as its own original software tool.

10 References

1. Albee, A., Battel, S., Brace, R., Burdick, G., Casani, J., Lavell, J., Leising, C., MacPherson, D., Burr, P., Dipprey, D. (2000) 'Report on the Loss of the Mars Polar Lander and Deep Space 2 Missions', *NASA STI/Recon Technical Report N. 61967*. (Accessed 17/11/2023), available at: https://smd-cms.nasa.gov/wp-content/uploads/2023/07/3338_mpl_report_1.pdf
2. Cephas Consulting Group, *UML Validation Extension* (Accessed 28/09/2023), available at: https://enterprisemodelingsolutions.com/ext-arch-valid/
3. Cockburn, A. (2001) 'Writing Effective Use Cases', Addison-Wesley.
4. Dardenne, A., Fickas, S., Lamsweerde, A. (1993) 'Goal-Directed Requirements Acquisition', *Science of Computer Programming.*, 20(1-2), pp. 3-50. doi:10.1016/0167-6423(93)90021-G.
5. Ireland, A. (2021), 'Success through Failure: A Problem Frames Perspective', Internal *Blue Book Note* series of the Mathematical Reasoning Group, School of Informatics, University of Edinburgh.
6. Jacobson, I., Spence, I., Bittner, K. (2011) 'Use Case 2.0 – The Guide to Succeeding with Use Cases', (Accessed 12/10/2023), available at: https://www.ivarjacobson.com/publications/white-papers/use-case-20-e-book
7. McQuaid, P.A. (2012) 'Software Disasters – Understanding the Past, to Improve the Future', *Journal of Software: Evolution and Process*, 24(5), pp. 459-470, doi: 10.1002/smr.500.
8. Mavin, A., Harwood, A. R. G., Wilkinson, P. (2009) *Easy Approach to Requirements Syntax (EARS)*, Requirements Engineering Conference, 2009, pp. 317-322, doi: 10.1109/RE.2009.9
9. OMG (2017), *UML Specification Version 2.5.1*, (Accessed 19/11/2023), available at: https://www.omg.org/spec/UML/2.5.1/PDF
10. Respect-IT (2007), *A KAOS Tutorial* (Accessed 23/09/2023), available at: https://objectiver.com/fileadmin/download/documents/KaosTutorial.pdf
11. Respect-IT (2008), *Objectiver – General Overview* (Accessed 10/11/2023), available at: https://objectiver.com/fileadmin/download/documents/generalleaflet.pdf

12. Respect-IT (2008), *The Objectiver Tool* (Accessed 10/11/2023), available at:

    https://objectiver.com/fileadmin/download/documents/presentations/KaosCEE-Grail.pdf

13. Respect-IT, *Using Objectiver in Combination with UML*, (Accessed 10/11/2023), available at:

    https://objectiver.com/index.php?id=20

14. Serlio Software, *CaseComplete* (Accessed 15/10/2023), available at:

    https://casecomplete.com/

15. Sparx Systems, *Enterprise Architect* (Accessed 11/11/2023), available at:

    https://sparxsystems.com/

16. UMLet GitHub repository (Accessed 27/09/2023), available at:

    https://github.com/umlet/umlet

    Andreas (2023), *Information for Developers* (Accessed 27/09/2023), available at:

    https://github.com/umlet/umlet/wiki/Information-for-Developers

11 Appendix: PLES - Professional, Legal, Ethical and Social Issues

Professional: The project will be carried out in accordance with the British Computer Society's code of conduct. All sources will be referenced appropriately. The creation of the DUCK methodology would propose a new standard in requirements engineering. If adopted it could potentially benefit the field as it may improve the quality of systems being produced. However, using it can also leave some inconsistencies uncaught while giving false confidence, which would be a negative impact.

Legal: UMLet is available under GNU General Public Licence meaning that it can be freely modified to make the assistant application and distributed if it is marked as changed from the original version and the source code is also made freely available, giving others the same set of permissions. Additionally, GNU GPL gives a Disclaimer of Warranty and a Limitation of Liability, meaning those who use the application are doing so at their own risk. When published under GNU GPL, DUCK's code will be made open source on a public GitHub repository in the same fashion as UMLet's. While some general inspiration might be taken from proprietary software discussed in section 2.2, this will be within the bounds of copyright law and there will be no attempt to access its source code.

Ethical: An approval from the ethics committee will be received before conducting the usability study. No personal information is gathered, all the questionnaire responses will be fully anonymous. It will also be explained to the participants that only their feedback on the application's GUI is measured rather than their ability to use it, so they would not worry about their performance.

Social: Making the DUCK assistant application open source would have a positive societal impact as it would provide a free means to learning requirements engineering and use case modelling. It can also be used by small organisations, emerging entrepreneurs, and charities who may not be able to afford proprietary software.

12 Appendix: Project Management

12.1 Gantt chart

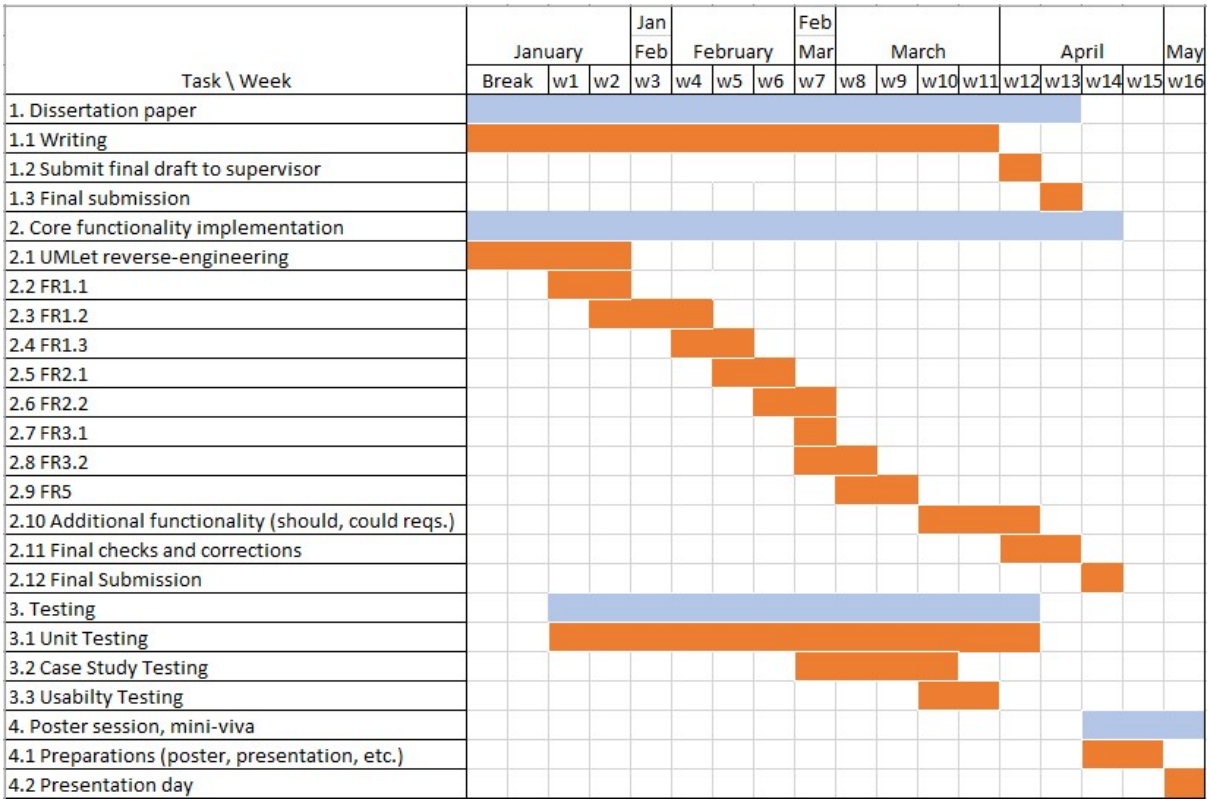


Figure 378. Project Gantt chart

12.2 Risk analysis

ID	Risk	Likelihood	Impact	Prevention and/or Mitigation
R-1	Loss of data	Low	High	**Prevention:** Make daily code pushes to GitHub and report backups to OneDrive. **Mitigation:** Restore yesterday's version.
R-2	Difficulties using UMLet, Eclipse or Java for the project	Low	Medium	**Prevention:** UMLet source code already imported into Eclipse and modification attempts have been successful, thus issues unlikely. **Mitigation:** Find alternatives.
R-3	Difficulties building part of the implementation	Medium	Medium	**Mitigation:** See if a simplified version can be implemented or if it can be abstracted instead.

