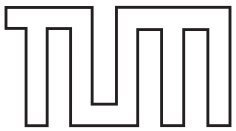


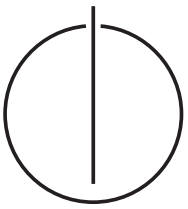
Technische Universität München

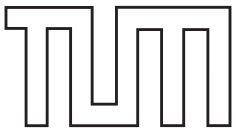
Fakultät für Informatik

Bachelorarbeit in Informatik

A framework for remote usability evaluation on mobile devices

Daniel Bader





Technische Universität München

Fakultät für Informatik

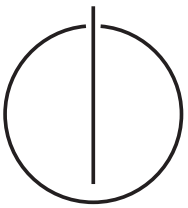
Bachelorarbeit in Informatik

A framework for remote usability evaluation on mobile devices

Ein Framework für Remote Usability Evaluation auf mobilen Geräten

Author:	Daniel Bader
Supervisor:	Prof. Bernd Brügge, PhD.
Advisors:	Dipl.-Inf. Univ. Dennis Pagano
	Dipl.-Inf. Univ. Damir Ismailović

Submission Date: February 28, 2011



I assure the single handed composition of this bachelor's thesis only supported by declared resources.

Date, Signature: .....
(Daniel Bader)

Abstract

On mobile platforms, such as Google Android and Apple iOS, few software frameworks are available that support remote usability evaluation methods. Nevertheless, remote usability evaluation methods are an interesting area for research because they can be partly automated, thereby making them very time- and cost-efficient. In this thesis we propose a software, the muEvaluationFramework, that allows remote usability evaluation on mobile platforms and supports automation in the capture, analysis and critique phases of an usability evaluation. To demonstrate the abilities of the framework, we provide an application-independent prototypical implementation for the Apple iOS platform.

Kurzfassung

Auf mobilen Plattformen, wie zum Beispiel Google Android und Apple iOS, sind kaum Softwareframeworks verfügbar welche die Durchführung von Remote Usability Evaluationen unterstützen. Methoden der Remote Usability Evaluation sind ein interessantes Forschungsfeld da es möglich ist, sie teilweise zu automatisieren. Dadurch werden diese Methoden sehr zeit- und kosteneffizient. In dieser Bachelorarbeit beschreiben wir eine Softwarelösung, das muEvaluationFramework, welches die Durchführung von Remote Usability Evaluationen auf mobilen Geräten unterstützt und eine Automatisierung in der Aufzeichnungs-, der Analyse- und der Kritikphase einer Usability Evaluation ermöglicht. Um die Fähigkeiten des Frameworks zu demonstrieren, stellen wir eine applikationsunabhängige prototypische Implementierung für Apple iOS zu Verfügung.

Acknowledgment

I am grateful to my supervisors Dennis Pagano and Damir Ismailović for their strong support throughout the writing of my thesis.

Contents

List of Figures

1 Introduction

On mobile devices, such as smartphones or tablet computers, good software usability is especially important. This is because users may interact with their devices under difficult and distracting conditions, for example in noisy environments or at low light levels. Although good usability is vital on these platforms it is still difficult to sample and collect data about user interactions. This also affects popular platforms like the Apple iPhone. We believe that is mainly the case because tool support for collecting and analyzing usability data is not very good or even nonexistent on most mobile platforms.

However, there is room to improve this situation by taking technologies and ideas from another area of software development, namely performance analysis, and applying them to the context of usability testing. Tools like profilers are readily available to the modern mobile application developer and allow for structured examination and analysis of application performance using software-based sensors. With similar techniques interaction patterns can be captured, and in a further step, be examined using heuristic algorithms that find and point out usability problems. This is called *automated* usability evaluation. If we also remove the need for human evaluation experts to be present during an evaluation session then the technique becomes automated *remote* usability evaluation.

This thesis describes a framework for automated remote usability evaluation on mobile devices. We provide a sample implementation for the Apple iOS platform and use it with the two open-source applications *Wordpress for iOS* and *PlainNote*.

1.1 Problem statement

Software usability is important for every application on every platform and usually tools and methodologies exist to help developers improve the usability of their software. However, there is a lack of software support for usability evaluation on mobile platforms [9,23]. Thus, usability evaluations on these platforms are often problematic and involve complicated laboratory setups to film screen contents and user interactions [15].

The users are also removed from their natural environment during a laboratory evaluation and thus do not experience many of the distractions (such as bad lighting, movement or noise) that are common in a mobile context. This makes evaluation results less valid in real life usage scenarios [22].

Several approaches have been taken to mitigate these problems. For example, remote usability evaluation for mobile devices allows to perform usability evaluation outside laboratories [27]. But these evaluation methods are not available for all mobile device platforms and the methods focus on older or by now deprecated platforms.

Especially on the rather novel class of touch-enabled devices, like the Apple iPhone, there still is no proper software support for conducting remote usability evaluations. Therefore, we propose a framework for remote usability evaluation on mobile platforms called `muEvaluationFramework` ("mobile usability Evaluation Framework"). The framework supports logging and analyzing usability data in both traditional laboratory-based evaluations and field tests.

1.2 Outline

Chapter 2 provides an overview of the application domain of the framework and introduces three existing technologies for remote usability evaluation. To document the requirements elicitation phase, the chapter describes the high-level functional and non-functional requirements. In addition, the chapter also describes the analysis model in the form of use cases, object models and dynamic models for the framework. It contains the complete functional specification and therefore it can be seen as the *Requirements Analysis Document* (RAD) for the framework.

Chapter 3 supplies an overview of framework's software architecture and documents the system design model with the subsystem decomposition, hardware/software mapping and information about persistent data management. This chapter represents the *System Design Document* (SDD).

Chapter 4 is organized by the packages documented in Chapter 3 and is considered to be the *Object Design Document* (ODD).

Chapter 5 describes the prototypical implementation of the framework that was developed during the writing of the thesis.

Chapter 6 gives several ideas for future work on the framework.

Chapter 7 shows conclusions drawn from the development of the framework.

1.3 Document conventions

The following conventions are used throughout the document:

Typographical conventions

- The names of system and modeling elements such as classes, packages, attributes, and methods are written in `typewriter font`.
- Important parts of the text are highlighted by using *italics*.

Notation and diagrams

- We use the Unified Modeling Language (UML) version 2.1.2 for all diagrams of the thesis [26].

1.4 Foundations

This section introduces the terms and concepts that are required to understand the application domain of the `muEvaluationFramework`. First, we define the term *usability* and explain how it is used in the context of mobile applications. Second, we briefly describe six state-of-the-art approaches towards *usability evaluation*. Third, we provide an overview of *common activities* in usability evaluations that we use as an overall structure for the thesis. Fourth, we introduce *remote usability evaluation* and two ways to perform it. Finally, we summarize the advantages and disadvantages of *automated usability testing*.

1.4.1 Usability of mobile applications

While the importance of usability is now widely recognized, there remains a confusion regarding the exact meaning of the term [11]. Generally, *usability* is an indicator for the ease of use and the acceptability of a system [20]. Researchers frequently define usability as the sum of five common attributes [8, 24, 32] (Figure 1.1):

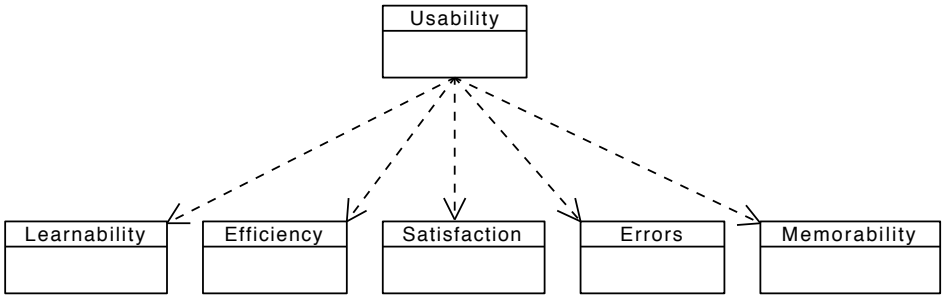


Figure 1.1: Five common attributes of usability (UML class diagram)

- *Learnability*: how easily a first time user without prior knowledge operates an interface;
- *Efficiency*: the performance of experienced users who already have knowledge about the interface;
- *Memorability*: how easily previous users of an application can recall how to operate it after some time has passed;
- *Errors*: the number of errors a user makes and how easily they can recover from an error; and
- *Satisfaction*: how pleasant it is for users to operate an interface.

So far, the term usability is defined, but how is *mobile usability* different from it? Zhang and Adipat identified the following contributing factors that affect mobile usability [32]:

- *Mobile context*: the fact that users are distracted by the surrounding environment, e.g. by noise, movement, or light level;
- *Connectivity*: many mobile devices do not have an internet connection all the time and most have considerably slower network speeds than stationary devices;
- *Small screen size*: small screens are harder to read and require different aesthetic decisions;
- *Different display resolutions*: a number of differing screen sizes and display resolutions exist in the mobile market;
- *Limited processing power*: compared to stationary devices mobile devices are a lot slower (this may prevent, for example, the implementation of graphically demanding features); and
- *Restrictive data entry methods*: small or virtual (on-screen) keyboards make it difficult to quickly enter data and may increase the error rate.

1.4.2 Usability evaluation

Usability evaluation helps to determine and improve the usability of an application by taking a structured and scientific approach towards the problem. Common usability evaluation techniques can be divided into two groups: *inspection methods* and *test methods* [20].

Inspection methods do not require the presence of a user when conducting the evaluation. Instead, they are "a set of methods for identifying usability problems and improving the usability of an interface design by checking it against established standards" [20].

As described by Holzinger [20], the three most common inspection methods are:

- *Heuristic evaluation*: a usability expert judges the user interface according to several established guidelines;
- *Cognitive walkthrough*: a step-by-step simulation of a hypothetical user's behavior when interacting with the interface. This method is performed by a skilled analyst and concentrates on cognitive issues; and
- *Action analysis*: similar to the cognitive walkthrough but focuses less on cognitive issues and analyzes action sequences and statistics such as click counts instead.

Test methods on the other hand are performed in the presence of real users and are fundamental to usability evaluation [28]. Only real users can provide direct information about how people use systems and what their exact problems with a specific interface are [20]. According to Holzinger [20], the three most common usability test methods are:

- *Thinking aloud*: a test users uses "the system for a given set of tasks while being asked to 'think out loud'" [24]. This method also provides an insight into *why* users are performing an action;
- *Field observation*: a simple method where users are watched in their workplace as unobtrusively as possible; and
- *Questionnaires*: collect opinions regarding an interface by directly querying the users. This can also be done in the form of an interview.

1.4.3 Common usability evaluation activities

According to Ivory and Hearst [21], there is a common sequence of activities that are performed in a usability evaluation; they are called *Capture*, *Analysis*, and *Critique* (see Figure 1.2).

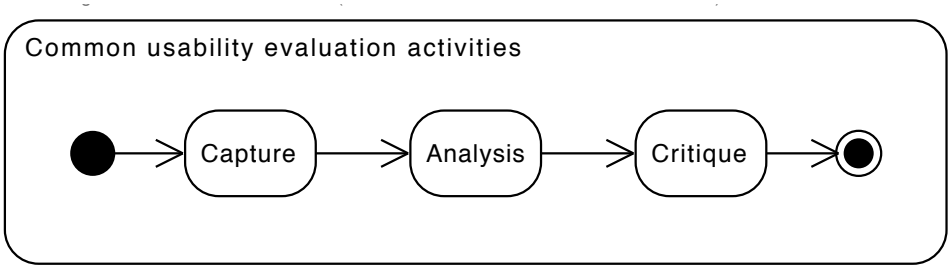


Figure 1.2: Common activities during usability evaluation (UML activity diagram)

Each of the three activities includes a different set of actions:

- During the *Capture* phase usability data are gathered and stored.
- During *Analysis* the data from the previous phase are interpreted and examined for problems in the usability.
- In the final *Critique* phase solutions or improvements to detected usability problems are suggested.

The following exemplary flow of events shows the three activities more concretely. Here, a usability evaluation was performed with the *field observation* test method described in Section 1.4.2:

1. The usability test is performed in an evaluation session during which a test user interacts with an application. Every interaction of the user is recorded by an observer and then stored in a session log or journal. (*Capture*)
2. After the session, an evaluator interprets the raw data from the session journal and generates knowledge about the application's usability. This knowledge may include statistical data, helpful visualizations of the captured interactions as well as problems detected in the usability. (*Analysis*)
3. The evaluator creates a report that summarizes the knowledge from the analysis phase. The report is used by a developer to improve the application's usability. (*Critique*)

1.4.4 Remote usability evaluation

Traditional usability evaluation methods require evaluators to stay close to the test users when performing an evaluation. But what if similar results could be achieved even when users and evaluators were separated? *Remote usability evaluation* is based on this idea. That is, evaluators and users are separated in space and possibly time during an evaluation [18].

Remote evaluation is becoming more and more popular because it offers several advantages compared to other, laboratory-based evaluation methods. For example, it allows the collection of usability data in real usage situations or locations where users are difficult to assess by direct observation [27]. Furthermore, users behave more naturally when the evaluation is conducted in their familiar environment [27].

Available remote evaluation methods can be grouped into the two categories of *synchronous* and *asynchronous* methods. Andreasen et al. summarize the difference between the two categories as follows:

> "With a synchronous method, the evaluator is separated from the user spatially, but not temporally. When conducting an asynchronous test, the evaluator is separated from the user both temporally and spatially." [1]

1.4.5 Automated usability testing

Automated usability testing tries to remedy some of the disadvantages and problems associated with established usability evaluation methods. For example, traditional evaluation methods are often not systematic and predictable enough [21]. One solution for this problem is to automate some of the common usability activities described in Section 1.4.3 [21]. Au et al. [8] also conclude, that it would be a "logical solution [...] to automate as many aspects of usability testing as possible" because traditional evaluation methods carry several disadvantages, for example high complexity and resource inefficiency.

Automated usability evaluation methods, on the other hand, can be very cost- and time-efficient. Automation leads to several potential benefits, such as decreased costs, increased consistency between results, and a reduced need for evaluation expertise [21]. Because companies are still often reluctant to employ the traditional, cost-intensive usability evaluation methods they have a disadvantage in the market against companies that make use of remote evaluation. Automation can help to boost the acceptance of usability evaluation and leads to better products and better market performances [8].

Based on the work of Balbo [10], Ivory and Hearst [21] use a taxonomy that discriminates between four approaches towards automated usability testing. The taxonomy is closely linked to the common usability evaluation activities described in Section 1.4.3:

- *No automation*: all steps of the evaluation method are performed by the evaluator, i.e. there is no automation at all;
- *Automatic capture*: software automatically logs usability data such as interface usage, user speech, or visual data;
- *Automatic analysis*: software identifies usability problems automatically by using logged data; and

- *Automatic critique*: software points at improvements and fully automates analysis.

While automated usability evaluation is a solution to many problems in the field, it must be noted that automation might miss important subjective information such as preferences and misconceptions of the users [21]. To mitigate these issues, automated usability evaluation should be used in combination with traditional evaluation methods, such as heuristic evaluation or questionnaires. These traditional methods are described in Section 1.4.2.

2 Requirements specification

In this chapter we describe the `muEvaluationFramework` in terms of external behavior. This includes a use case model to present the functionalities of the framework, an object model that explains the application domain concepts, a dynamic model that shows the framework's behavior in terms of interactions, and the nonfunctional requirements that represent user-visible aspects of the framework that are not directly related with the functional behavior.

2.1 Objectives

As mentioned before it is the goal of this thesis to develop a framework that allows a developer to monitor and analyze the usability of mobile applications. The framework should be capable of monitoring the activity of the user and be able to capture usability related data. The information collected during this step should be analyzed and finally reported to the developer in an understandable and well-readable fashion.

2.2 Related work

In this section we present three existing frameworks that support remote usability evaluation for mobile devices. The first framework is called *MultiDevice REMUsine*. It is used to analyze the usability of Windows CE applications by comparing user behavior to a planned task model. The second project is the *EvaHelper Framework* that helps developers to analyze the usability of applications on the Google Android platform. The third software is *Google Analytics for Mobile* which extends Google's existing user tracking technology for web pages towards the Google Android and Apple iOS mobile platforms.

2.2.1 MultiDevice RemUsine

Paternò et al. describe a framework called *MultiDevice RemUsine* for the remote evaluation of mobile applications on the Microsoft Windows CE platform [27]. The solution

focuses on the collection of contextual information, that is, about the surrounding environment of the users. The MultiDevice RemUsine framework consists of a logging tool that collects usability data (capture), as well as a software that analyzes the collected data and provides an interactive view of the results (analysis and critique).

The logging tool, called Mobile Logger, supports the detection of user interactions within mobile applications and also the detection of environmental conditions that might affect the user's activity. Such conditions are, for example, noise in the environment or the battery level of the device. The logging component stores usability data as events and supports four event classes: contextual events that are created as a consequence of updated contextual information; intention events that allow users to indicate that they changed the target task; system events that are generated by the system as a response to user actions; and interaction events which are further specialized into categories such as clicking, selecting, and scrolling. All event types are collected by tracking the internal messages that are sent to the application by the Windows CE operating system. Because numerous message types exist, the logging functionality is distributed across several modules that track the activity of the user in one specific aspect.

To analyze the collected usability data, the framework uses an internal task model that describes how a user should complete tasks in the application. The task model has to be provided by a developer in a preparation step. The collected events are then analyzed by comparing the planned user behavior (the task model) and actual user behavior (the sequence of events that the user performed). This allows the framework to detect deviations from the internal task model; these deviations can then be investigated by a developer to see if there is a problem with the application's usability or if the task model must be updated to accept the new user behavior.

2.2.2 EvaHelper Framework

Balagtas-Fernandez and Hussmann found that software-supported usability testing on mobile devices is often troublesome because the devices have many restrictions and software tools for usability testing are missing. They describe a framework, called *EvaHelper*, which provides tool-support for usability evaluations on the Google Android platform [9].

The process of conducting a usability evaluation with the EvaHelper framework is structured by the authors into four common activities that are performed as consecutive phases: First, *preparation*, where the developer sets up the application for the logging of usability data. Second, *collection*, where usability data about the application are collected. Third, *extraction*, where the collected data are made available for interpretation. And fourth, *analysis*, where the extracted data are interpreted to detect usability problems in the application. The EvaHelper framework consists of several tools which support each of the activities.

Usability data are made available to the framework by calling log functions in the application that should be analyzed. These source code modifications must be performed by the developer, but the authors also propose a software tool which automatically inserts the necessary code into the application. Once this preparation step is complete, the framework is able to collect usability data. The data are stored in a text file as comma-separated values (CSV).

The framework also performs an analysis of the collected data by generating a graph that visualizes the user interactions. Each user interface (UI) element that was accessed by the user is represented as a node in this graph. The UI elements are then grouped by assigning them to containers which represent the screens or views of the application. Interactions or transitions are represented by the edges of the graph. This approach to visualization allows evaluators to retrieve information about an application's usability in the learnability, efficiency, memorability, effectiveness, error rate and simplicity dimensions.

While its approach towards automated usability evaluation is generally very nice, we think that the the EvaHelper framework could be improved in two ways. First, it is possible to detect user interactions automatically by using the reflection abilities of modern programming languages. This means that no additional software tool that inserts logging code into the application would be needed. Hence, the preparation step would become much easier for the developer because no source code must be changed. Second, we believe that on today's graphically-rich mobile UIs, it is important to collect user interactions graphically, for example as screenshots or video sequences. We believe that this yields valuable insights to the usability of an application, and especially so in an asynchronous remote evaluation setting.

2.2.3 Google Analytics for Mobile

Google Analytics allows a webmaster to track user activity on standard web pages. This is done by adding the Analytics JavaScript code to existing web pages. The embedded code, which is provided by Google Inc., collects information about actions that the users perform on the web page, for example navigating to a different web page or interacting with elements within one page. This tracking data are then sent to Google's servers where they are analyzed and a report is prepared that can be browsed by the webmaster.

Google Analytics for Mobile is an extension of the Google Analytics framework for web pages, and similarly, it allows developers to track user interactions within mobile applications [17]. The Google Analytics for Mobile framework is available for the Google Android and the Apple iOS platforms. It consists of two parts: First, a native software development kit (SDK) that replaces the JavaScript code of the web-based version. And second, an underlying web service, which is provided by Google free of charge, that stores and analyzes the collected data.

Before data can be collected, developers must setup an account with Google Inc. and must then link the application they want to analyze against a library from the Analytics SDK. Data collection or *tracking* is performed in a similar way to Google Analytics for web pages, but is less automated, i.e. developers have to trigger named events and *pageviews* (which provide a mechanism for grouping events) at appropriate times by calling a special function in the Analytics library. On the Google Android platform, the framework can also track clicks on advertisements that are embedded into the application. The collected data are stored internally into a SQLite[1] database and are wirelessly transmitted to Google's web service. Data transmission is performed in batches and developers can choose when the data should be dispatched.

The framework provides a web application which summarizes the collected data from all users and presents it to the developers; this data processing step can include the collected data of thousands of users. Developers have access to several tools, such as a graphing tool and detail reports, which they can use to inform themselves about several variables: how often the application is launched by the users (*visits*), how long users interact with the application (*session length*), the percentage of users that leave the application very quickly and never return (*bounce rate*) and the number of unique users who use an application (*unique visitors*). The data are displayed anonymously, that is, developers cannot get information about individual users.

2.2.4 Summary

We examined three existing approaches towards automated remote usability evaluation on mobile devices in the related work section. While the presented solutions are very satisfying in some aspects, we think that there is a distinct lack of such systems for the Apple iOS platform even though it is very popular at the moment. Furthermore, existing systems ignore the graphically-rich user interfaces of modern, touch-based mobile platforms and provide no support to graphically record user interactions, for example, as screenshots or video sequences.

2.3 Scenarios

In this section we present three usage scenarios for the `muEvaluationFramework`: First, a problem scenario that documents the issues and pitfalls with current approaches towards usability evaluation on mobile platforms. Second, a visionary scenario that includes a

[1] SQLite is a lightweight relational database that implements most of the SQL standard. It is often used in the software of mobile devices [19].

view of how usability evaluation on mobile platforms could be performed in the future. And third, a demo scenario that shows how the `muEvaluationFramework` helps a developer to conduct an usability evaluation.

2.3.1 Problem scenario

Scenario name	Problem scenario
Participating actor instances	John and Clara (developers); Alice, Marcus, and Mona (test users)
Flow of events:	1. The mobile application developer John is working on the *weMakeWords* application for the Apple iOS platform. The aim of the software is to playfully teach Chinese characters to children aged four to eight. John performs an usability evaluation of the *weMakeWords* application using an *iPod Touch* device. 2. After adding a feature that makes a new character available for learning, John notices that the five year old user Alice has trouble playing through a full session of the game. The application seems to freeze before Alice can finish the session. This never happens when John interacts with the application. John asks Alice about the problem she is having with the program. But, because of her age, Alice fails to explain the steps she took before the problem appeared. 3. John decides to monitor Alice's interactions with the game in order to find out what leads to the problem. While Alice is using *weMakeWords* once more, John sits besides her and takes written notes of her actions. 4. Alice is shifting around a lot while interacting with the *iPod Touch* mobile device and thus it is hard for John to take proper notes of all interactions. Later looking at them, it is hard for him to remember all steps that Alice performed and he fails to reproduce the problem that came up during the session. 5. John comes up with a new idea and tries to record Alice's interactions by using a video camera. The resulting video again is not very useful, as Alice is obstructing the camera view with her fingers. And she becomes upset because she cannot freely move around with the device in her hands while the screen is being filmed. 6. Sadly, the video of the recorded session also proves inconclusive because the issue did not appear this time. John has now been working effortlessly for about two hours. He resolves to filming more sessions

and invites two other children, Marcus and Mona, to interact with the application in order to reproduce the freezing issue. To take good care of the bigger test group he requires the help of another software developer, Clara.

7. After spending several hours recording the children's interactions together with Clara, John starts watching the recorded videos and searches them for occurrences of the problem.
8. Feeling very exhausted, he finally manages to fix the fault around midnight after a long and dull evening watching recorded interactions.

Problems in this scenario:

1. Recording user interactions by hand is tedious for John, even more so on mobile platforms. The quality of the monitoring results is not very good, because the small screens get obstructed easily and taking notes by hand is slow.
2. Conducting a larger number of usability evaluation sessions becomes very work intensive. John cannot do this by himself.
3. He has to search for potential problem areas by hand. It is difficult for him to browse many recorded sessions in order to find a single issue. Application freezes could be detected easily by a simple heuristic that scans for screen updates, for example.
4. In order to allow for proper monitoring of her interactions, Alice is not allowed to use the software the way she would in an unrestricted situation, that is, she is forced to be still while playing with the program so that the screen can be filmed or notes can be taken.

2.3.2 Visionary scenario

Scenario name	Visionary scenario
Participating actor instances	John (developer)
Flow of events:	1. The developer John wants to test a new version of the *Wordpress for iOS* application with a test group of 40 individuals. As a freelance developer he has only limited testing resources available, so he goes online and decides to acquire application testers via the website *www.evaluate-my-app.com*. He makes a request to the system for 40 test users and specifies their required properties: they should

be 20 to 30 years old; half of them should be male, half of them female; they should all own *iPhone 4* devices, and one third of them should be novice users. Additionally, John wants the testing session to be less expensive than 500 EUR. The system chooses the testers automatically according to their profile and previous work.

2. All testers are guided through the process of installing the application. John gives each of them a common task that stresses the newly implemented *Delete weblog posting* feature.
3. The testers begin working on the tasks they were given. The monitoring framework records all the user interactions in the test group. It generates an interactive report from the cumulative data.
4. The report contains:
    - a timeline of events for each of the testers; including touch events, device motion and shake gestures
    - visual and audio recordings of the interactions for think-aloud protocols (Screen touches are indicated via graphical effects)
    - the think-aloud audio recordings are transcribed automatically to text form
    - the think-aloud audio recordings are searched for user exclamations ("Oh no!", "What happened?", etc.) that hint at problems
    - a video recording of the testers face via the built-in front camera that is scanned for strong emotions (being upset, surprise, etc.)
    - a graphical representation of the tester's navigation through the applications's view hierarchy
    - a list of the most used views and user interface (UI) elements
    - a rating of the applications's performance and visual fluidity
    - a list of to small or hard to read text elements
    - a list of buttons or UI elements that were missed by users several times. This is hinting at the elements being to small to use well.
    - a list of areas that might be hard to read for people with disabilities, for example people with colorblindness. Also pointed out are the UI elements that do not have good accessibility metadata (iOS has support for users with impaired vision, for example)
    - eye-tracking data that graphically shows where the user was looking at; these data are either recorded via an external system or inferred from the front camera video feed

- a reachability graph of all the views; hints where shortcuts might be needed
- a list of occasions where identical information was repeatedly entered by the user; this hints at areas where the data might be simply entered by the application itself to minimize typing required by the user
- a list of interactions that did not cause visual feedback when the user performed them
- a list of "UI design rules of thumb"-violations, e.g. the application is using to many different fonts or using inconsistent colors

5. Using the report John can now more easily identify problem areas in the usability of his application. John has saved several days of work because he did not have to do the usability evaluation on his own.

2.3.3 Demo scenario

Scenario name	Demo scenario
Participating actor instances	John (developer); Jack, Jill, and Jane (test users)
Flow of events:	1. Mobile application developer John is working on *Wordpress for iOS*, an editing software for weblogs. He wants to know how a group of testers, Jack, Jill and Jane, are interacting with the software when given the task of publishing a new posting to a weblog. 2. John includes the `muEvaluationFramework` framework into the application, making only minor changes to the project source code. For example, he triggers events when key locations in the interface code are reached by using simple function calls to mark them: *[monitor beginEvent: @"Edit blog posting"].* 3. After that, he starts the *MobileMonitor* application, launches the *Wordpress for iOS* application on three *iPod Touch* devices and gives them to the testers. They were each given a written statement with the task of publishing a new blog posting. Also, they were asked to think aloud about their actions while interacting with the application. 4. While the testers are working, the `muEvaluationFramework` records user interactions with the *Wordpress for iOS* application and forwards them to an instance of the Mobile Monitor application running

on John's computer. Screen contents of the devices and the voices of the testers are also recorded and forwarded.

5. After Jack, Jill and Jane finish their assigned tasks, John uses the Report Generator software on the data recorded by the framework. A human-readable report is generated and John can view it using a web browser.
6. The report includes a timeline of the testers interactions with the *Wordpress for iOS* application. It includes generic events such as *Button "Publish" was pressed* or *User entered view "Write posting"* and also custom events that were defined by John: for example, *User executes a shake gesture* or *User selects category "Work"*. John can choose to exclude certain events from the timeline in order to make the stream of events less cluttered. Furthermore, the report includes summarizing information concerning the whole interaction. For example, a list of the most often generated events or the most often used views as well as a heat map for each view that shows where the user touched the screen.
7. John now uses the information from the report to improve usability issues in the *Wordpress for iOS* application. Also, he finds out about hot spots in the interaction; that is, user interface elements that were accessed frequently by Jack, Jill and Jane. Using this data he can later decide efficiently on what area of the application he should work on next in order to improve usability where it is most important.

2.4 Functional requirements

In this section we describe how users and any external systems interact with the `muEvaluationFramework`. The functional requirements are independent of the framework's implementation and follow the structure of the three common usability activities described in Section 1.4.3: First, the *capture phase*, during which usability data are collected. Second, the *analysis phase*, where the collected data are interpreted and examined for usability problems. And third, the *critique phase*, where improvements to the detected usability problems are suggested.

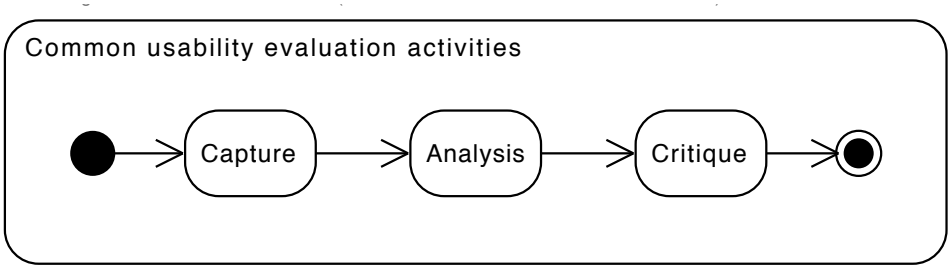


Figure 2.1: Common activities during usability evaluation (UML activity diagram)

2.4.1 Capture phase support

Usability data collection in sessions During the capture phase, the `muEvaluationFramework` should gather and store usability data belonging to a specific application. This application is called the host-application. Usability data should be collected only in the course of an evaluation session. An evaluation session should be initiated by the developer and exactly one test user should be able to participate in each session. The usability data collection during a session should be performed by a number of software-based sensors that attach to the host-application.

Session control An evaluation session should be started and stopped by the developer. The developer should be able to control which data are collected by selecting a set of active sensors before the evaluation session starts. As soon as the developer stops the evaluation session, the capture phase should end and the framework should start the analysis phase.

Support for multiple usability data sources During the evaluation session, the active sensors should continuously observe a number of sensor targets in the host-application. Such sensor targets should be function calls or variables in the address space of the host-application that are monitored for state changes (for example, a function is called or the value of a variable changes). As output, sensors should generate events to store their findings. A single event should contain a description of the state change of the sensor target and the time at which the change occurred. The `muEvaluationFramework` should support the following event types:

- *User-input events*: direct input actions by the test user, such as touch gestures (e.g. screen taps), or text entry.
- *User-interface events*: events related to the user interface (UI) of the host-application, for example, the selection of UI widgets (e.g. button presses), or changes in the on-screen content. This is different from user-input events because user-interface events

are events that lie completely within the functionality of the host-application, i.e. a test user might touch the screen of the device (this generates a user *input* event) but the UI of the host-application might not register the touch and hence no user *interface* event is generated.

- *Device sensor events*: modern mobile devices have a number of built-in sensors (e.g. accelerometers, light sensors, video cameras and microphones) that can be used to track the test user's behavior. Device sensor events should encapsulate the data produced by these sensors (e.g. audio samples or video frames).
- *Application events*: internal state changes in the host-application. For example, launch and termination, bringing the host-application to the foreground or the background, as well as exceptions that were raised by the underlying runtime environment.
- *Custom events*: events that are completely under the control of the developer. Custom events should be raisable by calling special functions in the host-application's source code.

Live preview for collected events While an evaluation session is active the developer should see the events as they are recorded by the framework. This helps to ensure that events are successfully captured and allows to developer to see what the test user is doing in a synchronous remote evaluation setting.

2.4.2 Analysis phase support

Usability data interpretation During analysis the framework should interpret the usability data, e.g. the events, that were collected in the capture phase. To do so, several interpreter algorithms should process the events and generate interpretation results that should be used in the subsequent critique phase. The interpretation results should either describe a particular usability problem in the host-application or they should contain summarizing and statistical information collected during the evaluation session. When all interpreters have completed their work, the analysis phase should end and the framework should advance to the critique phase.

Usability problem detection The framework should be able to detect violations against the *iOS Human Interface Guidelines* (HIG) that are provided by Apple Inc. [6]. Because many of the HIG are diffuse and hard to quantify with machine verifiable rules, the framework must only detect when a host-application violates a quantifiable guideline such as:

- "**Display a hint in the text field if it helps users understand its purpose**, such as 'Name' or 'Address.' A text field can display such placeholder text when there is no other text in the field." [6]; or
- "**Maintain a hit target area of at least 44 x 44 pixels for each toolbar item.** If you crowd toolbar items too closely together, people have difficulty tapping the one they want." [6]

Usability summary generation The summarizing and statistical interpretation results that should be produced by the framework are:

- overview information regarding the evaluation session, for example, the session's duration or the total number of events;
- a list of the most used views and UI widgets that is sorted by the time spent on each element;
- heat maps that should graphically show the touch events for each view in the host-application;
- a representation of the test user's navigation path through the host-application's views

2.4.3 Critique phase support

Report generation As final output, the `muEvaluationFramework` should generate a report that summarizes the interpretation results that were found in the analysis phase. For each usability issue that was found, the report should recommend changes to the host-application that mitigate the issue. The developer should be able to use the report to achieve two goals: First, to understand and to reproduce any actions the test user has performed during the evaluation session. And second, to improve the usability of the host-application by making the changes that were recommended in the report. The developer should be able to view the report using a standard web browser. The report should be structured into sections and should offer interactive elements (such as dynamically filterable event logs). Where applicable, the report should include media such as images, videos, or audio files.

Report configuration Developers should be able to configure the report in two ways: They should be able to choose the directory where the report is stored; and they should be able to choose one or more sections that are included in the report. Reports should support the following section types:

- an overview section that lists general information about the evaluation session, e.g. its duration or the number of recorded events;
- a graphical event log or timeline that can be interactively filtered by event type;
- a touch heat map section that graphically shows user interaction hot spots for each view in the host-application; and
- a section that shows how the test user navigated through the views of the host-application.
- a section that shows the *Human Interface Guidelines* [6] violations that were detected.

2.5 Nonfunctional requirements

After the functional requirements of the `muEvaluationFramework` are defined, we specify a number of additional nonfunctional requirements. These requirements are user-visible aspects of the framework that are not directly related with the functional behavior.

2.5.1 Usability

Minimal setup work Only minimal setup work should be required from the developer to perform a usability evaluation on an existing host-application. The framework should only require two modifications to an existing host-application: first, linking the host-application against a single static library; and second, calling one method in the host-application's source code.

User interface usability The user interface used for report configuration should be accessible within five seconds after a session was recorded in order to not interrupt the workflow of the developer. The developer should be able to start and stop an evaluation session with a simple graphical user interface. The starting and stopping of a session should be possible by performing just one mouse click. A graphical user interface should also be available to configure and generate a report. The report should be pre-configured so that the developer can generate a report with a single click.

Wireless communication with the mobile device The framework should communicate with the mobile device over a wireless connection. This is preferable to a tethered connection because it allows test users to move the mobile device around freely and not feel restricted by wires.

2.5.2 Reliability

In order to provide meaningful results, the framework should not make the host-application behave differently. Any side effects on the host-application should therefore be minimized and influence on the test user's behavior should be as low as possible.

2.5.3 Security

The framework may not compromise the security of the host-application. Existing safety measures of the operating system, such as application sandboxing, should not be affected.

2.5.4 Privacy

Recorded user interactions and other events should only be accessible to an authorized developer.

2.5.5 Performance

To ensure meaningful data, the framework should not slow down the host-application to the point that its usability is affected. Together, the analysis and critique phases should take less than 10 minutes for every hour spent in the capture phase.

2.5.6 Supportability

Full source code comments should be provided for all code in the framework. The framework should be structured so that it can be easily extended with new capturing abilities, interpretation algorithms, and report section types.

2.5.7 Implementation requirements

The prototypical implementation of the framework should be usable with host-applications developed for the Apple iOS platform. Therefore, parts of it must be programmed in the Objective-C language. The framework should also be prepared for future cross-platform compatibility. To do so, all code that does not directly run on the mobile device should be programmed in the Python language.

2.6 System models

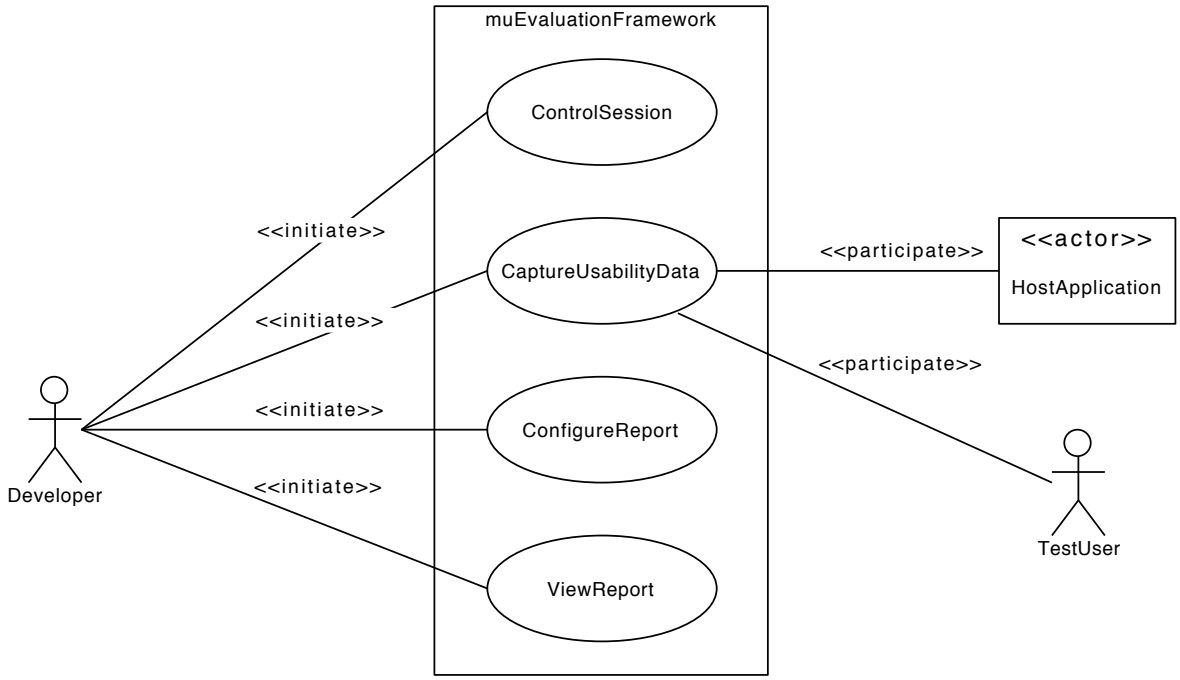


Figure 2.2: The main use cases for the actors `Developer`, `TestUser` and `HostApplication` (UML use case diagram)

2.6.1 Use case model

From the requirements described in Sections 2.4 and 2.5 three actors and three use cases were identified. The use cases specify how the actors interact with the system (Figure 2.2). In this section, we describe the actors and explain each use case.