R-4	Difficulties completing the project on time	Medium	Medium	**Prevention:** Trying to finish each withing outlined timeframes and starting on the next one earlier. **Mitigation:** Simplifying or cutting out functionality where possible.
R-5	Little to no year 2 students from F28SD course agree to take part in the study	High	Low	**Mitigation:** Recruit participants from fellow year 4 students instead, offering to participate in their studies in exchange.
R-7	Difficulties finishing the implementation before the usability testing is planned to start.	Medium	Medium	**Prevention:** Following the outlined project plan. **Mitigation:** Only carry out user testing for finished features.
R-8	Supervisor unable to oversee the project	Low	High	**Mitigation:** Applying for Mitigating Circumstances and reaching out to course leader to see if another supervisor can is willing to oversee the project.

Table 12. Risk analysis

13 Appendix: The SSSS Running Example

ID	Description	Priority
FR1	The SSSS shall grant entry to a staff member when they provide a valid PIN.	M
FR2	The SSSS shall grant entry to a client when: i. The provide a valid PIN AND ii. The client with this PIN is not currently in the building AND iii. Current time is between 08:00 and 18:00	M
FR3	The SSSS shall record the person's PIN, name, and time of entry.	M
FR4	The SSSS shall trigger the alarm and send a message to the SMS if anyone attempts to enter using a PIN belonging to a person who is recorded as currently present in the building.	M
…	…	…

Table 13. Part of the requirements table for the SSSS

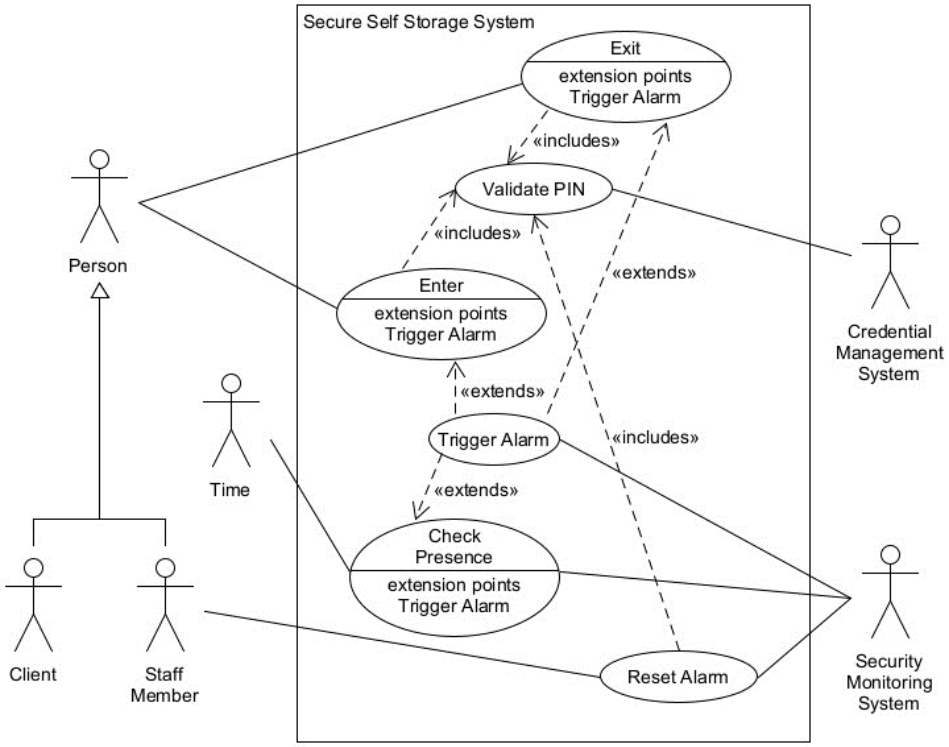


Figure 389. A use case diagram for the SSSS

Use Case: person*Enter*
ID: ***1***
Goal: ***A person to enter the building***
Primary actor: ***person***
Secondary actor(s): ***none***
Preconditions: 1. ***The external keypad, SSSS and CMS function correctly*** 2. ***The connection between the external keypad, SSSS and CMS functions correctly*** 3. ***The door is closed and locked***
Postconditions: 1. ***The person has been granted access to the building*** 2. ***The person's ID, name and the time of entry have been recorded***
Main flow: 1. ***The person enters their PIN on the external keypad.*** 2. ***Include(validatePin)*** 3. ***The SSSS checks its records to see if the person with this PIN is currently present in the building.*** 4. ***The SSSS checks that the person is authorised to enter the building at current time.*** 5. ***The SSSS displays an "Access Granted" message on the keypad and unlocks the door.*** 6. ***The SSSS adds the person's PIN, name, and time of entry to its records.*** 7. ***The person enters the building.*** ***Extension point: triggerAlarm***
Alternative flows: ***4a. The person's type is "client", and current time is outside the 8:00 –18:00 period authorised for clients.*** 1. ***The SSSS displays a message on the keypad saying the person is not authorised to enter the building at current time.*** 2. ***Use case terminated.***

Table 20. A use case scenario

14 Appendix: Existing Software Screenshots

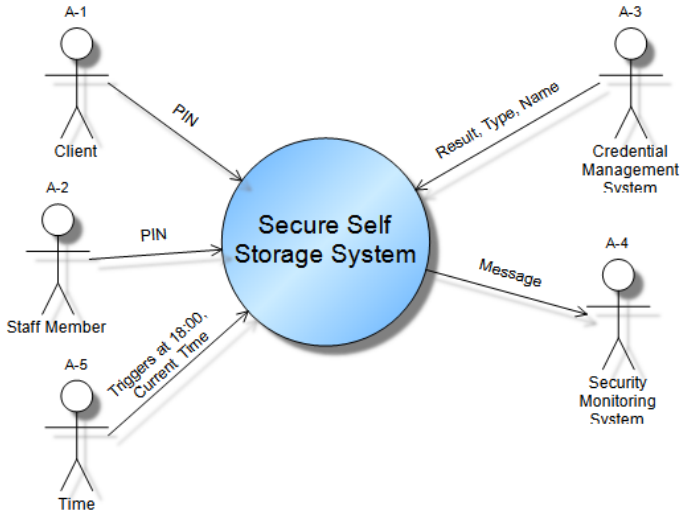


Figure 4039. A context diagram in CaseComplete

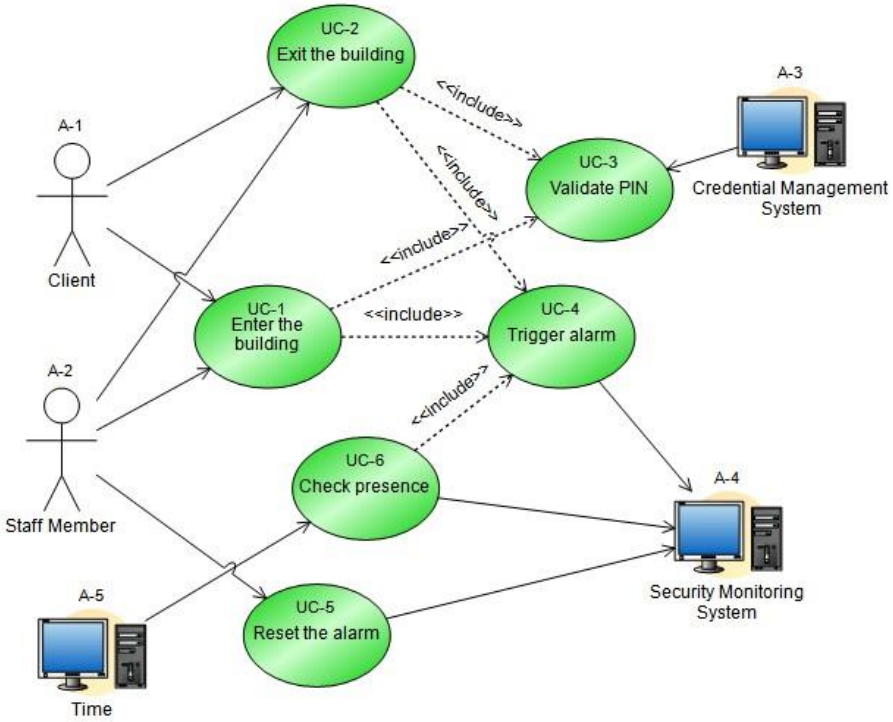


Figure 40. A Use Case Diagram in CaseComplete

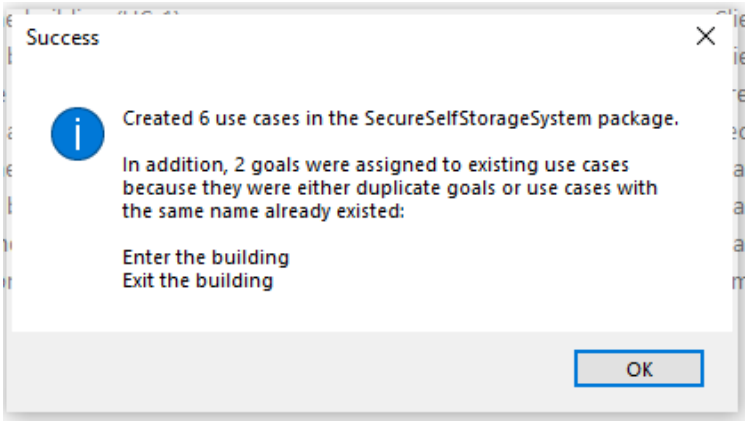


Figure 41. Same goals combined in CaseComplete

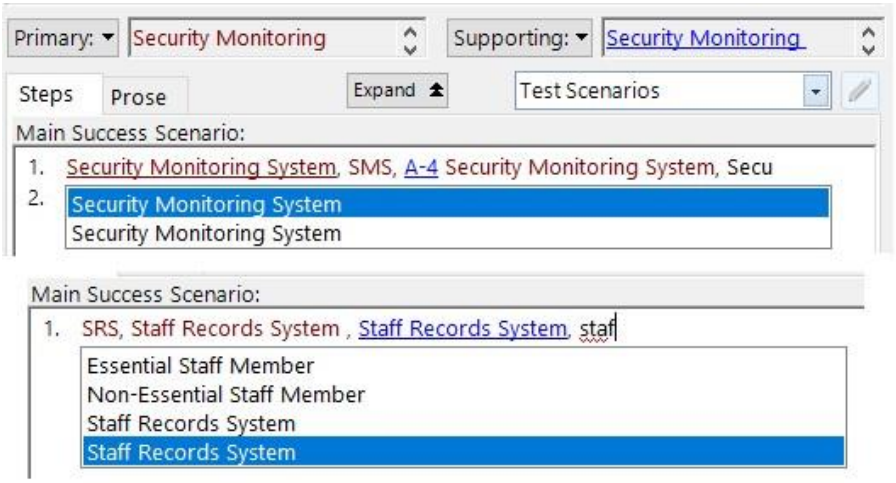


Figure 42. Duplicated actor and definition in CaseComplete

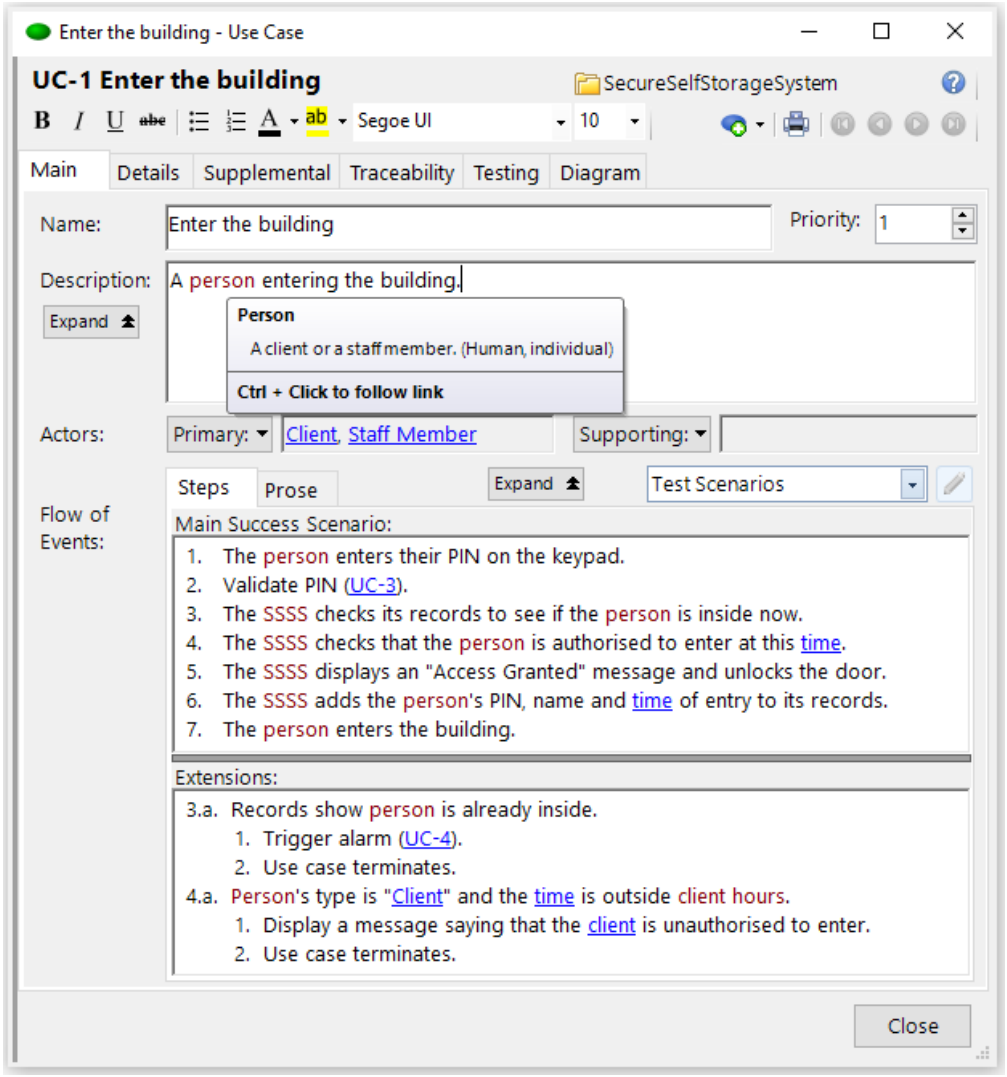


Figure 43. A use case scenario in CaseComplete

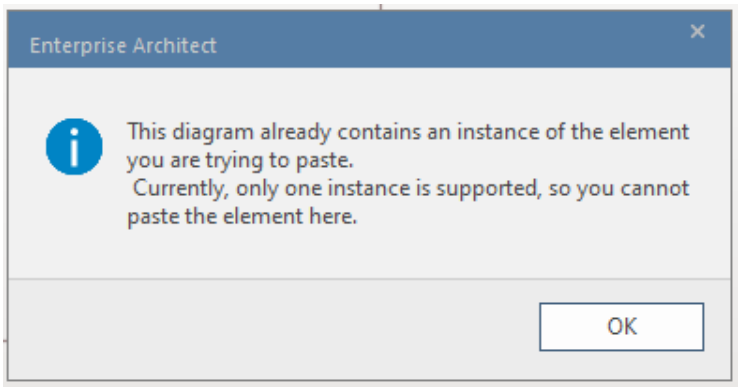


Figure 44. Use case duplication error in Enterprise Architect

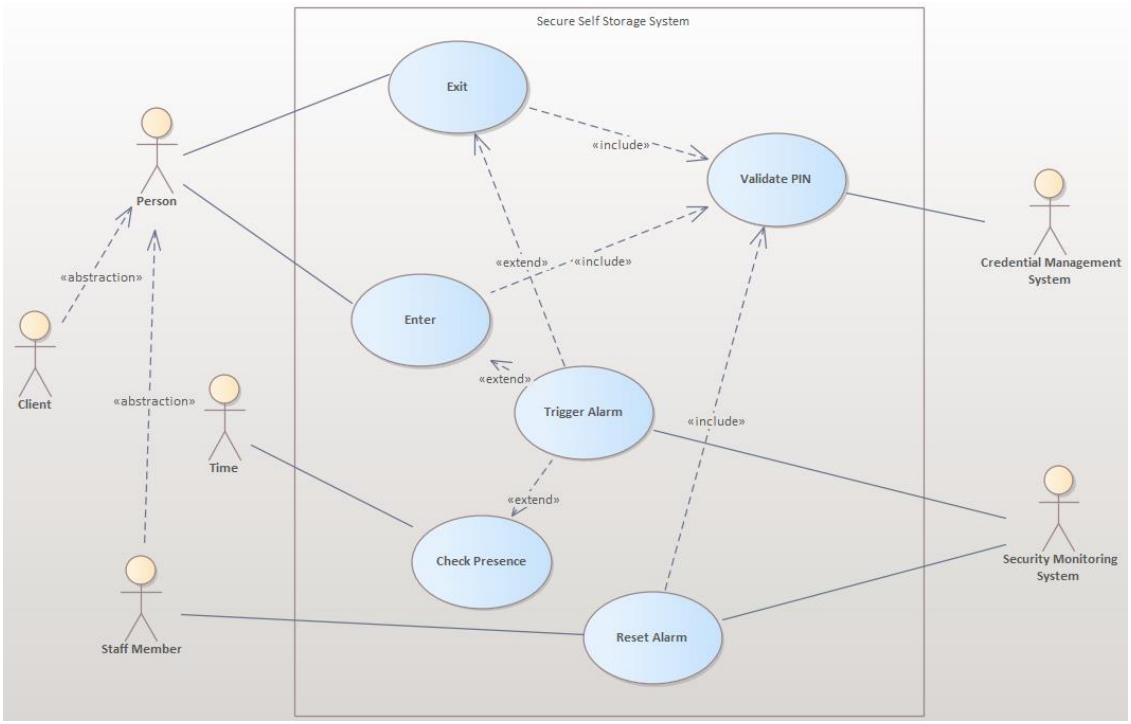


Figure 45. A use case diagram in Enterprise Architect

Step		Action	Uses	Results	State
	1	The person enters their PIN on the keypad.		The SSSS has the entered PIN.	
	2	<include> Validate PIN			