Actors

From the objectives and functional requirements, the actors `Developer`, `TestUser`, and `HostApplication` were identified:

- `Developer:` The `Developer` wants to assess and improve the usability of the `HostApplication`. To do so, the `Developer` uses the `muEvaluationFramework` to capture usability data and to generate a report that contains information about the usability of the `HostApplication`.
- `TestUser:` The `TestUser` interacts with the `HostApplication` during an evaluation session. His or her actions are captured and form the basis of the usability analysis.

- `HostApplication`: The `Developer` uses the `muEvaluationFramework` to assess and improve the usability of the `HostApplication`.

Use cases

As a result from the functional requirements, the following activity was identified as the main use case for the actors `Developer`, `TestUser`, and `HostApplication` (Figure 2.2):

- `CaptureUsabilityData`: Usability related data are captured during an evaluation session initiated by the `Developer`. The actors `TestUser` and `HostApplication` participate in the evaluation session.

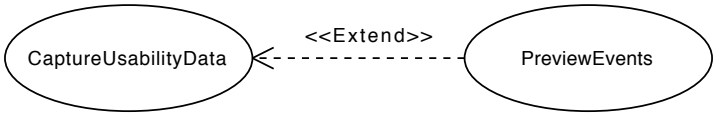


Figure 2.3: The `PreviewEvents` use case (UML use case diagram)

To satisfy the "Live preview for collected events" requirement, the `PreviewEvents` use case is inserted, which extends the `CaptureUsabilityData` use case (Figure 2.3):

- `PreviewEvents`: The `Developer` watches a live preview of the captured events during an evaluation session.

Now that we have identified the the main use case of the framework, we describe three new use cases for the developer and then further refine one of the new use cases. The following three use cases were identified for the actor `Developer` (Figure 2.2):

- `ControlSession`: The `Developer` selects the active sensors and starts or stops a session.
- `ConfigureReport`: The `Developer` configures the contents of the report.
- `ViewReport`: The `Developer` views the report that summarizes the usability issues found in the `HostApplication`.

When inspecting the `ControlSession` use case it became obvious that it is very wide in scope. Subsequently, the three specialized use cases `SelectSensors`, `StartSession`, and `StopSession` were identified (Figure 2.4):

- `SelectSensors`: Before an evaluation session starts, the `Developer` chooses a set of sensors that are active during the session.
- `StartSession`: The `Developer` starts the evaluation session.
- `StopSession`: The `Developer` stops the evaluation session.

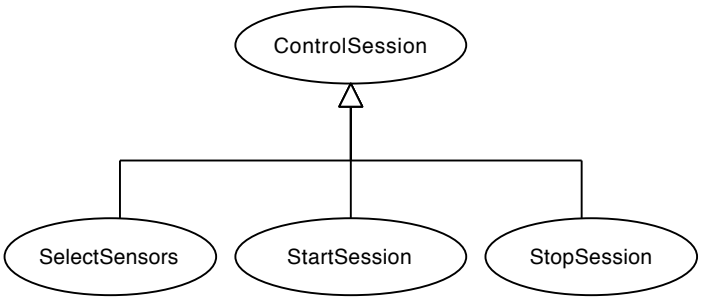


Figure 2.4: Specialization of use case `ControlSession` (UML use case diagram)

2.6.2 Object model

This section describes the conceptual model of the `muEvaluationFramework`. The included objects were identified during the requirements elicitation phase and are now explained. Because many objects were identified, the object model is split into several conceptual sub-models.

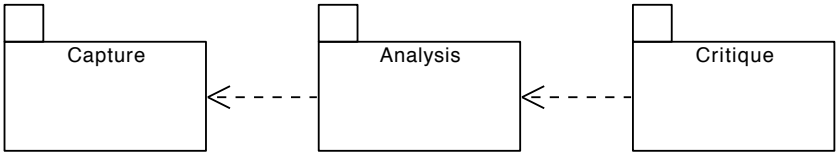


Figure 2.5: Relationships of the sub-models (UML package diagram)

From a high-level perspective, the automated usability evaluation performed by the framework follows the three-step pattern of consecutive *capture*, *analysis* and *critique* phases as explained in Section 1.4.3. The following sections of the system description also reflect this pattern: First, we describe the object model of the capture phase. Second, we describe the object model of the analysis phase. And third, we describe the object model of the critique phase.

Capture phase

During the capture phase, the `muEvaluationFramework` gathers and stores usability data belonging to a host-application. The collection of usability data is performed during an evaluation session in which a test user interacts with the host-application.

In this section we will explain: *when* usability data are captured; *what* kinds of usability data are captured; *how* the data are captured; and *where* the captured data are stored for later use. We will do so by grouping the object model of the capture phase into three parts:

First, we provide a general overview of a usability evaluation session and describe the objects that are involved with it. Second, we explain what usability data are and how they are collected by sensors. Finally, we explain how the framework stores the collected usability data for use in the analysis phase.

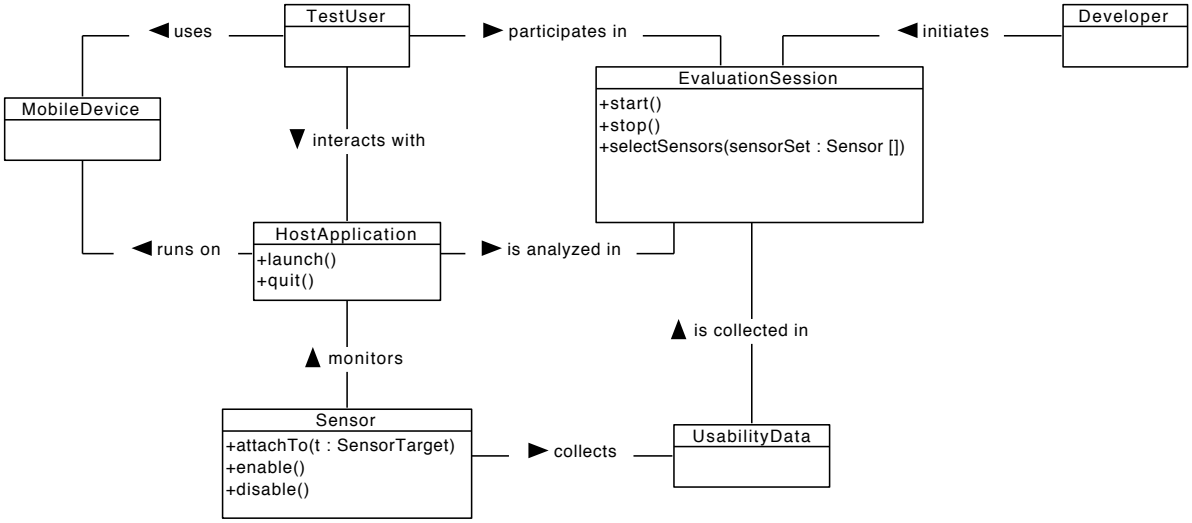


Figure 2.6: High level overview of the participating entities of the capture phase (UML class diagram)

Usability evaluation sessions The framework analyzes the usability of an application that runs on a `MobileDevice`. We call this application the `HostApplication`. The framework collects usability data (represented by the class `UsabilityData`) about the host-application during the capture phase of an `EvaluationSession`. We describe the different kinds of usability data and how they are detected in the next paragraph. But first, we explain the concept of evaluation sessions. An overview of the related objects is given in Figure 2.6.

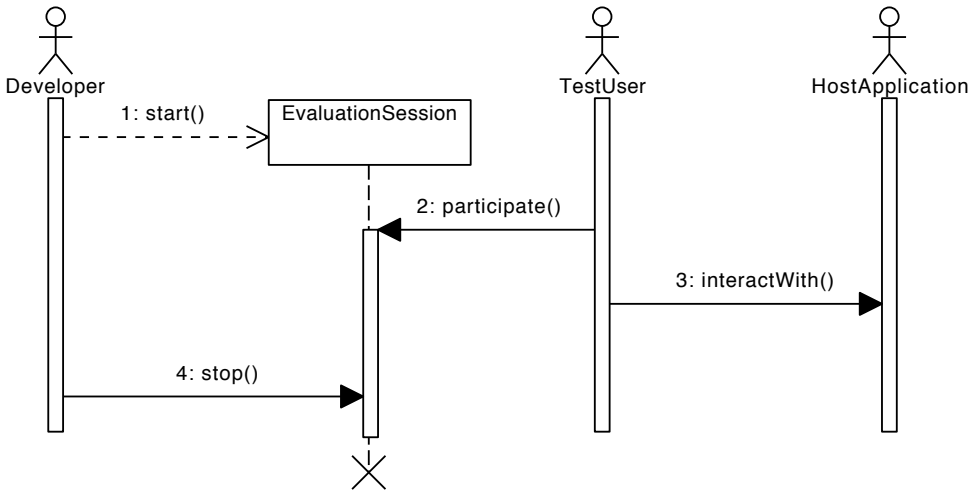


Figure 2.7: The life cycle of a usability evaluation session (UML sequence diagram)

Two persons participate in an evaluation session: the `Developer`, who controls the session; and a `TestUser`, who uses the mobile device during the session to interact with the host-application. When the `Developer` starts the `EvaluationSession`, the framework begins to monitor the `HostApplication` and starts to collect `UsabilityData`. The `Developer` can stop a running `EvaluationSession` at any time; this stops the data collection and ends the capture phase. This life cycle is depicted in Figure 2.7.

Usability data collection In this section we describe usability data by answering the following questions: "*What* are usability data?" and "*How* does the framework capture usability data?".

First, we define the term usability data. As the name suggests, usability data are related to the usability of the host-application that is analyzed during an evaluation session. The usability data that the framework collects in the course of an `EvaluationSession` are represented by the `UsabilityData` object. There are three factors that influence `UsabilityData` (Figure 2.8):

- The `TestUser`, who interacts with the `HostApplication`. Different test users, or even the same user, might show different performances in each evaluation session.
- The `HostApplication`, which the `Developer` wants to improve usability-wise.
- The `MobileDevice`, which executes the `HostApplication`. Both the device's hardware and its software influence the performance and behavior of the test user. For example, the hardware of different mobile devices might vary in screen resolution, screen size, or device weight. These are all aspects that affect usability. Similarly, the software that runs on a mobile device, for example the underlying runtime en-

vironment or operating system, influences the test user as well. A newer version of the operating system, for example, might offer better graphics performance or enhanced text entry capabilities and thus alter the usability.

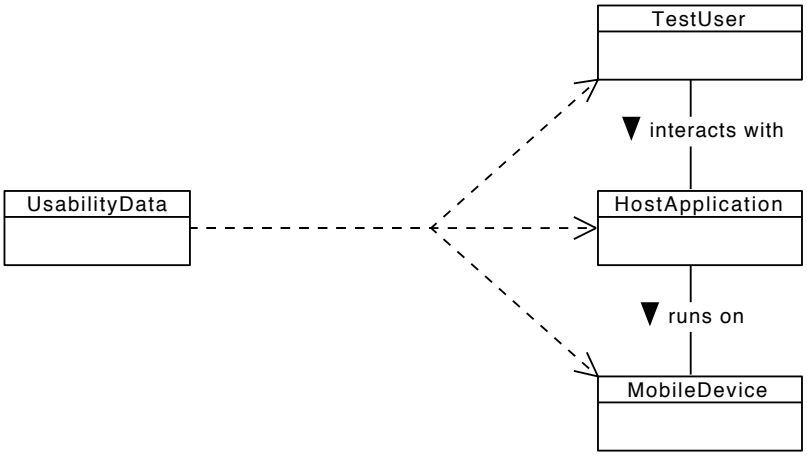


Figure 2.8: Influencing factors on usability data (UML class diagram)

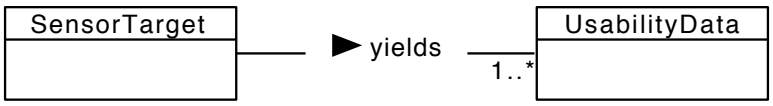


Figure 2.9: The relationship between `SensorTargets` and `UsabilityData` (UML class diagram)

The question arises how usability data can be detected in a host-application. In the `muEvaluationFramework`, usability data are collected from `SensorTargets` which can be observed by `Sensors` to yield the wanted usability data.

Before the `EvaluationSession` starts, the `Developer` selects a set of `Sensors` to control the types of usability data that are collected. The `Sensors` are the framework's means of collecting usability data; `Sensors` are software-based and do not exist as physical objects in the world. Instead, a `Sensor` is program code that attaches to one or more `SensorTargets`, which reside in the `HostApplication`, and observes them.

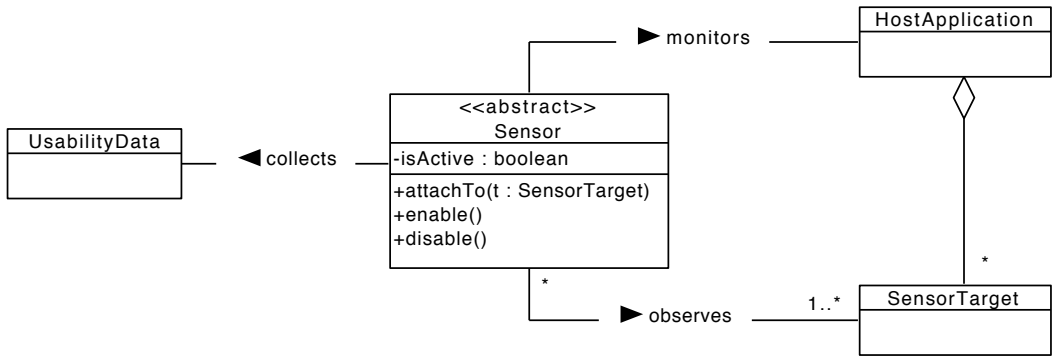


Figure 2.10: `Sensors` and `SensorTargets` (UML class diagram)

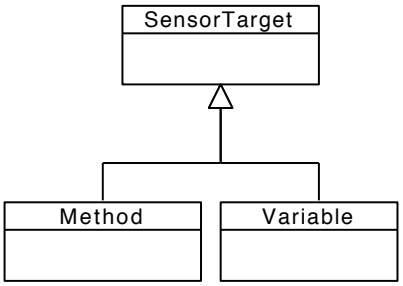


Figure 2.11: `SensorTarget` specializations (UML class diagram)

The framework uses two types of sensor targets: `Methods` and `Variables`.

A `Method` sensor target is, for example, a function in the host-application's address space that is called every time the `TestUser` touches the device's screen. A `TouchInputSensor` can then "spy" on this function, and for each invocation of the function, record a timestamp and the arguments of the call. This sensor can tell the framework when and how the test user interacts with the `MobileDevice`'s touch screen. This information is clearly usability-related and is therefore within the scope of the the framework.

The second type of sensor target is called `Variable`. It represents the memory address of an important variable inside the host-application. A sensor either retrieves the contents of the variable every few seconds to see if the variable has changed (polling) or subscribes with the variable's owner object to be informed of any changes (interrupt-driven). An example variable might indicate the light level of the mobile devices's environment and can be observed by a `LightLevelSensor` to provide helpful usability data.

The framework supports a number of specialized `Sensors` and `SensorTargets`, such as:

- `TouchInputSensor`: tracks how the test user interacts with the mobile device's touch screen (e.g. finger tapping, scrolling and other gestures)

- `AudioSensor`: records sound from the mobile device's built-in microphone (if available)
- `ScreenSensor`: records the contents of the mobile device's screen, and
- `TextEntrySensor`: tracks any data entry to text input fields that is performed by the test user.

`UsabilityData` objects are abstract data that are not suitable for storage by the framework. Therefore, the framework internally stores the collected usability data as `Events` which serve as an representation of the usability data that is useful for later analysis. `Events` are explained in the next paragraph.

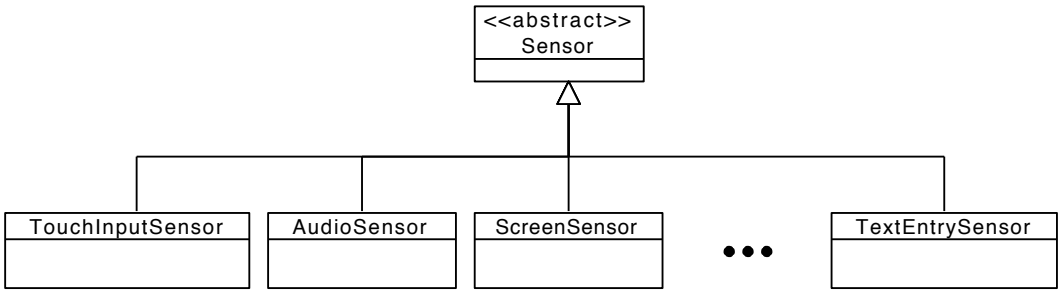


Figure 2.12: `Sensor` specializations (UML class diagram)

Usability data storage During the observation of `SensorTargets`, the `Sensors` collect `UsabilityData`. For storage, these usability data are wrapped into `Event` objects, which represent the usability data and contain additional metadata, such as a timestamp that shows when the event was created.

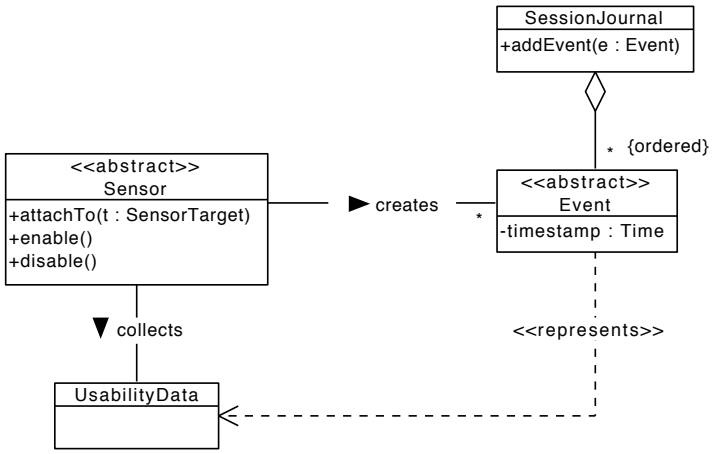


Figure 2.13: `Event` as a storable representation of usability data (UML class diagram)

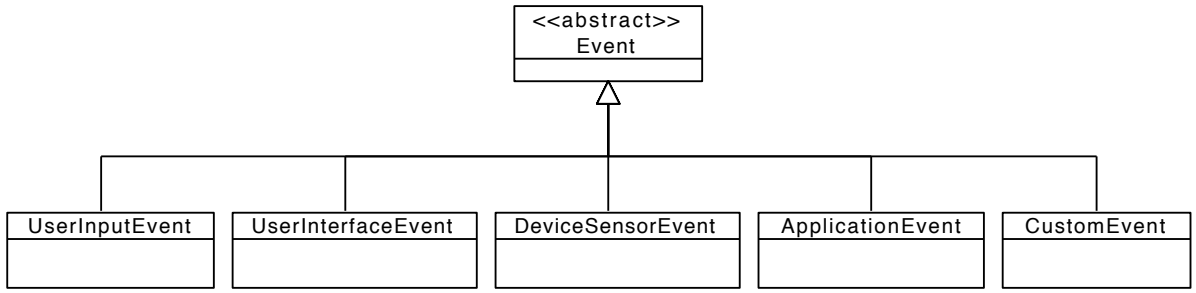


Figure 2.14: Specializations of the `Event` object (UML class diagram)

`Event` is an generalized base class for further specific event types which represent different kinds of usability data. The framework uses five specializations for `Events` that are depicted in Figure 2.14:

- `UserInputEvent`: stores direct input actions by the `TestUser`, such as touch screen input or text entry via the mobile device's keyboard (which could also be a virtual "software" keyboard).
- `UserInterfaceEvent`: describes interactions with the user interface (UI) of the `HostApplication`, for example, button presses or changing screen contents. This is different from `UserInputEvents` because user-interface events are events that lie completely within the functionality of the host-application, i.e. a test user might touch the screen of the device (this generates a `UserInputEvent`) but the UI of the host-application might not register the touch, so in this case no `UserInterfaceEvent` is generated. Usability issues of this kind do exist, for example, buttons that do not react to user touches because the buttons' event handlers are not set up correctly. Hence, this distinction between `UserInputEvent` and `UserInterfaceEvent` is needed.
- `DeviceSensorEvent`: encapsulates data provided by hardware sensors of the `MobileDevice`, for example, accelerometer data or GPS location updates.
- `ApplicationEvent`: are related to state changes of the `HostApplication`, for example, launching and quitting it. `ApplicationEvents` can also be exceptions that are raised by the underlying runtime environment.
- `CustomEvent`: represents events that are completely under the control of the `Developer`. This allows developers to generate specialized events by simply calling a function with the event name as a string parameter.

The `Events` that are collected in the capture phase must be stored for the following analysis phase that accesses them. Therefore, event storage is provided by the `SessionJournal` object.

User interface To satisfy the `ControlSession` use case and the `PreviewEvents` use case two user interfaces for the `Developer` are added. With the `SessionControlUI` object the `Developer` can start and stop evaluation sessions, as well as select the set of active sensors before a session starts. The `PreviewEventsUI` object allows the `Developer` to see which `Events` are captured during an `EvaluationSession`.

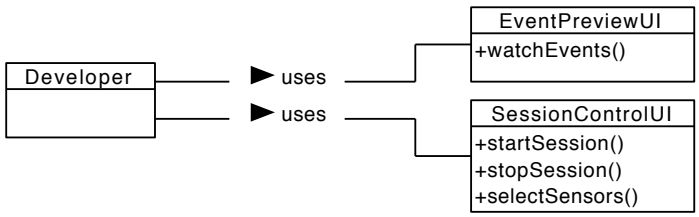


Figure 2.15: User interfaces for the capture phase (UML class diagram)

Analysis phase

In the analysis phase the `Events` that were collected during the capture phase are interpreted to provide the interpretation results required by the final critique phase. These results therefore have to be stored in order make them available to the next phase. In this section we describe how `Events` are interpreted in the analysis phase and how the interpretation results are stored.

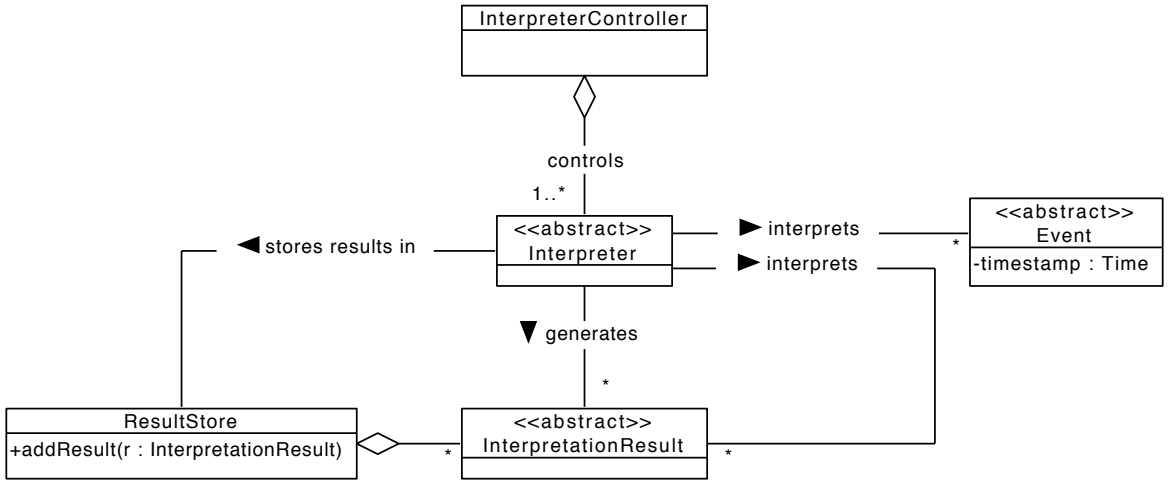


Figure 2.16: Overview of the analysis phase object model (UML class diagram)

Usability data interpretation The interpretation of the collected usability data is the most difficult aspect of the framework. During the analysis phase, multiple `Interpreter`

objects generate `InterpretationResults` which either represent a usability problem or provide summarizing information. Usability problems and summaries are represented by the `UsabilityProblem` or `UsabilitySummary` objects respectively.

`Interpreters` can either interpret events that were collected during the capture phase or they can further interpret already existing interpretation results. Such a blackboard[2]-based approach is very flexible and allows `Interpreters` to continuously produce more abstract and higher-level results by refining and working on previously generated interpretation results. Because interpreters can be dependent on the results of other interpreters, the framework must provide a way to resolve these dependencies and to find a valid order of execution. The whole interpretation session as well as the individual `Interpreters` are managed by the `InterpreterController`. The controller also determines the beginning and the end of the analysis phase.

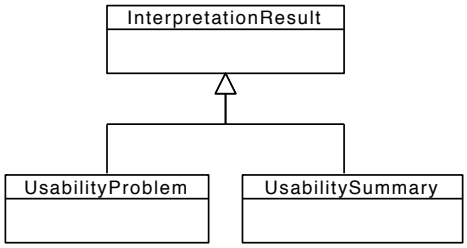


Figure 2.17: `InterpretationResult` specializations (UML class diagram)

Interpreter types The framework uses six different `Interpreter` types during the analysis phase. We now briefly describe the purpose of each `Interpreter` that is shown in Figure 2.18:

- `OverviewInterpreter`: provides summarizing information about the `EvaluationSession`. For example, when the evaluation session started, when it ended, and the number of events that were recorded.
- `ScreenshotInterpreter`: gathers the screenshots from the `Events` in the `SessionJournal` and sorts them by their timestamp, thereby creating a persistent ordering. This ordering can be used by other interpreters to reference a screenshot without duplicating the image data.
- `TouchHeatmapInterpreter`: gathers touch events and maps them to screenshot images.
- `ViewChangesInterpreter`: gathers all `UserInterfaceEvents` that indicate a view change in the host-application. The view changes are then mapped to the screenshot indices created by the `ScreenshotInterpreter`.

[2]The Blackboard design pattern [31]

- `ViewDurationInterpreter`: gathers the timestamps of all view change events and generates a statistic that shows for how long each view was active.
- `UIGuidelineChecker`: searches the host-applications view hierarchy that is contained in the `UserInterfaceEvents` for violations against the human-interface guidelines.

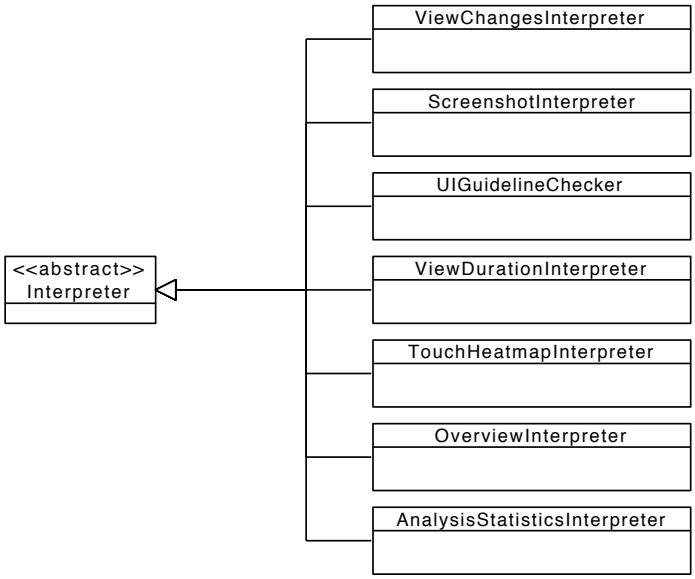


Figure 2.18: Interpreter types (UML class diagram)

Interpretation result storage Because of the three phase approach towards usability analysis taken by the framework, the generated `InterpretationResults` must be stored and made accessible to the subsequent critique phase. In the analysis phase, the `ResultStorage` object is responsible for the storage of `InterpretationResults` (Figure 2.16).

Critique phase

The functional requirements for the `muEvaluationFramework` state that, as final output, it should generate a report that summarizes the interpretation results that were found in the analysis phase (Section 2.4.3). The framework generates reports during the critique phase and we now describe the object model for this phase. To do so, we structure the description into three steps. First, we explain the document model used by the reports. Second, we describe how a developer configures a report. And third, we explain how reports are generated.

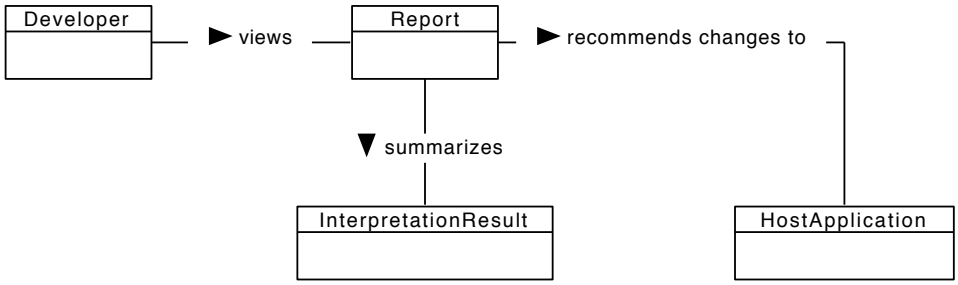


Figure 2.19: High-level object model of the critique phase (UML class diagram)

Report document model The report that is generated by the framework during the critique phase is represented by the `Report` object. `Reports` are viewed by `Developers` because they provide them with valuable information regarding the usability of the `HostApplication`. To help improve usability, a report recommends changes that should be made to the `HostApplication`. These suggested changes and helpful pieces of information are contained in the `InterpretationResults` provided by the previous analysis phase. A `Report` summarizes the information contained in the `InterpretationResults` for the `Developer`.

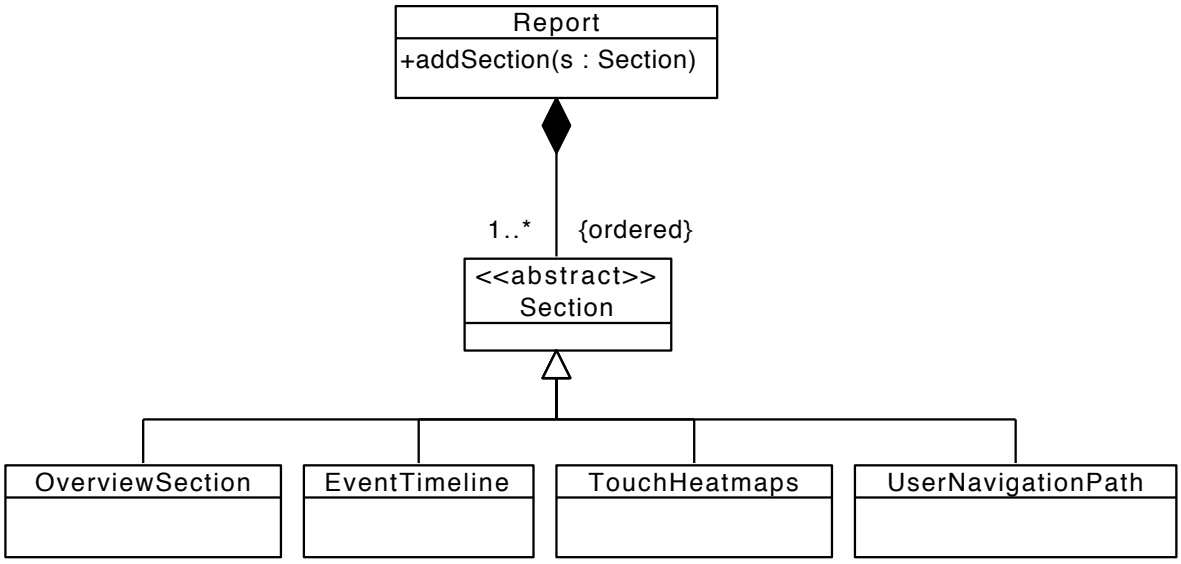


Figure 2.20: `Report` document model and `Section` specializations (UML class diagram)

Now that we have defined what `Reports` are used for, we explain the internal document model used by the reports. The document model of a report follows a simple structure: one `Report` consists of an ordered list of one or more `Sections`. Information for the developer is therefore contained in the `Sections`. The framework supports four `Section` types (Figure 2.20):

- `OverviewSection`: Provides general information about the report, for example, the name of the application that was analyzed and the number of events that were recorded.
- `EventTimeline`: An interactive list of the events that were recorded during an evaluation session.
- `TouchHeatmaps`: Shows how the user interacted with each view in the host-application. Finger touches are represented as colored circles.
- `UserNavigationPath`: Shows the test user's navigation path through the views of the host-application.

Report configuration Before a `Report` is generated the `Developer` must configure it. This allows the developer to choose where the resulting report is stored and which `Sections` are included. A finished report configuration is represented by the `ReportConfiguration` object, which contains the necessary information about the `Report`. As shown in Figure 2.21, this `ReportConfiguration` is then used by the `ReportGenerator` object (explained in Paragraph 2.6.2) to create the report document according to the configuration options provided by the developer.

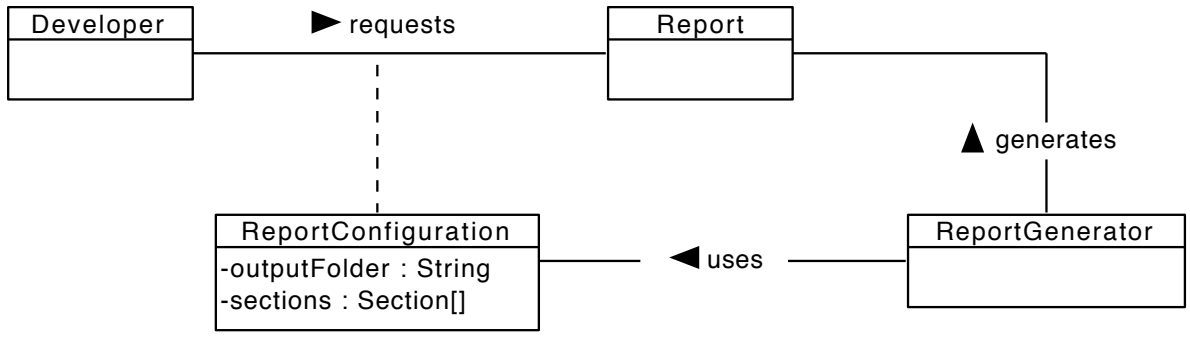


Figure 2.21: Report configuration (UML class diagram)

Report generation Reports are created by the `ReportGenerator` object which controls the report generation process. The report generator uses the configuration options provided by the developer to choose an output folder for the finished report document as well as to decide which sections must be included in the report. Each individual section is then created by the `SectionGenerator` object. The section generator uses the information contained in the `InterpretationResults` from the analysis phase to create `Sections` for the `Report`. The objects involved in the generation of a report are depicted in Figure 2.22.

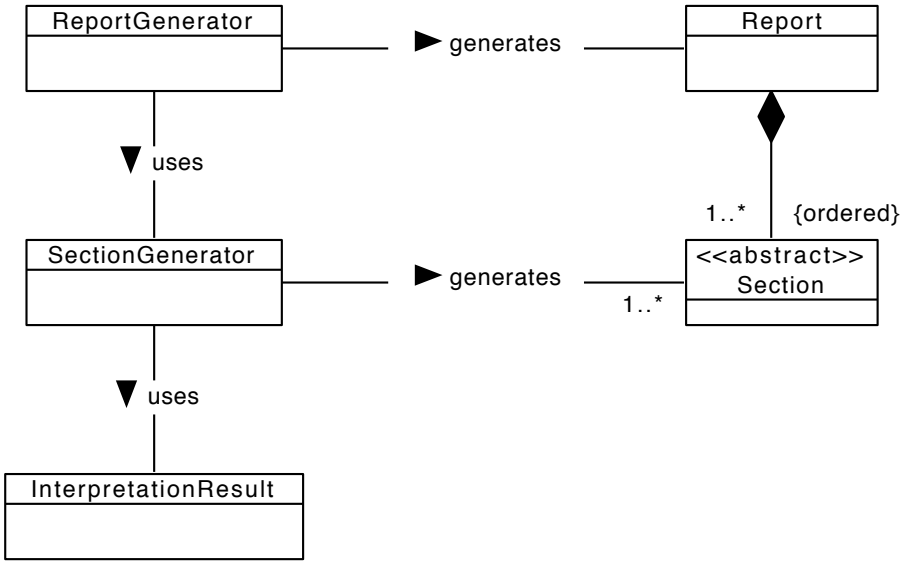


Figure 2.22: Report generation overview (UML class diagram)

2.6.3 Dynamic model

This section documents the behavior of the object model in terms of activity diagrams and sequence diagrams. Although some use case information is described redundantly, dynamic models enable us to specify the behavior more precisely. The dynamic model follows the general structure of the framework and is therefore divided into three parts: the capture phase, the analysis phase, and the critique phase. During the capture phase usability data are captured and stored as `Events` in a `SessionJournal`. These events are then used by the subsequent analysis phase to generate `InterpretationResults`, which are stored in the `ResultStore`. The interpretation results are finally used in the critique phase to generate a `Report` for the developer. This high-level behavior is shown in Figure 2.23.