	3	The SSSS checks its records to see if the person is already inside the building .	FR4		
	4	The SSSS checks that the person is authorised to enter at this time.			
	5	The SSSS displays an "access granted" message and opens the door.		Door is open.	Access granted
	6	The SSSS adds the person's PIN, name and time of entry to its records.	FR3		
	7	The person enters the building.	FR1, FR2	The person is inside the building.	

Entry Points | Context References | Constraints

Step	Path Name	Type	Join
0	Basic Path	Basic Path	-
3a	Person with this PIN is already recorded as present	Alternate	End
4a	Person's type is "client" and current time is outside the 8:00 –18:00 period authorised for clients.	Alternate	End

Figure 46. A use case scenario in Enterprise Architect

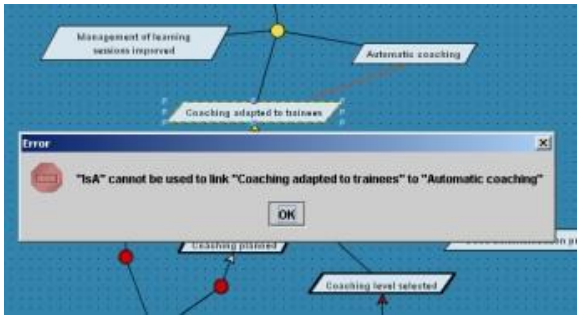


Figure 47. KAOS syntax conformance error in Objectiver (Respect-IT, 2008 [11])