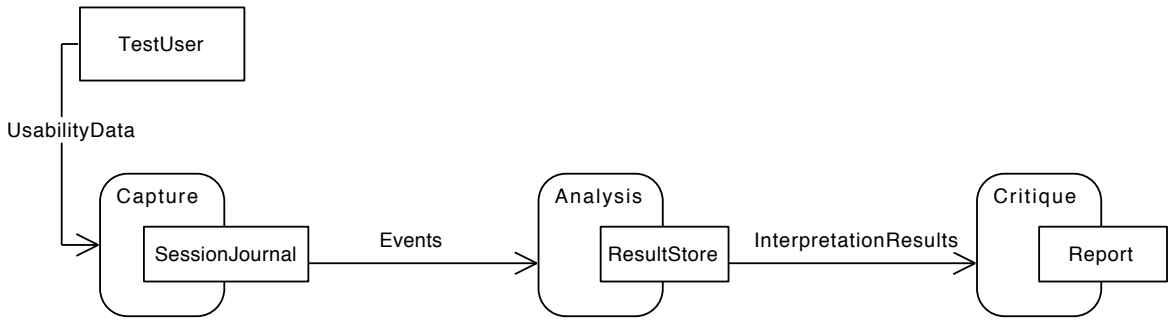


Figure 2.23: High-level dynamic model of the framework (UML activity diagram)

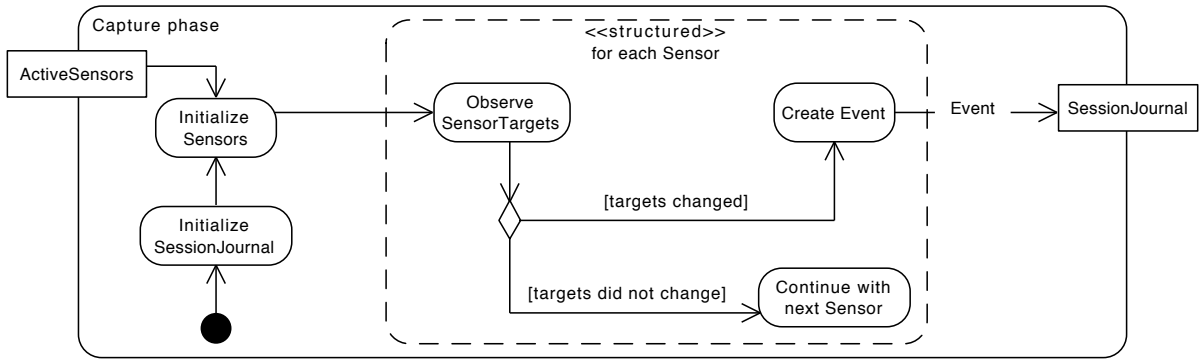


Figure 2.24: Capture phase (UML activity diagram)

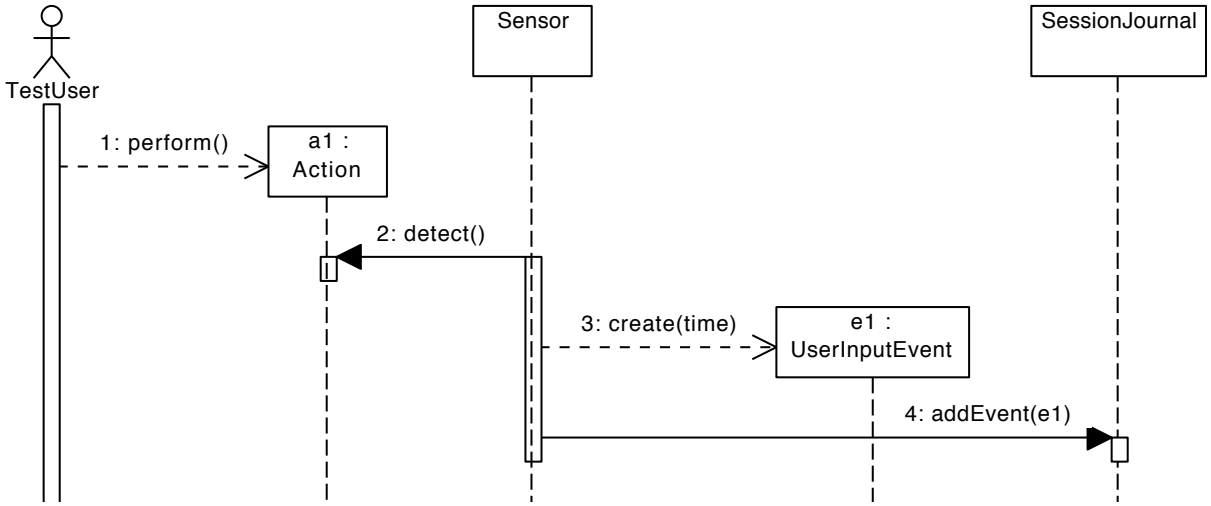


Figure 2.25: Event detection example (UML sequence diagram)

Capture phase

The framework captures usability data as `Events`. This means that detected usability data must be converted into `Event` objects; this is performed by the `Sensors`. At the start of an `EvaluationSession` the `SessionJournal` and the `Sensors` are initialized. As long as the session is running the `Sensors` observe `SensorTargets` for changes. If a change is detected, the `Sensor` creates a new `Event` and adds it to the `SessionJournal`.

Figure 2.25 shows, for example, how user input is captured by the framework as `Events`: first, the `TestUser` performs an action, such as touching the `MobileDevice`'s screen. This action is then detected by a `Sensor` that creates the appropriate event instance, in this case an `UserInputEvent`. The `Sensor` finally adds the event to the `SessionJournal` where it is stored to be accessed by the `Interpreters` during the analysis phase, which takes place after the `EvaluationSession` ends.

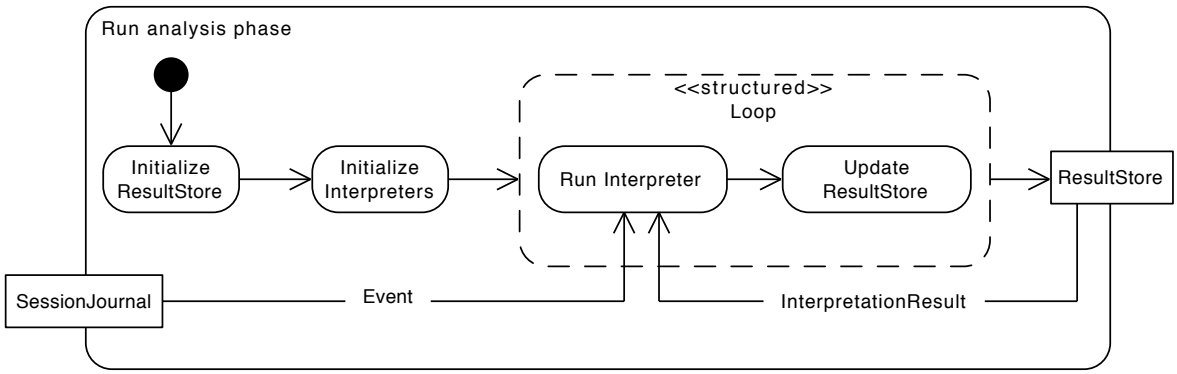


Figure 2.26: Analysis phase (UML activity diagram)

Analysis phase

During the analysis phase, the framework interprets the `Events` that were gathered in the previous capture phase. This interpretation is performed by several `Interpreters` that analyze the `Events` in the `SessionJournal` and produce `InterpretationResults` which are then stored in the `ResultStore`.

The analysis phase works as follows (Figure 2.26): First, a setup step is performed where the `ResultStore` and the `Interpreters` are initialized. After this, the framework selects an `Interpreter` and runs it to generate or update the `InterpretationResults` in the `ResultStore`. In multiple rounds of execution this step is repeated until the `InterpreterController` decides that the available `InterpretationResults` are satisfying. The framework then continues with the subsequent critique phase.

But what happens when an `Interpreter` runs? First, the interpreter checks if all of its dependencies are fulfilled. A dependency can be, for example, the existence of a required `InterpretationResult`. If all dependencies are fulfilled the interpreter continues, otherwise it waits to execute again in the next round. The interpreter now gathers all of its required resources, that is, `Events` from the `SessionJournal` and `InterpretationResults` from the `ResultStore`. After all of the "working materials" are available the interpreter decides if it can contribute to the common solution represented by the `ResultStore`. If it can contribute the interpreter updates an existing `InterpretationResult` or generates a new one. In either case, the `ResultStore` must then be updated. Because interpreters may execute multiple times, the interpreter must now decide whether it can contribute once more and should execute again in the next round or if it is finished. A finished interpreter does not execute again in the current analysis phase. Figure 2.27 shows the interpreter activity as a flow chart.

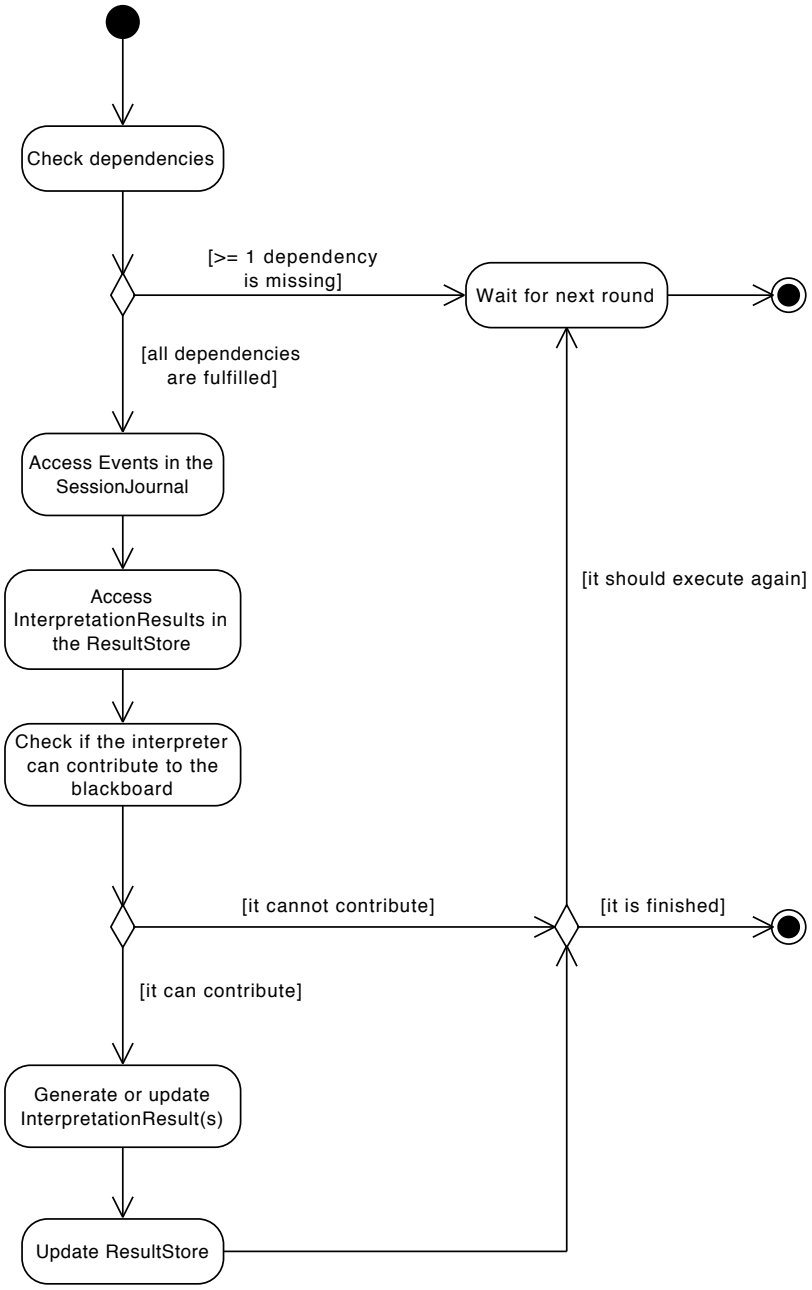


Figure 2.27: Interpreter execution (UML activity diagram)

Critique phase

Figure 2.28 contains an overview of the report generation flow of work. For each section in the report, the `ReportGenerator` accesses the `ReportConfiguration` to determine which section must be generated next.

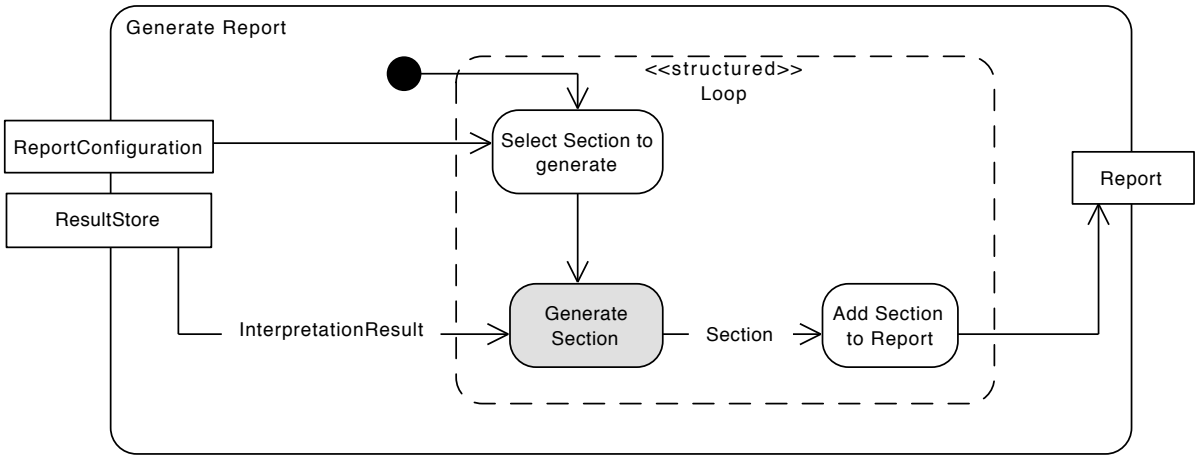


Figure 2.28: Report generation (UML activity diagram)

This `Section` is then constructed by the `SectionGenerator` using the interpretation results in the `ResultStore` (Figure 2.29 and Figure 2.22). The finished `Section` is added to the `Report` and if there are sections left, the section generation process is performed again. Once all `Sections` are processed, report generation is complete and the finished `Report` can be viewed by the `Developer`.

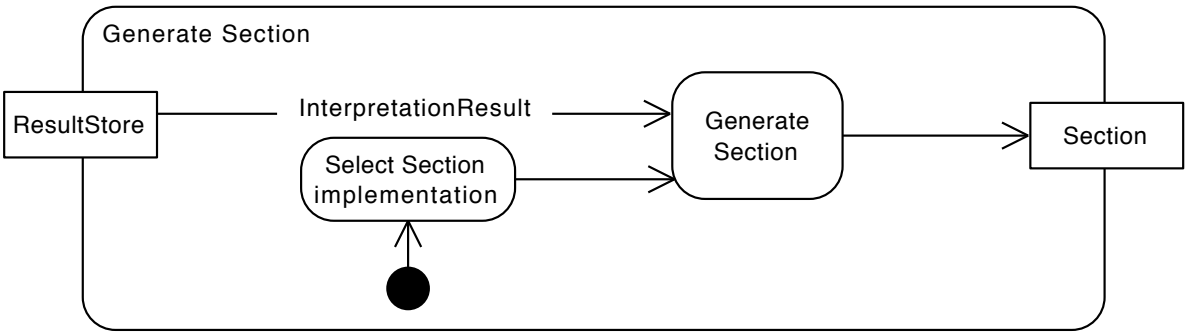


Figure 2.29: Section generation (UML activity diagram)

2.7 User interface

Capture phase

Session control To fulfill the `ControlSession` use case a user interface is needed that allows the `Developer` to carry out the following three tasks: First, to select the active sensors before the start of an evaluation session. Second, to start an evaluation session.

And third, to stop an evaluation session. The `PreviewEvents` use case also requires a form of live output of the events that are generated. Minimally this can be simply a textual log of the generated events as shown in Figure 2.31. A preliminary mockup for a more complex graphical user interface is depicted in Figure 2.32. Here, an application we call `Mobile Monitor` allows for browsing of existing sessions as well as the recording of a new session. The mockup also shows the live preview of events.

Critique phase

Report configuration The `ConfigureReport` use case requires a user interface for developers. Developers must be able to choose the output directory of reports and they must be able to toggle the inclusion of individual report sections. A preliminary user interface mockup is depicted in Figure 2.33. Here, the report configuration is a part of the `Mobile Monitor` application.

Report user interface Interactive `Reports` that are generated as HTML files, also require a user interface for the `Developer` to fulfill the `ViewReport` use case. A sample illustration of such an interface is displayed in Figure 2.30. In this figure, an interactive timeline of events makes it possible to display only a subset of all recorded events by allowing the developer to filter the stream of events by event type.

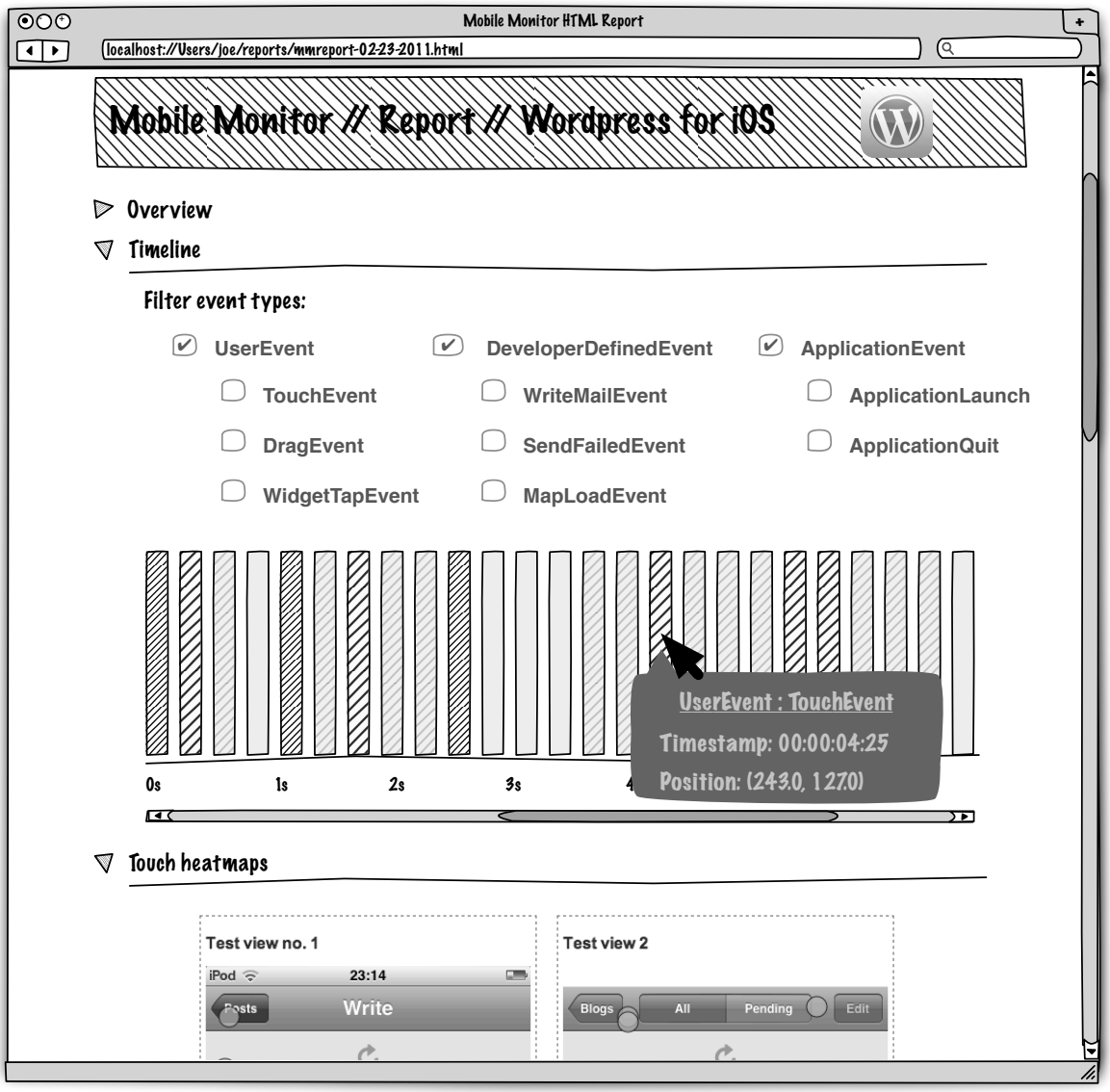


Figure 2.30: A finished interactive report (Mockup screenshot)

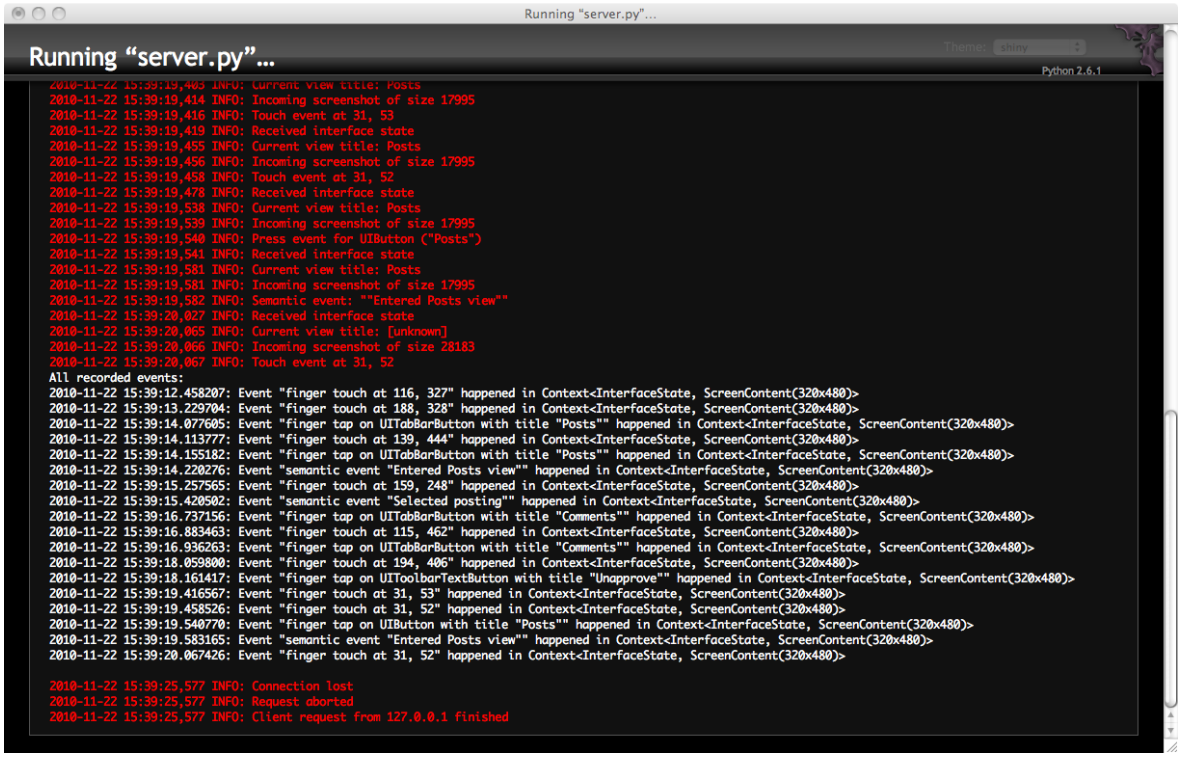


Figure 2.31: Event log user interface in an early version of the prototype (Screenshot)

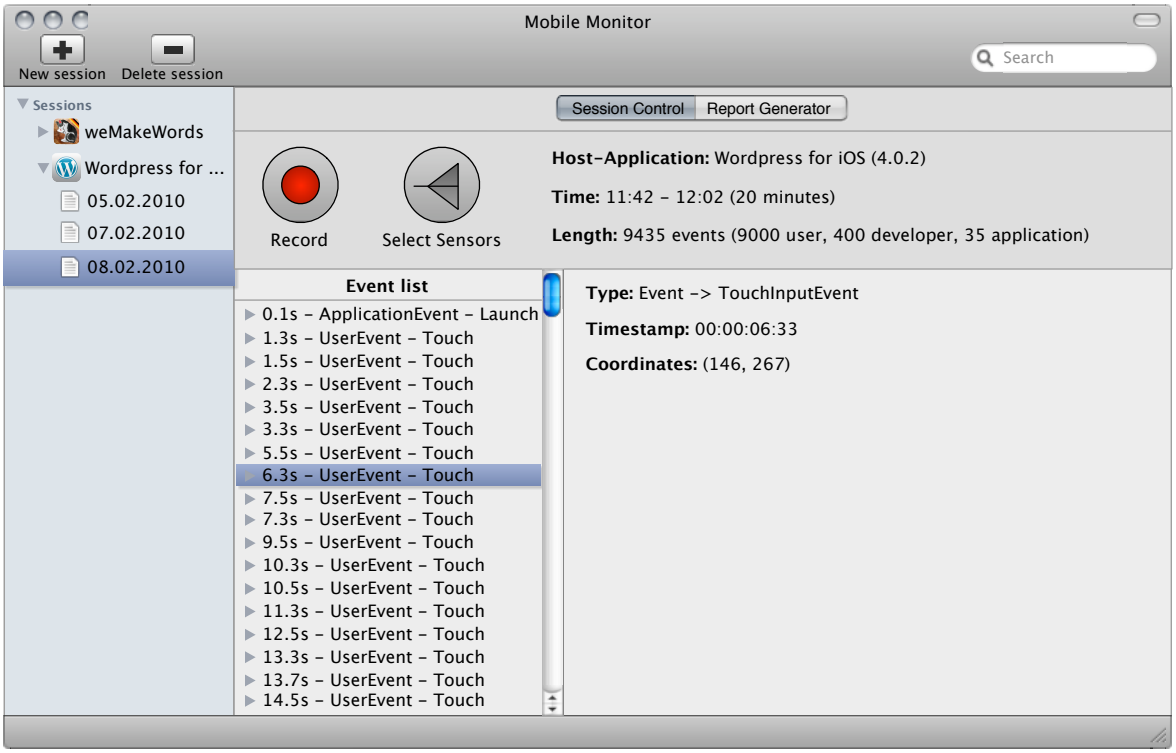


Figure 2.32: Session control user interface during the capture phase (Mockup screenshot)

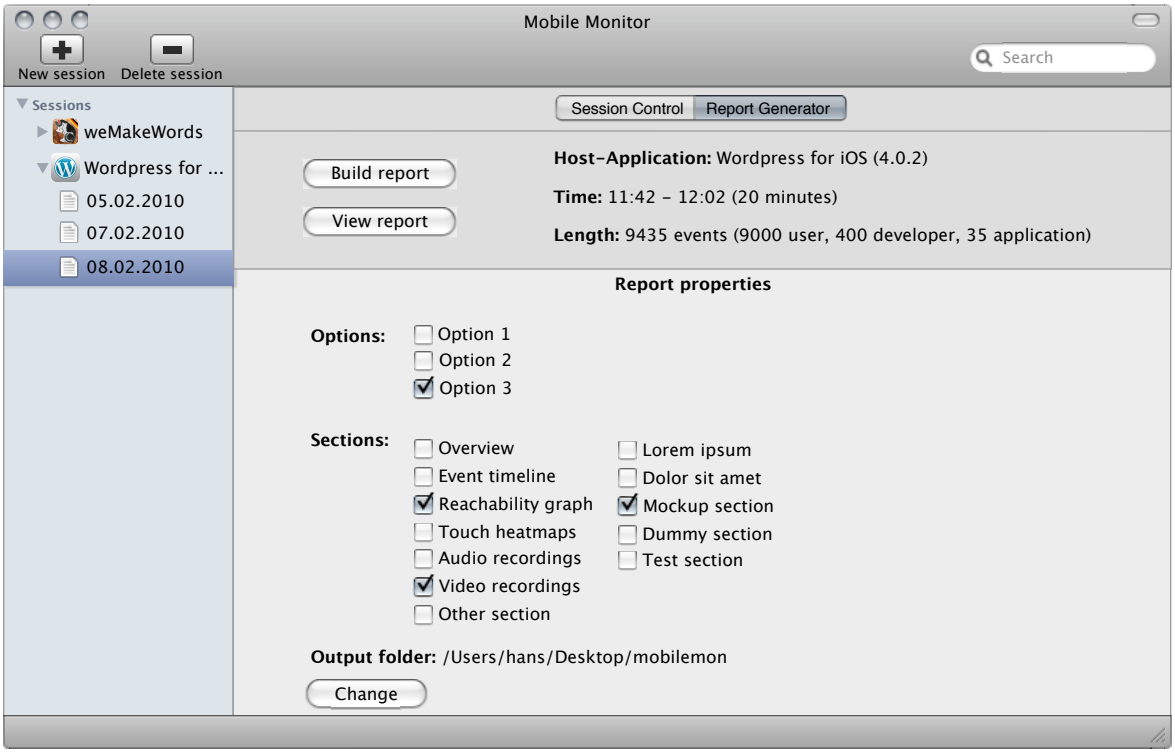


Figure 2.33: Report configuration user interface (Mockup screenshot)

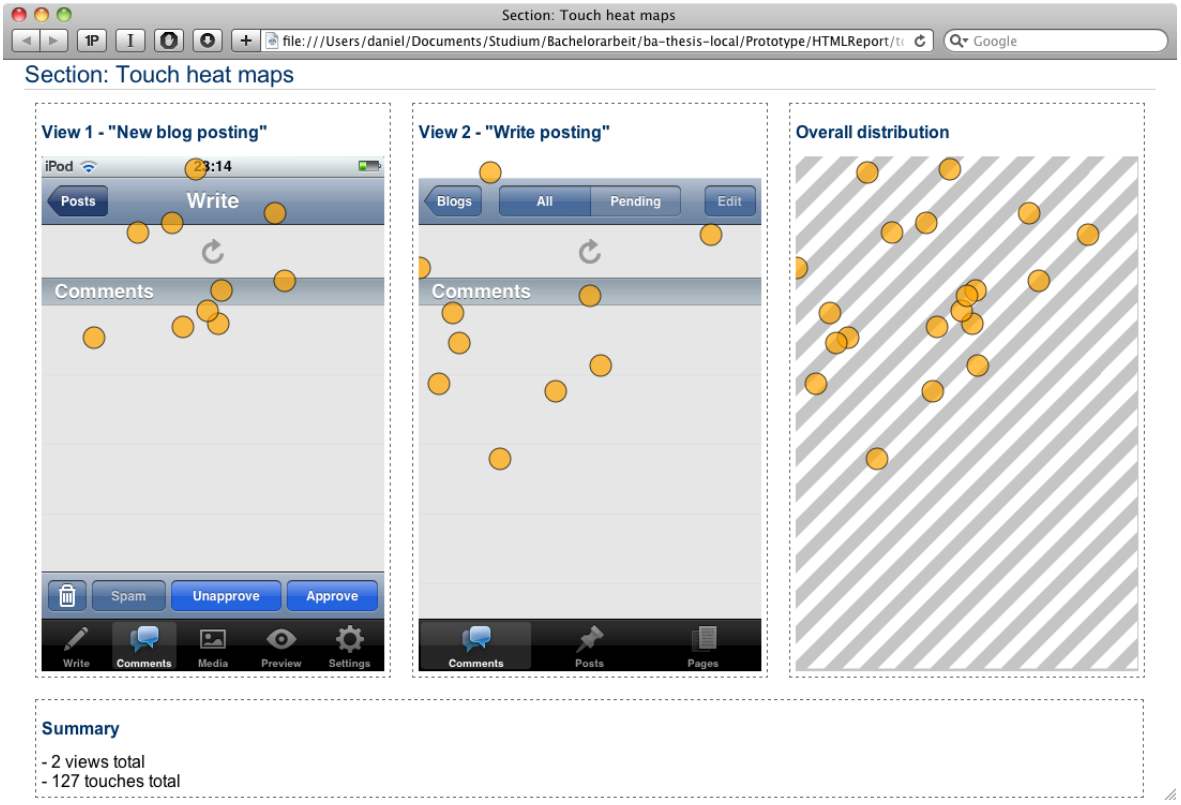


Figure 2.34: A touch heatmap section (Mockup screenshot)

3 System design

This chapter describes the transformation of the analysis model into a system design model. The system design model presents the preliminary architecture and the design goals that influenced it.

3.1 Design goals

Extensibility and maintenance

Because usability evaluation is such a dynamic field of work, the probability of users requesting changes to the framework is high. Therefore the `muEvaluationFramework` should be easy to extend with new capture, analysis, and critique abilities. This means that the framework must be easily extensible with new sensors (capture), interpreters (analysis), and report sections (critique). Another important extensibility and maintenance goal is the portability of the framework. Even though we focus our development on the Apple iOS [5] platform for mobile devices it should also be possible to port the framework to other platforms, such as Google Android [16] or Nokia Symbian [25].

3.2 Subsystem decomposition

In this section we introduce the initial decomposition of the `muEvaluationFramework` into subsystems and describe the responsibilities and boundaries of each subsystem. First, based on the functional requirements and the models shown in the previous chapter, the following main subsystems depicted in Figure 3.1 were identified: the `Capture`, the `Analysis` and the `Critique` subsystem.

The overall structure of a usability evaluation (as described in Section 1.4.3) leads to two dependencies between the subsystems: First, the analysis subsystem depends on the events collected by the capture subsystem. And second, the critique subsystem depends on the results of the analysis subsystem.

Because of these dependencies there is only one reasonable order of execution in which the capture phase executes first, then the analysis phase follows, and finally the critique phase executes. The subsystems that were identified are now presented in closer detail.

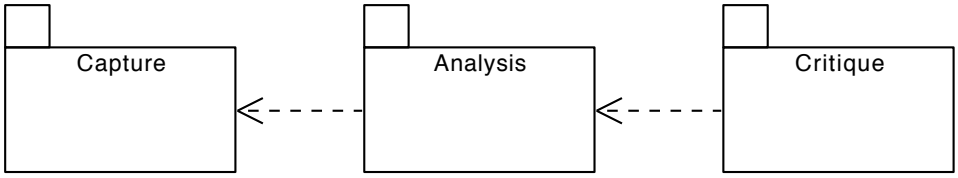


Figure 3.1: Relationship of the main subsystems (UML package diagram)

3.2.1 Proposed software architecture

The framework was divided into subsystems in order to separate the functionalities. This decomposition was performed with the principle of *low coupling and high coherence* in mind [12]. This means that an attempt was made to minimize dependencies between subsystems and to maximize cohesion between objects in a subsystem. In general, the architectural design of the `muEvaluationFramework` follows very closely the three phase model described in Sections 1.4.3 and 2.6.2.

To describe the used software architecture more closely we must distinguish between the overall architecture of the framework (i.e. the main subsystems) and the architectures of each individual subsystem.

Overall, the framework uses a closed software architecture consisting of three layers. Each layer is represented by one of the main subsystems, i.e. the `Capture`, the `Analysis` and the `Critique` subsystem. The architecture is *closed* because each layer (or phase) may only access the outputs of the previous layer.

At the level of the individual subsystems the framework is structured using two more architectural styles. First, the `Capture` subsystem uses a *client/server* architectural style where a client, the `CaptureLibrary`, transmits usability data to the `CaptureServer`. Second, the other two main subsystems (`Analysis` and `Critique`) follow a *repository* architectural style because they both contain a central data structure that is modified while the subsystem is active (the `ResultStore` and the `Report` respectively).

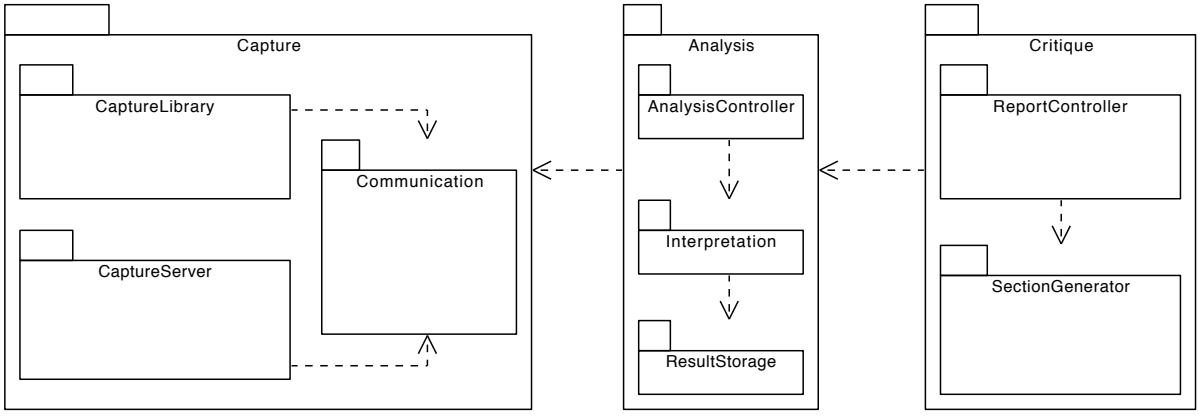


Figure 3.2: Overview of the main subsystems (UML package diagram)

3.2.2 Capture subsystem

The `Capture` subsystem is used during the capture phase of a usability evaluation. It collects usability data related to the host-application and stores them for later analysis. The capture subsystem consists of three subsystems (Figure 3.2 and Figure 3.3):

- `CaptureLibrary`: This subsystem is responsible for the collection of usability data during an evaluation session. It uses `Sensors` to collect the necessary data and encapsulates them in `Events`. The `CaptureLibrary` subsystem is the part of the `muEvaluationFramework` that is embedded into a host-application and must therefore be kept as small as possible. Hence, the `CaptureLibrary` subsystem does not store the collected data; it merely captures them and hands them over to the `CaptureServer` subsystem via the `Communication` subsystem. The newly identified `CaptureManager` object controls the subsystem.
- `CaptureServer`: This subsystem stores the `Events` that were collected by the `CaptureLibrary`. Thereby the events and their properties are made available for the `Analysis` subsystem. The `CaptureServer` subsystem is controlled by the newly identified `CaptureController` object and it uses the `Communication` subsystem to receive events from the `CaptureLibrary`.
- `Communication`: This subsystem facilitates the transmission of events from the `CaptureLibrary` to the `CaptureServer` subsystem.

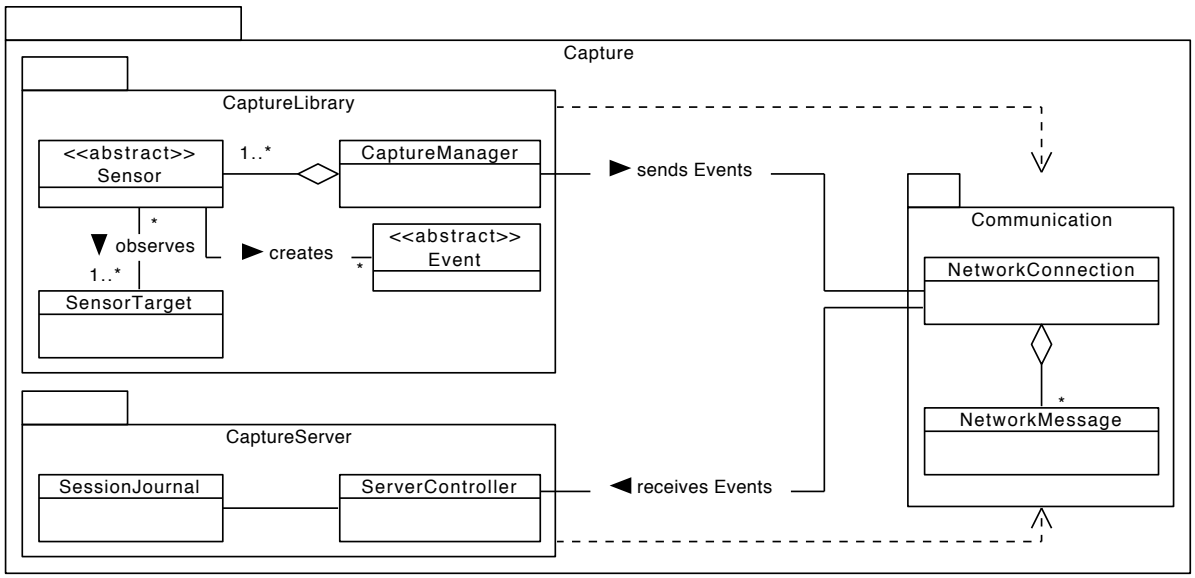


Figure 3.3: The `Capture` subsystem (UML package diagram)

Communication subsystem The `Communication` subsystem is used by the `CaptureLibrary` to transmit `Events` to the `CaptureServer`. Because we newly identified the functionality of this subsystem in the system design phase we must extend the analysis object model with new objects. The `Communication` subsystem consists of two objects, `NetworkConnection` and `NetworkMessage`. The new objects are depicted in Figure 3.4.