15 Appendix: UMLet Modification Experiments

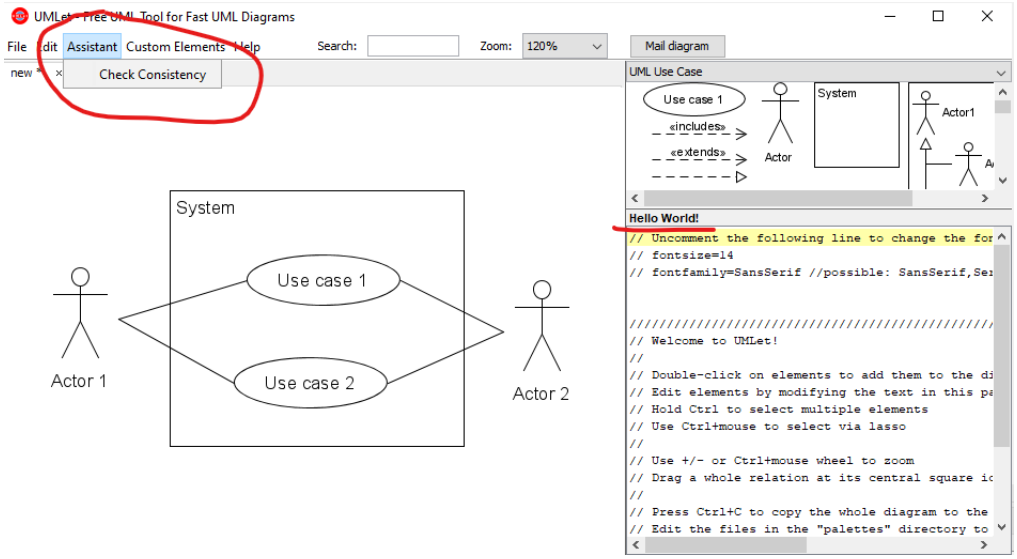


Figure 48. UMLet with a successfully modified GUI

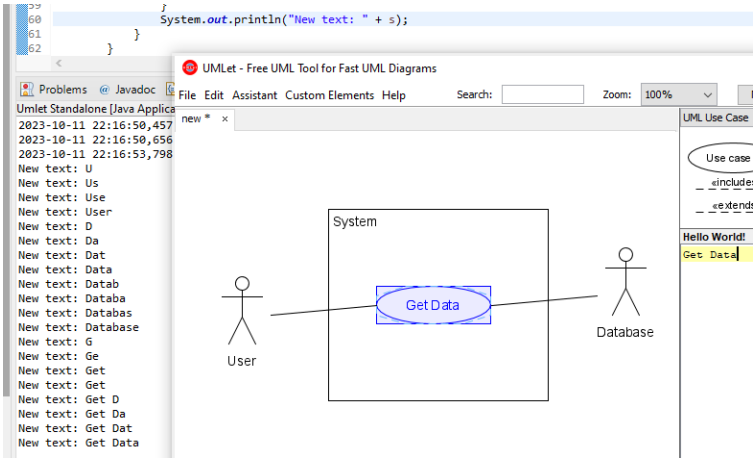


Figure 50. Changing UMLet diagram element's label linked to console output

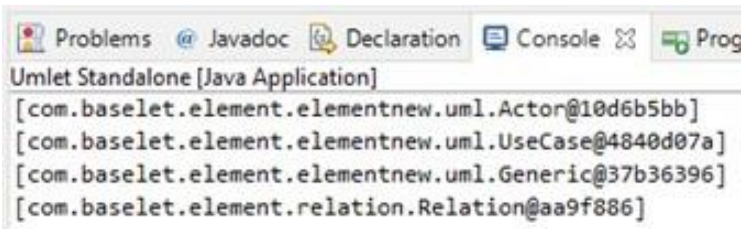


Figure 51. Early experiment printing element strings

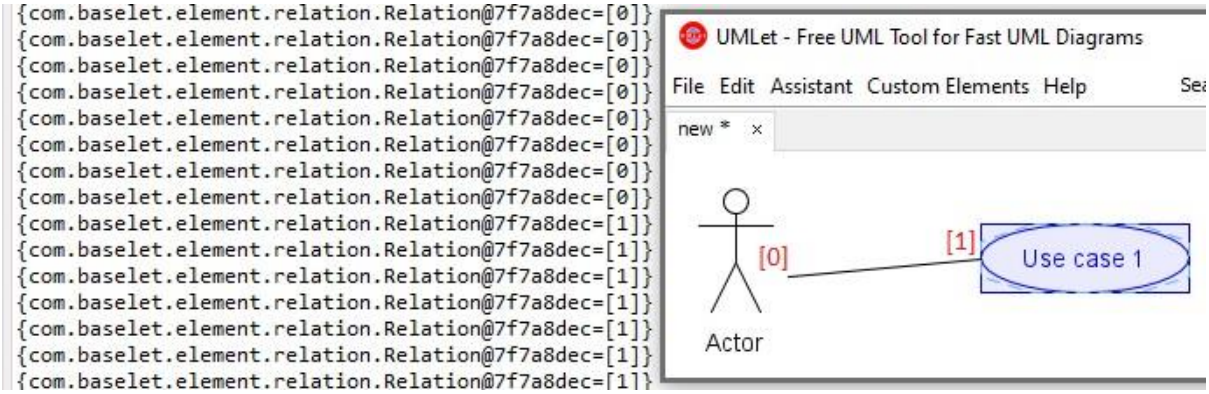
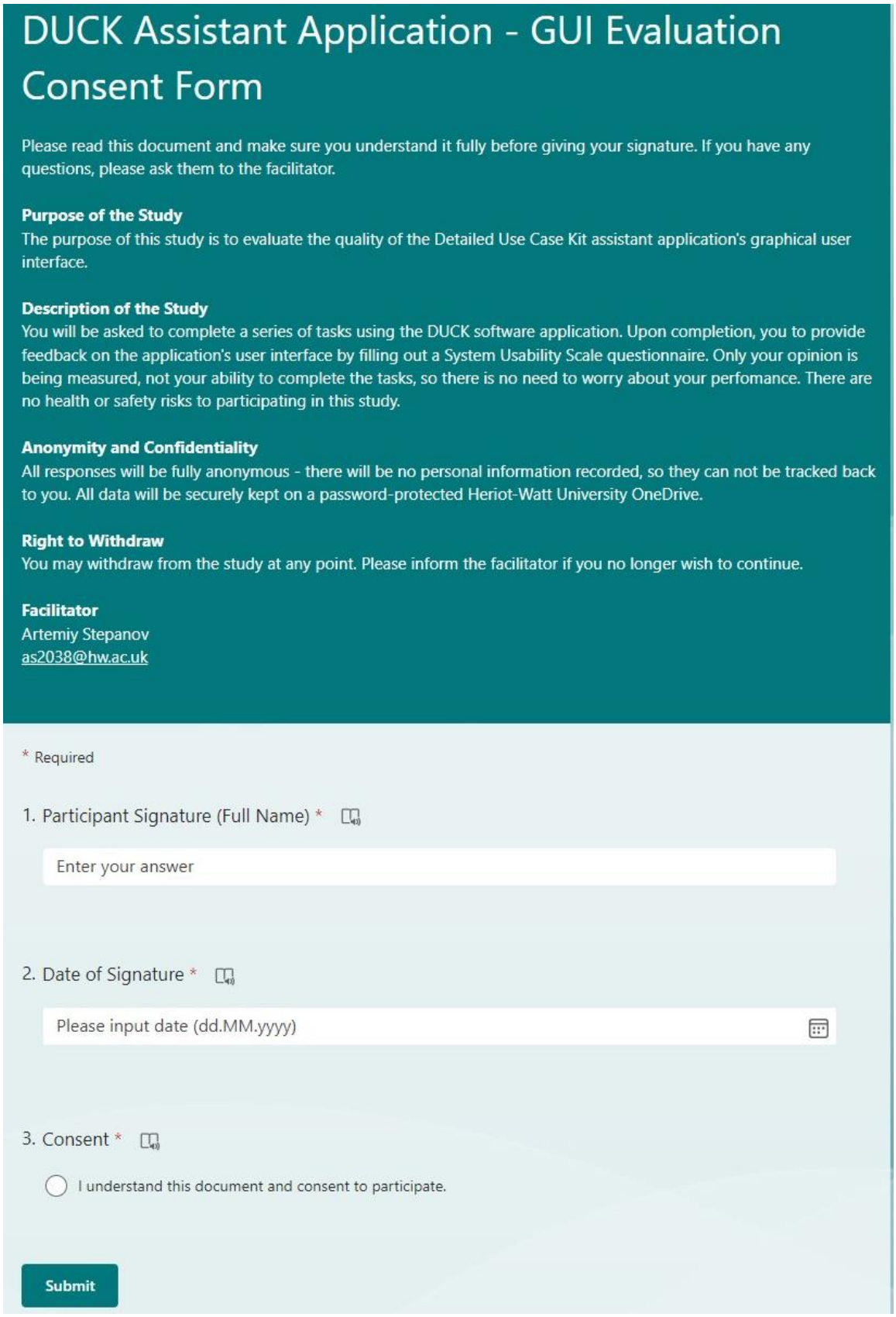


Figure 492. Experiment printing the StickableMap of moved point indexes

16 Appendix: Initial Sample Consent Form and Questionnaire



DUCK Assistant Application - GUI Evaluation Consent Form

Please read this document and make sure you understand it fully before giving your signature. If you have any questions, please ask them to the facilitator.

Purpose of the Study
The purpose of this study is to evaluate the quality of the Detailed Use Case Kit assistant application's graphical user interface.

Description of the Study
You will be asked to complete a series of tasks using the DUCK software application. Upon completion, you to provide feedback on the application's user interface by filling out a System Usability Scale questionnaire. Only your opinion is being measured, not your ability to complete the tasks, so there is no need to worry about your perfomance. There are no health or safety risks to participating in this study.

Anonymity and Confidentiality
All responses will be fully anonymous - there will be no personal information recorded, so they can not be tracked back to you. All data will be securely kept on a password-protected Heriot-Watt University OneDrive.

Right to Withdraw
You may withdraw from the study at any point. Please inform the facilitator if you no longer wish to continue.

Facilitator
Artemiy Stepanov
as2038@hw.ac.uk

* Required

1. Participant Signature (Full Name) *

Enter your answer

2. Date of Signature *

Please input date (dd.MM.yyyy)

3. Consent *

I understand this document and consent to participate.