A `NetworkConnection` represents a link between the `CaptureLibrary` and the `CaptureServer` subsystems. It is used by both the `CaptureManager` and the `ServerController` for communication.

A `NetworkMessage` is used by the `NetworkConnection` to exchange information over the network. It represents the `Events` that are encoded for transport and are sent over the `NetworkConnection`. A `NetworkMessage` can also be housekeeping data to setup or tear down a connection.

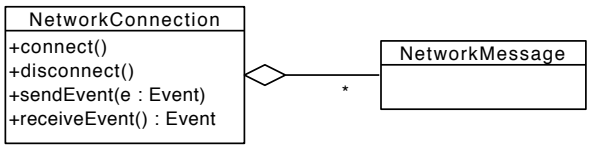


Figure 3.4: The newly identified objects for the `Communication` subsystem (UML class diagram)

3.2.3 Analysis subsystem

The `Analysis` subsystem contains the objects and subsystems which are needed during the analysis phase of a usability evaluation. As depicted in Figures 3.2 and 3.5, the subsystem consists of three subsystems:

- `AnalysisController` subsystem: This subsystem is responsible for setting up the working environment required by the interpreters and managing the interpreters themselves.
- `Interpretation` subsystem: Processes the events that were collected during the capture phase and infers knowledge about the usability properties of the analyzed application. Because it requires access to the `SessionJournal` it depends on the `Capture` subsystem.
- `Storage` subsystem: Stores the `InterpretationResult` objects produced by the `Interpretation` subsystem because they must be accessed by the `Critique` subsystem.

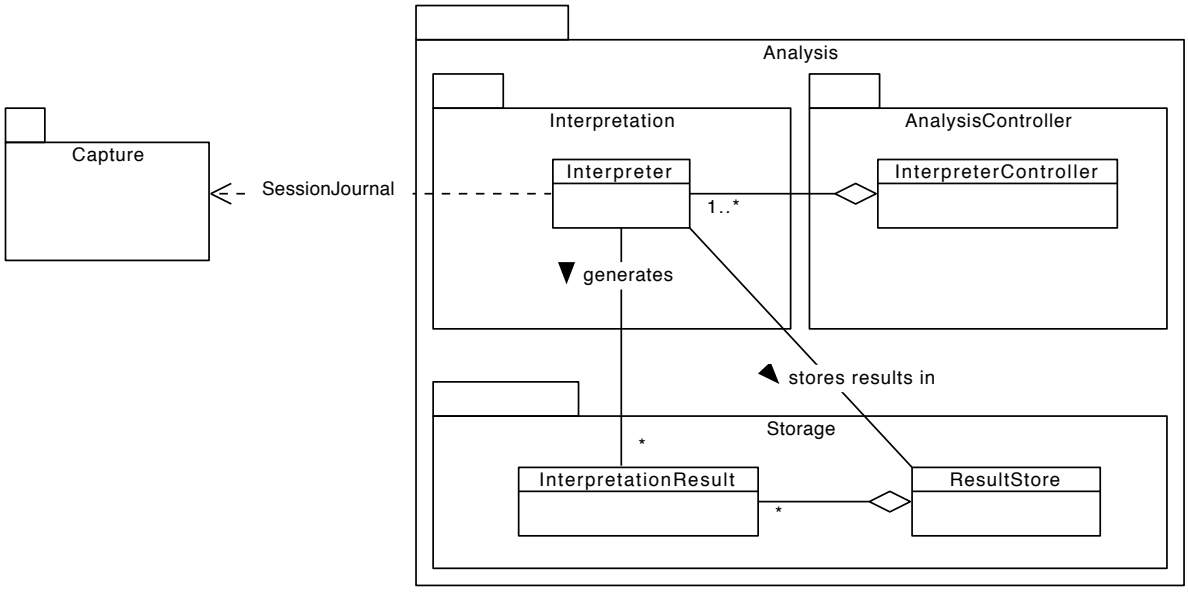


Figure 3.5: The `Analysis` subsystem (UML package diagram)

3.2.4 Critique subsystem

The `Critique` subsystem is used during the critique phase of a usability evaluation. It builds on the knowledge generated by the `Analysis` subsystem and its main purpose is to assemble a report document for the developer. As shown in Figure 3.2 and Figure 3.6, the subsystem consists of the following three subsystems:

- `ReportController` subsystem: This subsystem contains the `ReportGenerator` object which manages the overall structure of the report as well as the flow of work during the critique phase. The report generator controls the `Report` object and uses the `ReportConfiguration` provided by the developer to determine which sections must be created.
- `SectionGeneration` subsystem: Accesses the interpretation results stored by the `Analysis` subsystem and generates the individual `Sections` of the report.

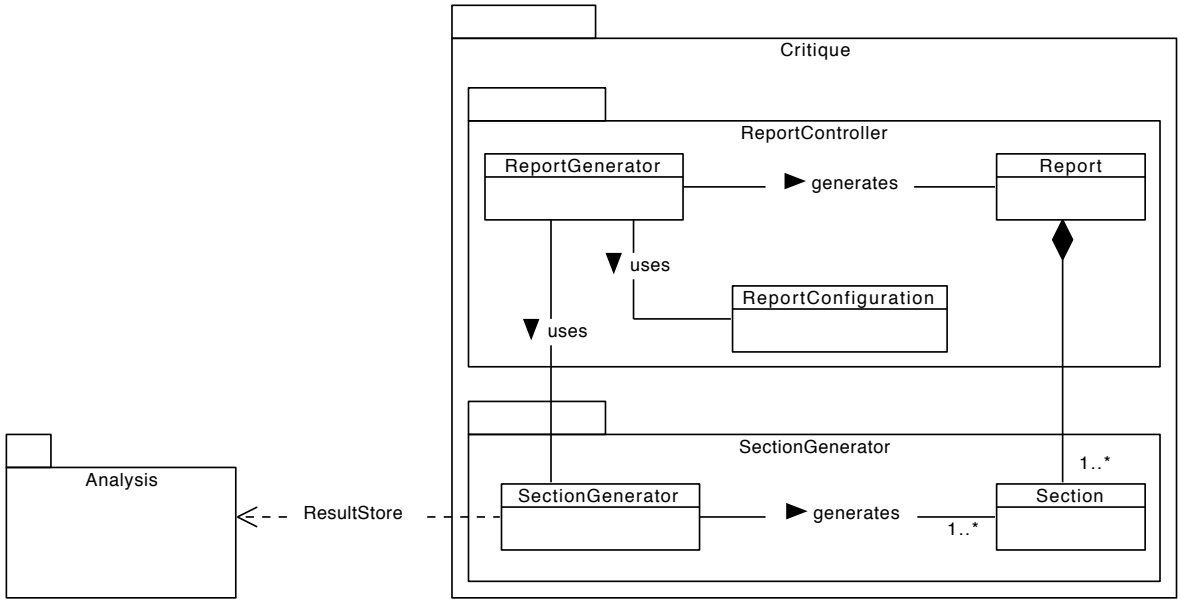


Figure 3.6: The `Critique` subsystem (UML package diagram)

3.3 Hardware/software mapping

The following section describes the relationship between runtime components and hardware nodes in the `muEvaluationFramework`.

Because the framework is used to monitor applications on mobile devices at least parts of it must be deployed on the mobile device. Therefore we decided to manifest the `CaptureLibrary` subsystem of the `Capture` subsystem in a separate library called `libCapture.a` which the host-application is linked against at compile time.

The `CaptureServer` subsystem of the `Capture` subsystem is manifested in the `CaptureServer.py` artifact. Also, the capture server does not run on the mobile device. Therefore, the capture library and the capture server must communicate over a network connection.

The `Analysis` and `Critique` subsystems of the framework are executed on a more power-

ful desktop computer for reasons of performance and data storage: the framework records hundreds or thousands of events each minute, some of which are heavily augmented with complex metadata such as screenshots or video frames. Both parts of the framework are manifested in the `Mobile Monitor.app` application artifact. Splitting up the framework among these two devices means that a way for communication between the nodes is required. To allow the end user to use the device in a natural way during an evaluation session, i.e. without any additional wires attached to the device, the network communication should work across a wireless connection. Figure 3.7 shows how the framework will be deployed and how it is separated into components on the two devices.

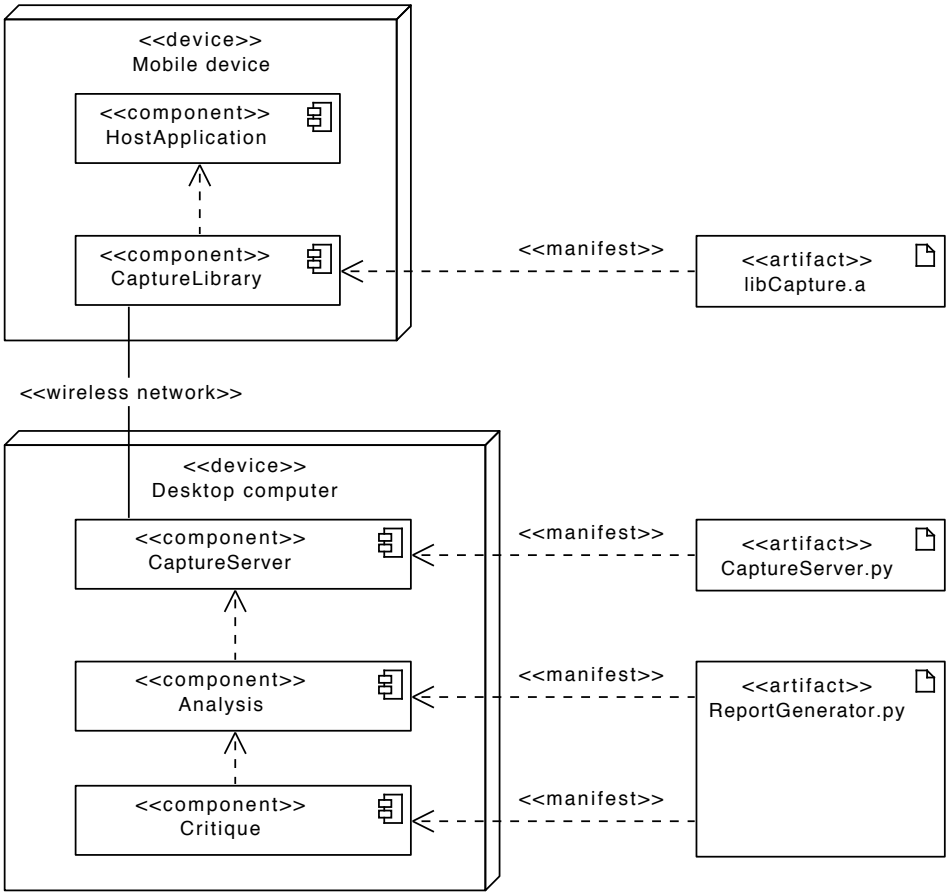


Figure 3.7: Deployment of the framework and its separation into components (UML deployment diagram)

3.4 Persistent data management

This section gives a summary of the persistent data objects used by the framework and describes how persistent data are stored. There are four types of data in the `muEvaluationFramework` that are candidates for persistent storage. In the following we will give a short overview for each data type.

Events

`Events` are produced by the `Capture` subsystem during the capture phase of an `EvaluationSession`. Developers might want to create multiple `Reports` from a single sequence of captured events to gain different insights. The framework must therefore store `Events` persistently after the capture phase.

InterpretationResults

The `InterpretationResult` objects generated by the `Analysis` subsystem are required for creating a `Report`. Because the critique phase is always performed directly after the analysis phase, the `InterpretationResults` do not have to be stored persistently from session to session. Still, the framework must be able to handle many of them during runtime.

Report configurations

A developer is likely to have several versions of a `Report` generated when analyzing the usability of a host-application. Each iteration of the `Report` is probably quite similar in the selected configuration options. It is also very likely that a developer who uses the framework many times comes up with a personalized set of frequently used report configurations. Therefore it seems sensible to store the `ReportConfiguration`, i.e. the order of the included sections as well as each section's configuration options, for reasons of convenience.

Report presentations

A `Report` essentially bundles and all the information contained in the generated `InterpretationResults`. Therefore the `Report` is the final and most important artifact produced by the framework during each `EvaluationSession` and must be stored persistently.

4 Object design

We divided the object model of the `muEvaluationFramework` into subsystem in the previous chapter. In this section, we go through the individual subsystem to refine the model where necessary and to add more detailed methods and attributes to each class. The object design presented here is structured according to the subsystem decomposition presented in Section 3.2.

4.1 Interface documentation guidelines

The following coding conventions are used as guidelines:

- Code developed in the Objective-C language should follow the coding guidelines provided by Apple Inc. [2]
- Code developed in the Python language should follow the style guide provided by the Python Software Foundation [30]

As a result of the coding guidelines, classnames for classes defined in the Objective-C language are prefixed with "`MF`" (for `muEvaluationFramework`). Prefixes like this are customary in Objective-C to avoid naming conflicts.

4.2 Subsystems

This section describes each of the three main subsystems of the framework in closer detail. Again, the description of the subsystems follows the overall *capture, analysis, and critique* structure of the framework. Therefore, we present the subsystems in the same order:

1. The `Capture` subsystem in Section 4.2.1
2. The `Analysis` subsystem in Section 4.2.2
3. The `Critique` subsystem in Section 4.2.3

4.2.1 Capture subsystem

The `Capture` subsystem includes the three subsystems `CaptureLibrary`, `CaptureServer`, and `Communication`. The `CaptureLibrary` is responsible for the actual collection of usability data. Whereas the `CaptureServer` stores the captured data as events to make them available to the subsequent analysis phase. Events are transmitted from the `CaptureLibrary` to the `CaptureServer` via a network connection that is controlled by the `Communication` subsystem.

CaptureLibrary subsystem The `CaptureLibrary` is the part of the framework that is deployed on a mobile device. It uses software-based `Sensors` to collect usability data in the form of `Events` that are then transmitted to the `CaptureServer` using the `Communication` subsystem.

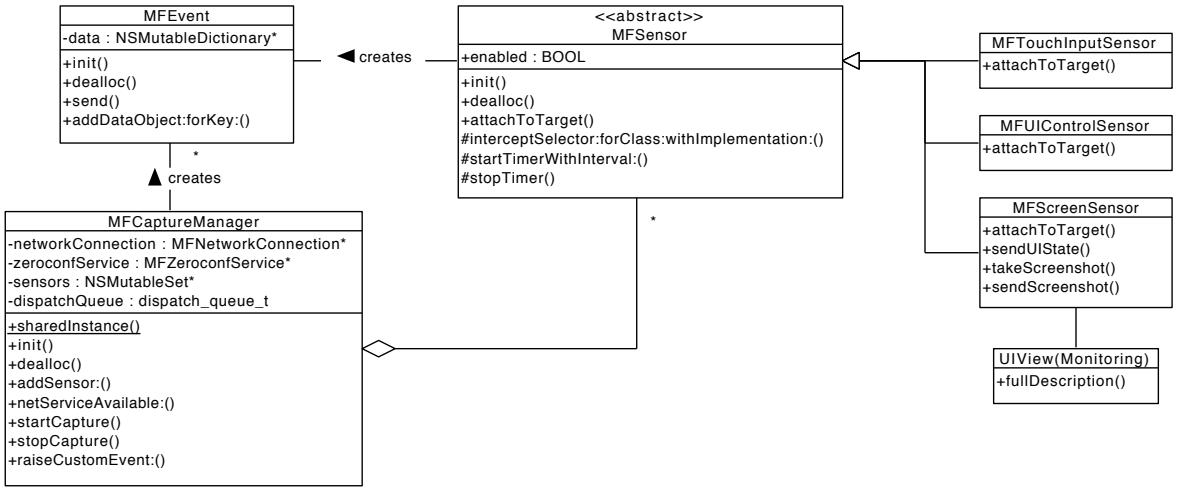


Figure 4.1: Object design for the `CaptureLibrary` subsystem

As said before, the names of the objects in the `CaptureLibrary` subsystem are prefixed with the letters "`MF`" to prevent naming conflicts. The following objects are the most important objects of the subsystem (Figure 4.1):

- `MFCaptureManager`: This object controls the subsystem. It is also the only object that a developer must interact with when deploying the `CaptureLibrary` and therefore it also serves as a facade for the subsystem. It uses the Singleton pattern to be easily accessible from within the host-application without requiring too many source code changes.
- `MFEvent`: This object represents a part of the usability data that are collected by the `CaptureLibrary`. It is implemented as an associative array (or hash map) by

using the `NSMutableDictionary` class provided by the iOS runtime. Usability data are thus described by a number of key-value pairs stored in the `data` attribute of a `MFEvent`.

- `MFSensor`: This object serves as an abstract base class for all concrete implementations of a sensor. It provides methods for recurring tasks that are needed by all sensors, such as method interception (`interceptSelector:forClass:withImplementation:`) or timer handling (`startTimerWithInterval:` and `stopTimer`).

Now we describe the most interesting aspects of the `CaptureLibrary` subsystem. We start by explaining how the `CaptureLibrary` is deployed to enable event capturing in an existing host-application. Then we describe how the `CaptureLibrary` uses sensors to collect usability data such as the state of the user interface or touch input events.

CaptureLibrary deployment It is a goal of the `CaptureLibrary` to make it easy to extend an existing host-application with event capture abilities. Therefore the functionality of the capture library is encapsulated by a single manager class (`CFCaptureManager`) that serves as a facade and hides the complexity from the developer. To make this manager class very simple to use it is implemented using the *Singleton pattern* [13]. This means that only one instance of the manager exists and thus it can be accessed very easily by calling the class method `[MFCaptureManager sharedInstance]`. The singleton pattern makes the manager class comfortably accessible from all locations in the host-application's source code, that is, the developer does not have to think about providing access to the manager object or allocating it. Listing 4.1 shows how the capture library is initialized.

Listing 4.1: `CaptureLibrary` **initialization**

```
// The developer must only include a single header file
// and link the application against the CaptureLibrary
#import "MFCaptureManager.h"

@implementation PlainNoteAppDelegate

- (BOOL)application:(UIApplication *)application
    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    // This call automatically finds a CaptureServer, connects to it,
    // and initializes all the sensors:
    [[MFCaptureManager sharedInstance] startCapture];

    // Standard setup code below:
    [window addSubview:[navigationController view]];
    [window makeKeyAndVisible];
    return YES;
}
```

Another way to improve the usability of the `CaptureLibrary` is the automatic setup of the network connection between the `CaptureLibrary` and the `CaptureServer`. The connection is established automatically and requires no setup by the developer. This is made possible by a technology called *Zero configuration networking*[1] (*Zeroconf*). Zeroconf provides support for automatic service discovery without the need for configuring IP addresses and ports by hand. The `CaptureServer` announces its service with a string identifier (`"_muEvaluationFramework-capture._tcp."`) and the `CaptureLibrary` can search the network for available servers and then automatically connect to one.