Submit

Figure 50. Initial consent form

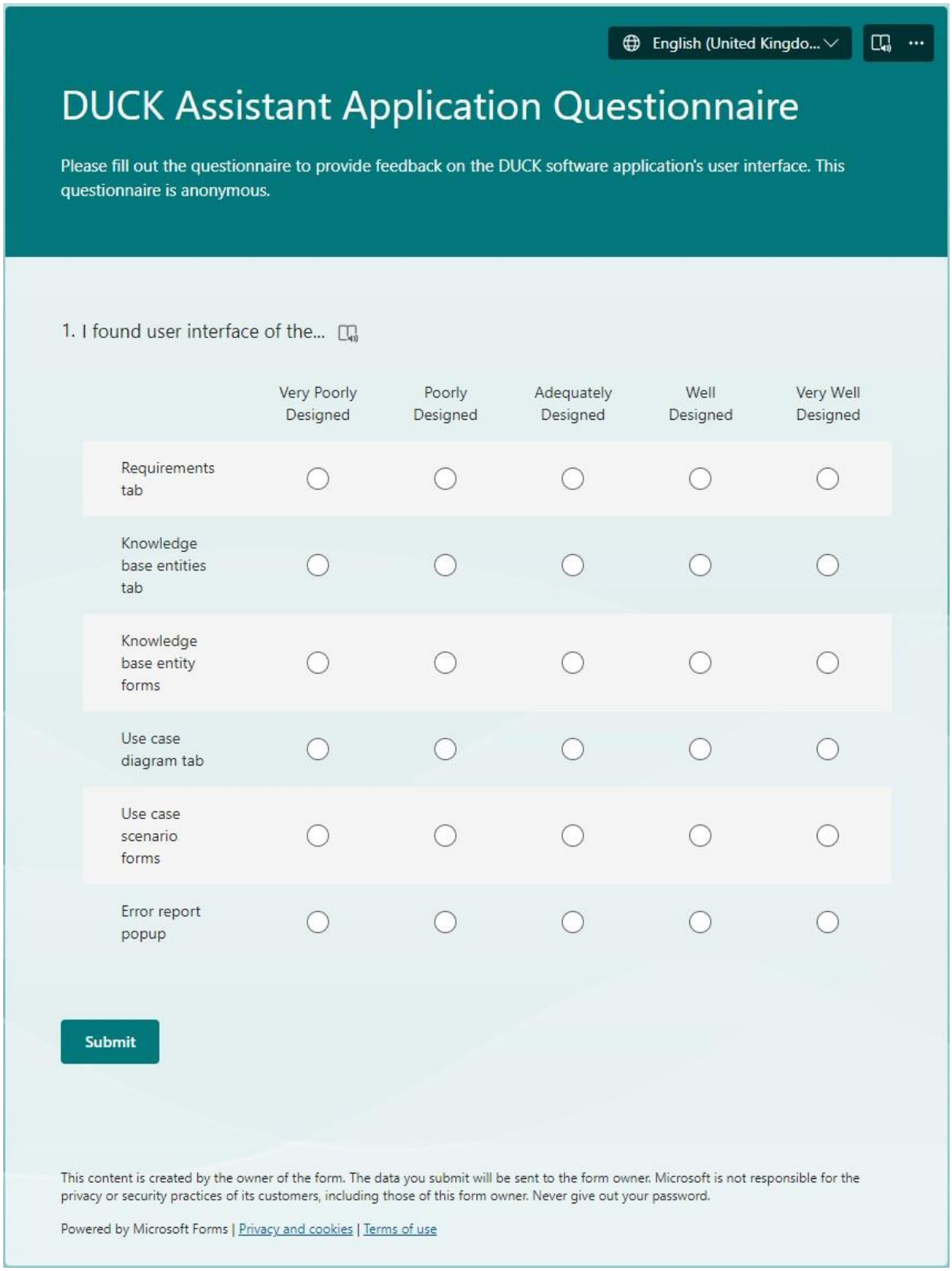


Figure 51. Initial exit questionnaire

17 Appendix: Additional Architecture Diagrams

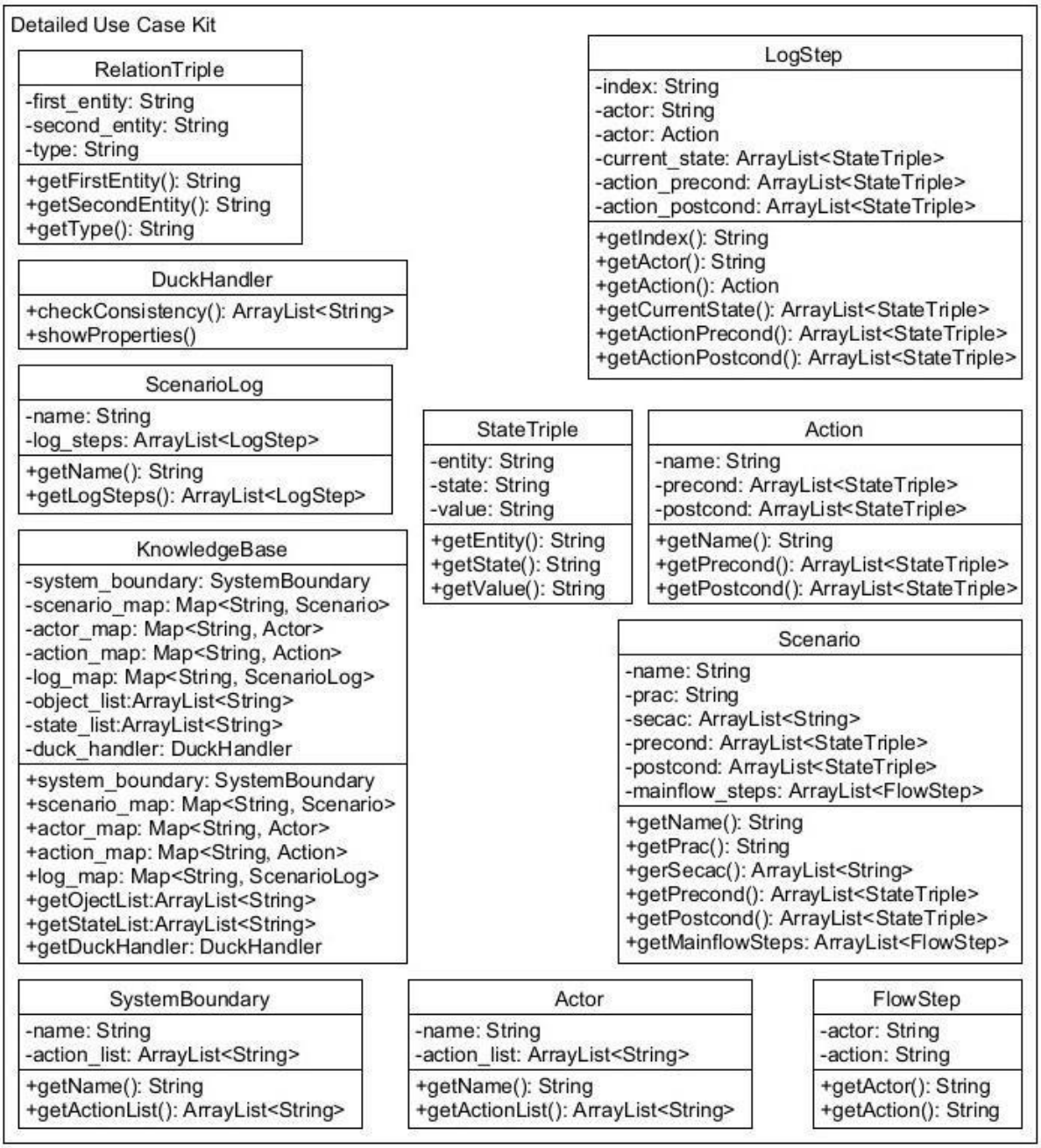


Figure 52. Class Diagram for the Detailed Use Case Kit

18 Appendix: Using Launch4j

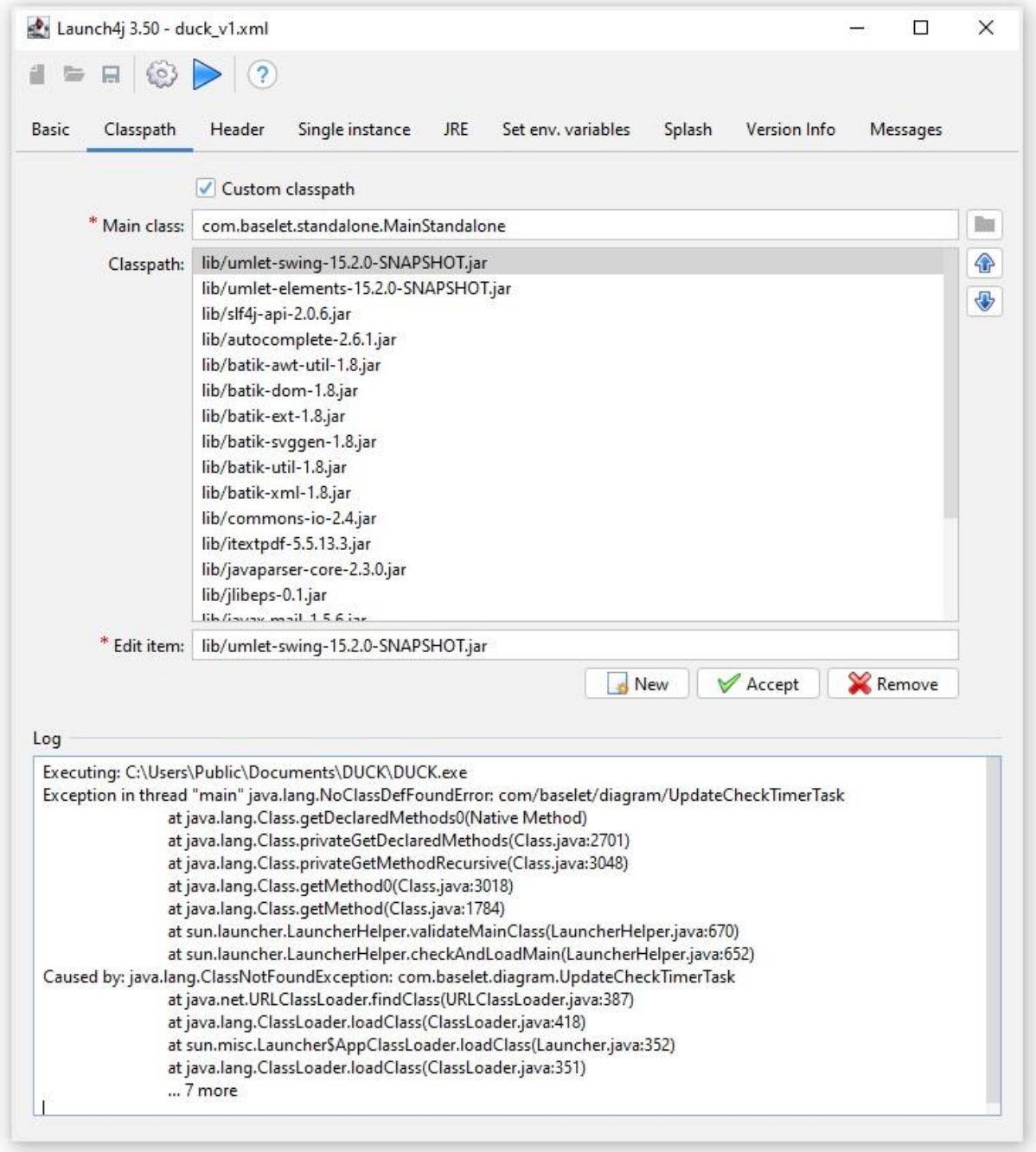
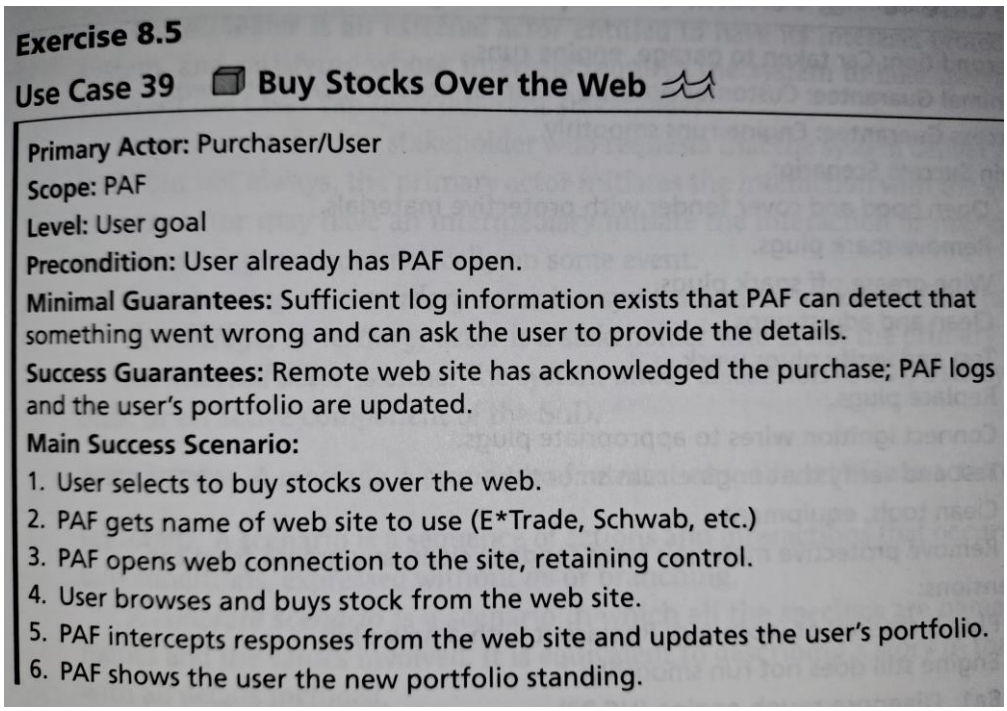


Figure 53. Failed attempt to build a DUCK executable using Launch4j

19 Appendix: Case Study Details



Exercise 8.5

Use Case 39 **Buy Stocks Over the Web**

Primary Actor: Purchaser/User

Scope: PAF

Level: User goal

Precondition: User already has PAF open.

Minimal Guarantees: Sufficient log information exists that PAF can detect that something went wrong and can ask the user to provide the details.

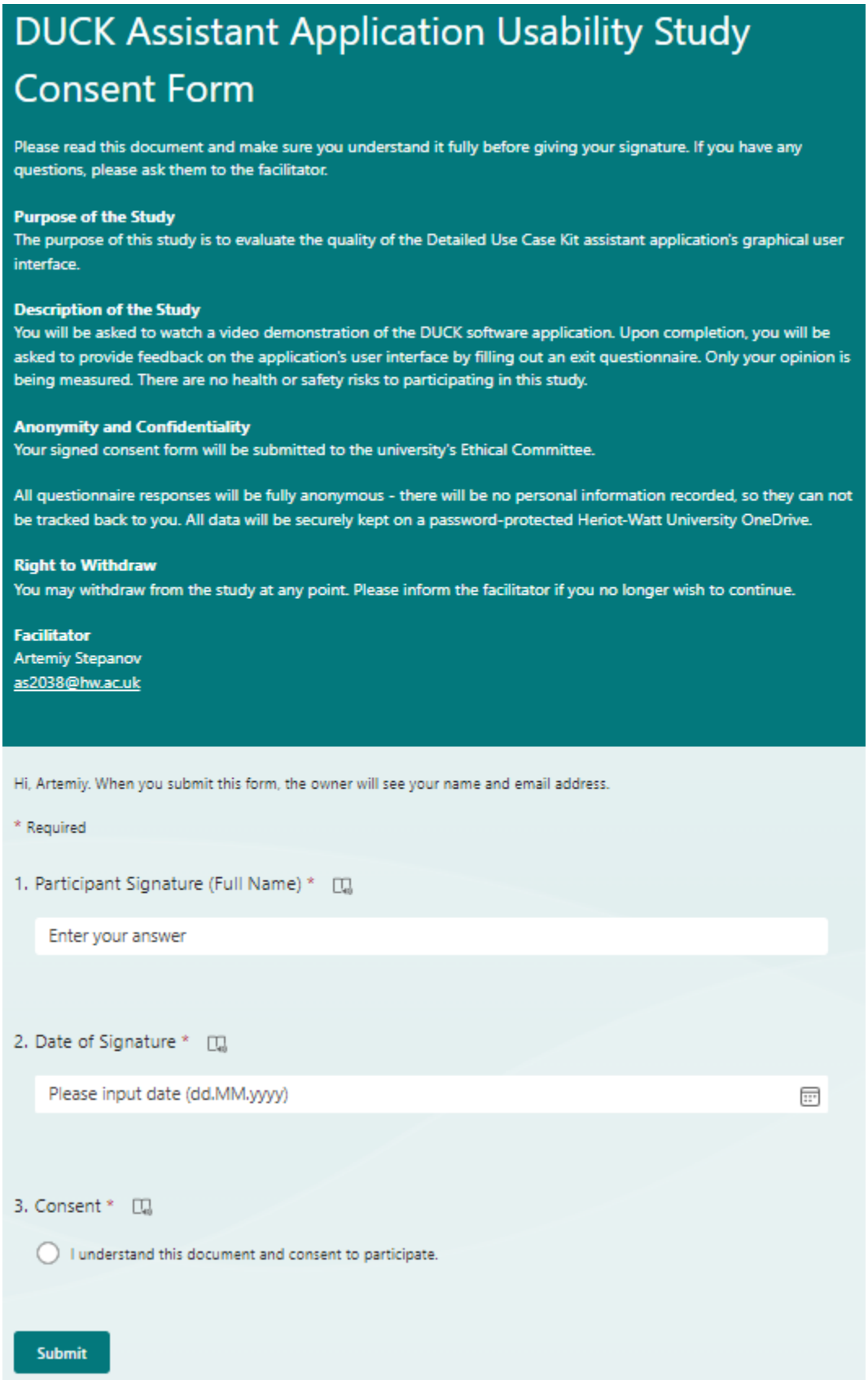
Success Guarantees: Remote web site has acknowledged the purchase; PAF logs and the user's portfolio are updated.

Main Success Scenario:

1. User selects to buy stocks over the web.
2. PAF gets name of web site to use (E*Trade, Schwab, etc.)
3. PAF opens web connection to the site, retaining control.
4. User browses and buys stock from the web site.
5. PAF intercepts responses from the web site and updates the user's portfolio.
6. PAF shows the user the new portfolio standing.

Figure 54. Use Case 39 "Buy Stocks Over the Web", "Designing Effective use cases", page 251 (Cockburn, 2001)

20 Appendix: Updated Sample Consent Form and Questionnaire



DUCK Assistant Application Usability Study Consent Form

Please read this document and make sure you understand it fully before giving your signature. If you have any questions, please ask them to the facilitator.

Purpose of the Study
The purpose of this study is to evaluate the quality of the Detailed Use Case Kit assistant application's graphical user interface.

Description of the Study
You will be asked to watch a video demonstration of the DUCK software application. Upon completion, you will be asked to provide feedback on the application's user interface by filling out an exit questionnaire. Only your opinion is being measured. There are no health or safety risks to participating in this study.

Anonymity and Confidentiality
Your signed consent form will be submitted to the university's Ethical Committee.

All questionnaire responses will be fully anonymous - there will be no personal information recorded, so they can not be tracked back to you. All data will be securely kept on a password-protected Heriot-Watt University OneDrive.

Right to Withdraw
You may withdraw from the study at any point. Please inform the facilitator if you no longer wish to continue.

Facilitator
Artemiy Stepanov
as2038@hw.ac.uk

Hi, Artemiy. When you submit this form, the owner will see your name and email address.

* Required

1. Participant Signature (Full Name) *

Enter your answer

2. Date of Signature *

Please input date (dd.MM.yyyy)

3. Consent *

I understand this document and consent to participate.

Submit

Figure 58. Updated consent form

DUCK Assistant Application Questionnaire

Please fill out the questionnaire to provide feedback on the DUCK software application's user interface. This questionnaire is anonymous.

* Required

1. I found the user interface of the ... *

	Very Poorly Designed	Poorly Designed	Adequately Designed	Well Designed	Very Well Designed
Knowledge base panel	○	○	○	○	○
Knowledge base entity forms	○	○	○	○	○
Use case scenario forms	○	○	○	○	○
Validation panel	○	○	○	○	○
Scenario log window	○	○	○	○	○

2. I think that most users would find ... *

	Very Difficult	Difficult	Easy	Very Easy
Adding entities to the knowledge base	○	○	○	○
Accessing an existing entity's details	○	○	○	○
Editing a use case scenario form	○	○	○	○
Understanding the warning report	○	○	○	○
Understanding a scenario flow	○	○	○	○

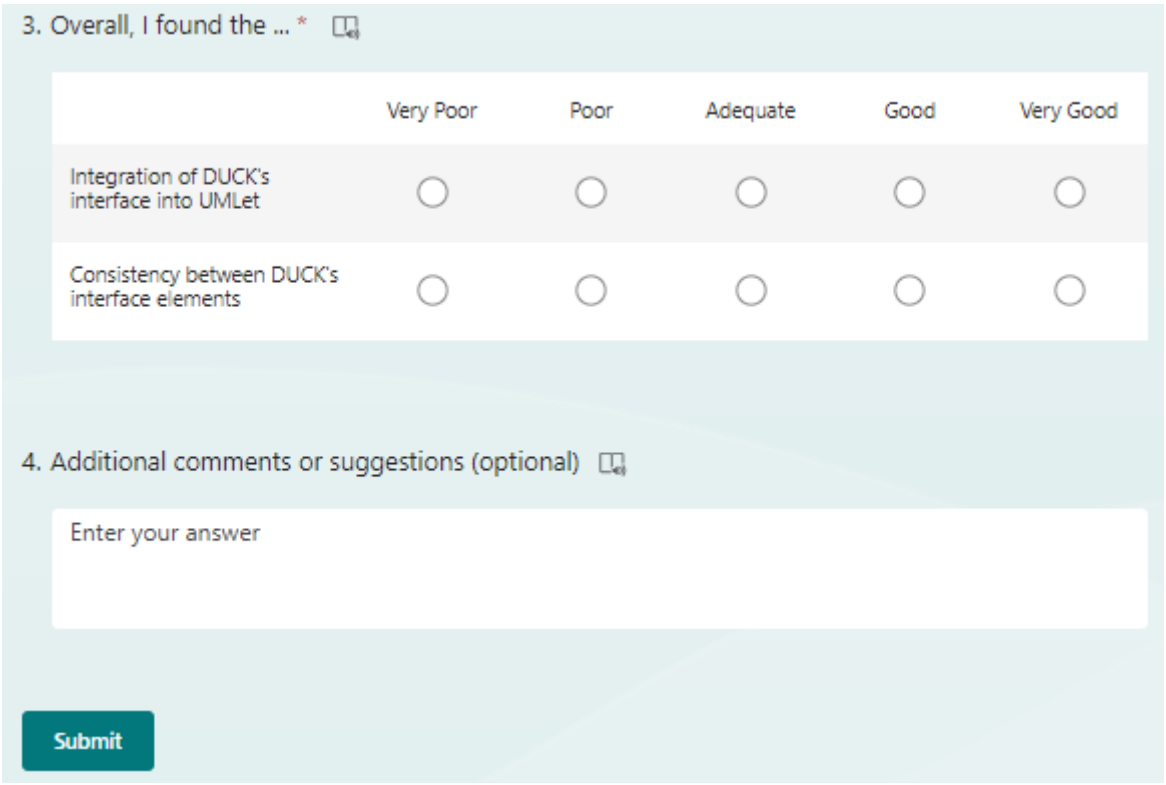


Figure 55. Updated exit questionnaire

21 Appendix: Video Links

The videos are available to watch on MS SharePoint (requires a HWU account).

5-minute demo: Artemiy-Stepanov-Year-4-Project-4-Minute-Video.webm

Usability testing video (12 minutes): DUCK-Usability-Testing-Video.webm