Determining the UI state To determine the state of an application's user interface we must find the root view of an application and then iterate through all of its sub-views, collecting data as we descend the view hierarchy. The root view of an iOS application is normally an instance of the `UIWindow` class. One approach is to create a subclass of `UIWindow` that adds a custom method `getUIState` which returns a description of the user interface's structure. Unfortunately, we cannot subclass the `UIWindow` class because existing code in the host-application will not automatically use the subclass. Therefore we have to extend `UIWindow` directly. Luckily, Objective-C provides a way to do this with a language feature called *Categories* [13]. We use the *Category pattern* to add a method `-(NSDictionary*)getUIState` to every `UIWindow` instance in the host-application. This method returns a `NSDictionary` object with the classname, dimensions and title of every UI element in the view hierarchy. Listing 4.2 shows the console printout of an example view hierarchy.

Listing 4.2: Example view hierarchy printout

```
{
    address = 97057632;
    className = UIWindow;
    frame =     {
        height = 480;
        width = 320;
        x = 0;
        y = 0;
    };
    subviews = (
            {
            address = 97114864;
            className = UITransitionView;
            frame = {
                (...)
            };
            subviews = ( ... )
    );
};
```

[1] Apple's implementation of Zero configuration networking is called *Bonjour*. Additional information about Zeroconf and Bonjour can be found at `www.zeroconf.org` [14]

Method interception On the iOS platform, applications use a toolkit for graphical user interfaces called `UIKit`. `UIKit` provides a set of objects for interaction handling and to define common user interface widgets such as buttons (`UIButton`) or scrollable lists (`UITableView`). Because these objects manage the interesting information about user interactions and the structure of the user interface, the `muEvaluationFramework` needs a way to access this information. This access is possible by modifying the internals of UIKit objects via the built-in reflection mechanisms of the Objective-C language. By extending or replacing the implementations of certain methods with our own code, we can intercept them and re-route the information flow if necessary. This procedure is best explained with a concrete example. Let us assume that we want to capture all touch events that the user performs while the host-application is active. By reading the iOS documentation we determine that the UIWindow class is our ideal target for intercepting touch events: It is the root view in the hierarchy and therefore all touch events must pass through it. We also find out that `UIWindow` has a method called `sendEvent:(UIEvent*)event` that dispatches user interface events to the correct places - which is exactly what we want to spy on. Now we need a way to modify this method to give us access to the `UIEvents`. We do this by changing a pointer in `UIWindow`'s method list (`struct objc_method_list`) that normally points at the original implementation of the `sendEvent:` method (Figure 4.2).

We overwrite the method pointer with the address of our own implementation of `sendEvent:` that forwards any `UIEvents`, which are related to touch input, to the `CaptureLibrary` (Figure 4.3).

To make sure that the behavior of the `UIWindow` object stays the same, our method implementation also executes the original `sendEvent:` method. This modification of runtime objects is possible to achieve with built-in functions of the Objective-C runtime [4]. In the prototypical implementation of the `CaptureLibrary` this method interception technique is performed using the *MAObjCRuntime* library [7] that provides a lightweight object oriented wrapper around the Objective-C runtime interface provided by Apple. Example code is shown in Listing 4.3.

Now that we can examine all the `UIEvents` that pass through `UIWindow`'s `sendEvent:` method, we can filter the event stream for interesting events, for example only those that have an event type of `UIEventTypeTouches`, and store them for later analysis.

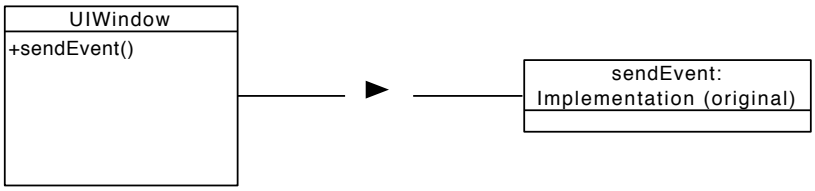


Figure 4.2: `UIWindow sendEvent:` behavior before method interception is performed

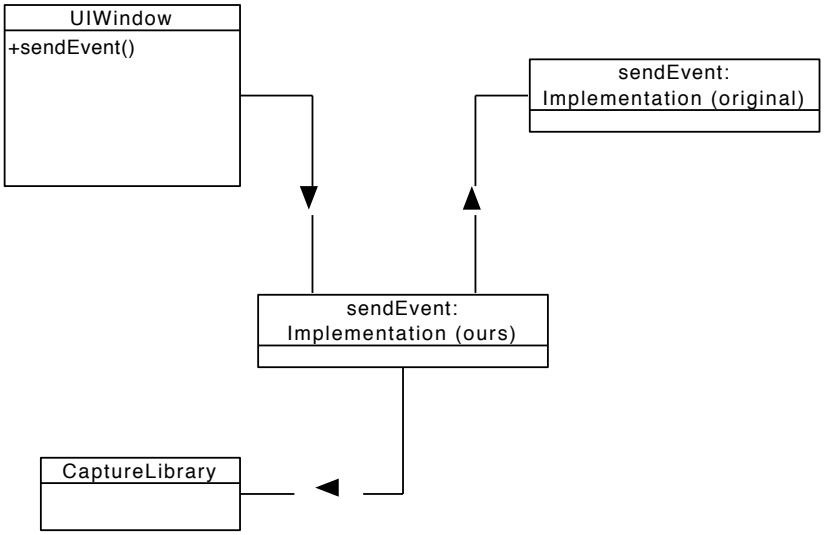


Figure 4.3: `UIWindow sendEvent:` behavior after method interception is performed

Listing 4.3: Method interception example code

```
// A pointer to the original implementation of the sendEvent: method.
IMP gOriginalSendEventImp = NULL;

// Our own version of [UIWindow sendEvent:]
static void HijackedUIWindowSendEvent(id self, SEL _cmd, UIEvent *event) {
    gOriginalSendEventImp(self, _cmd, event);
  /* Check for interesting events here. */
}

// Overwrites the implementation of a method for a given class with a custom pointer
- (IMP)interceptSelector:(SEL)selector
                forClass:(Class) aClass
      withImplementation:(IMP) anImplementation
{
    RTMethod *originalMethod = [aClass rt_methodForSelector:selector];
    IMP originalImp = [originalMethod implementation];
    [originalMethod setImplementation: anImplementation];
    NSLog(@"MFSensor - hijack: [%@ %@] rerouted from 0x%x to 0x%x",
        aClass, NSStringFromSelector(selector), originalImp, anImplementation);
    return originalImp;
}

// Enables the sensor by overwriting the sendEvent: method in
// the UIWindow object
- (void)attachToTarget {
    gOriginalSendEventImp = [self hijackSelector:@selector(sendEvent:)
                                        forClass:[UIWindow class]
                              withImplementation:(IMP)HijackedUIWindowSendEvent];
}
```

Network representation for Events `Events` are transmitted using a simple XML-based protocol that stores them as a set of keys and values and also allows nesting data to preserve a hierarchical structure (e.g. when transmitting UI states). Listing 4.4 shows the XML representation of an example `Event`.

Listing 4.4: A `UserInputEvent` represented as XML

```
<event>
  <key>class</key>
  <string>UserInputEvent</string>
  <key>timestamp</key>
  <date>2011-02-19T22:43:06Z</date>
  <key>x</key>
  <integer>155</integer>
  <key>y</key>
  <integer>267</integer>
</event>
```

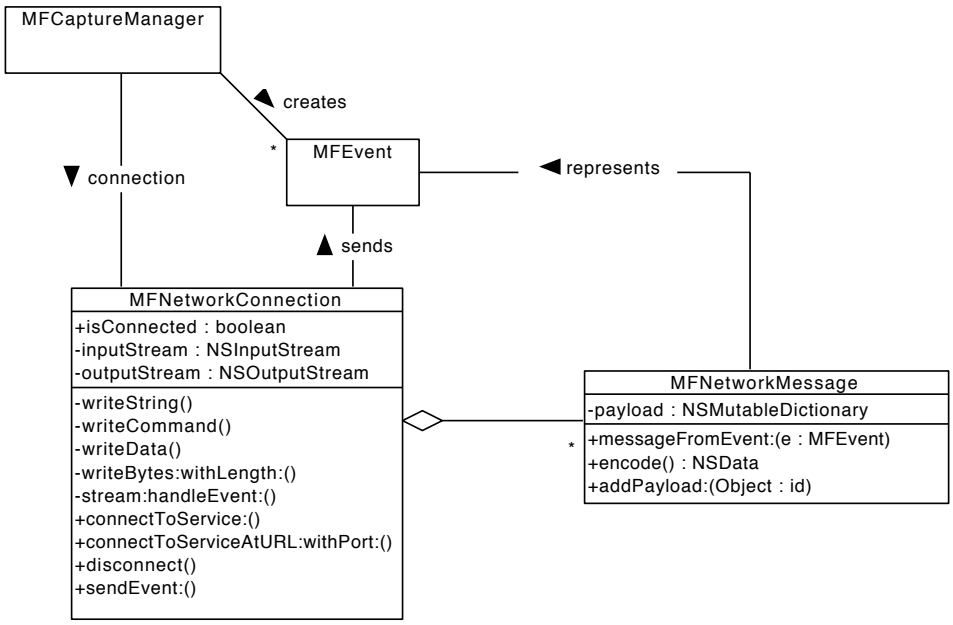


Figure 4.4: The `Communication` subsystem implementation for the `CaptureLibrary` (UML class diagram)

Communication subsystem The implementation of the `Communication` subsystem is split into two parts: One part for the `CaptureLibrary` subsystem (depicted in Figure 4.4), and one part for the `CaptureServer` subsystem (depicted in Figure 4.4)

Because `Events` must only be transmitted from the `CaptureLibrary` to the `CaptureServer`, the `Communication` subsystem represents a *unidirectional pipe*. That is, the part of the subsystem that is used by the `CaptureLibrary` sends `Events` and the part used by the `CaptureServer` receives `Events`.

CaptureServer subsystem The `CaptureServer` uses the `Communication` subsystem to receive `Events` from the `CaptureLibrary`. Therefore the `CaptureServer` can be seen as the counterpart of the `CaptureLibrary`. Figure 4.6 shows the objects of the `CaptureServer` subsystem:

- `ServerController`: Provides functionality to initialize and shut down the `CaptureServer`. Moreover it also controls the network connection provided by the `Communication` subsystem. Therefore it is the controlling class for the `CaptureServer`.
- `Event`: Like the `MFEvent` class of the `CaptureLibrary` the `Event` class of the `CaptureServer` is also implemented as an associative array (or hash map). Contrary to `MFEvent` it uses Python's built-in `dict` object to represent the associative array.

- `SessionJournal`: Stores the `Events` received from the `CaptureLibrary`. The `SessionJournal` can return all `Events` of a certain class with the `get_events_of_class()` operation. The `Analysis` subsystem uses this functionality to access the `Events` in the `SessionJournal`.

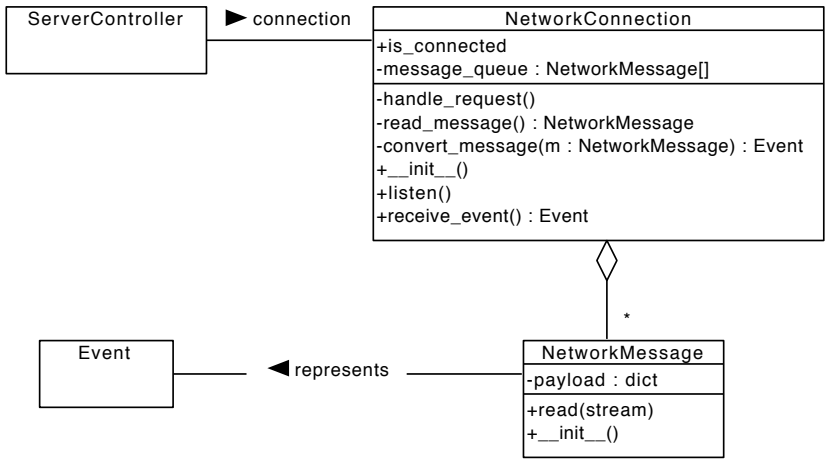


Figure 4.5: The `Communication` subsystem implementation for the `CaptureServer` (UML class diagram)

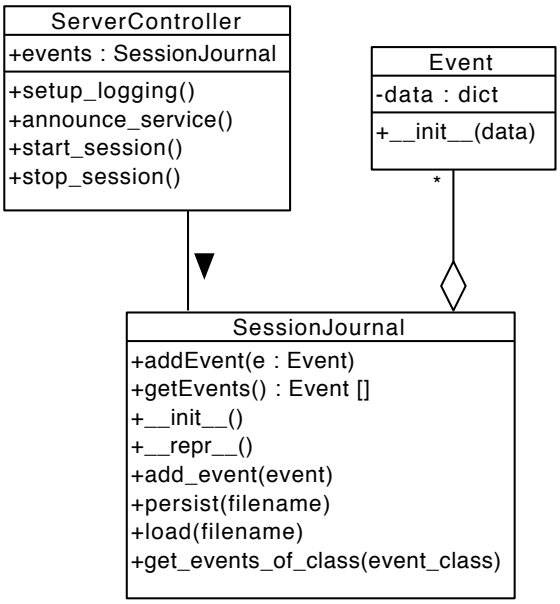


Figure 4.6: Object design for the `CaptureServer` subsystem

4.2.2 Analysis subsystem

The `Analysis` subsystem consists of the three subsystems `AnalysisController`, `Storage` and `Interpretation`. We now show the object design for each of the subsystems. After that we take a closer look at how the `Analysis` subsystem uses the *Blackboard pattern* to analyze the `Events` that were recorded by the `Capture` subsystem.

AnalysisController subsystem The `AnalysisController` subsystem consists of the `InterpreterController` and the `AnalysisFacade` objects. The former controls how the `Interpreters` work together on a shared blackboard called the `ResultStore` (this is described in closer detail later in the following paragraphs). The latter presents a clean interface to the `Analysis` subsystem.

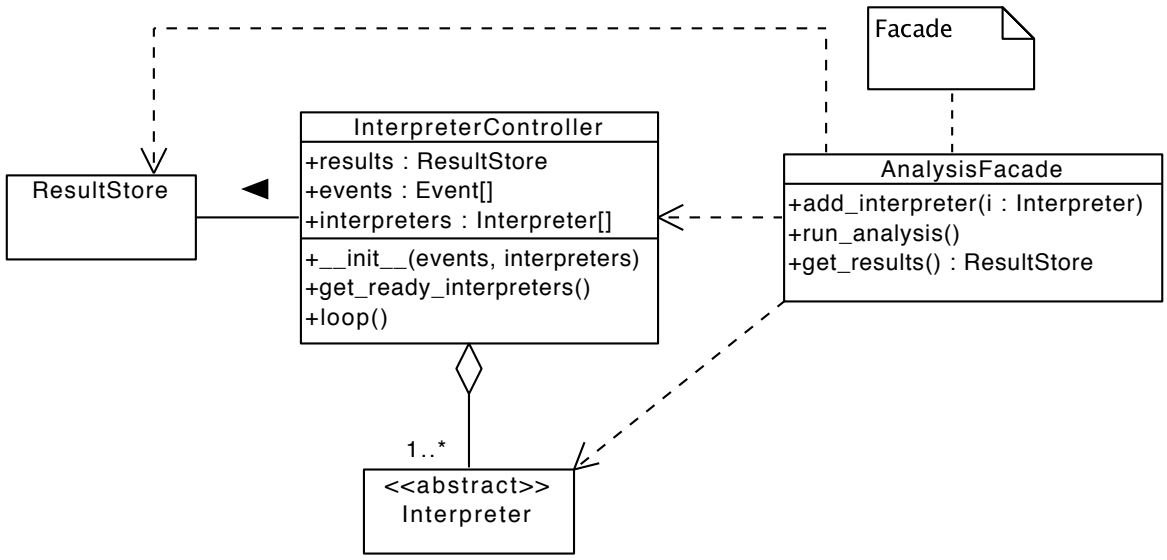


Figure 4.7: Objects of the `AnalysisController` subsystem (UML class diagram)

Storage subsystem This subsystem stores the `InterpretationResults` that are generated by the `Interpreters`. `InterpretationResults` store their payload similarly to the `Events` of the `Capture` subsystem by using Python's built-in `dict` object (an associative array implementation).

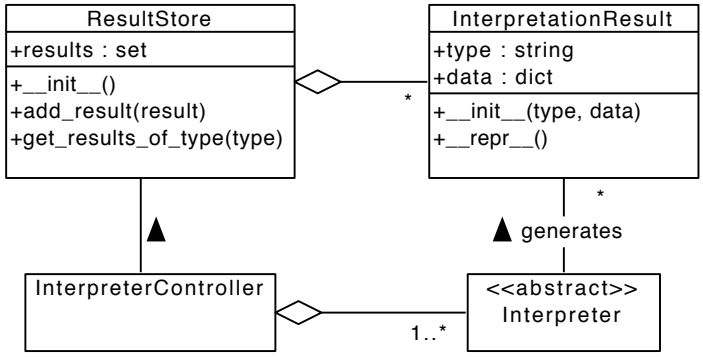


Figure 4.8: Objects of the `Storage` subsystem (UML class diagram)

Interpretation subsystem The `Interpretation` subsystem consists mainly of the abstract base-class `Interpreter` and a number of concrete `Interpreters`. In the following paragraphs we further describe how multiple `Interpreters` work together to produce `InterpretationResults`.

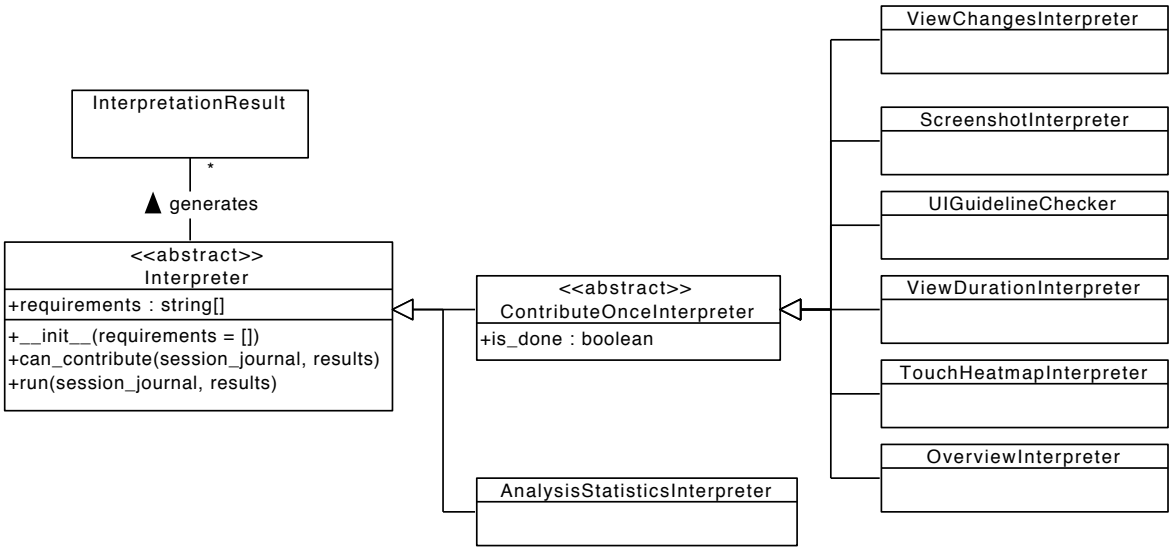


Figure 4.9: Objects of the `Interpretation` subsystem (UML class diagram)

Interpreter blackboard The interpreters of the analysis phase use the `ResultStore` as a shared blackboard that holds a common state in the form of `InterpretationResult` objects. The roles of the `Interpreter`, `ResultStore` and `InterpreterController` in the blackboard architecture for the analysis phase are shown in Figure 4.10.

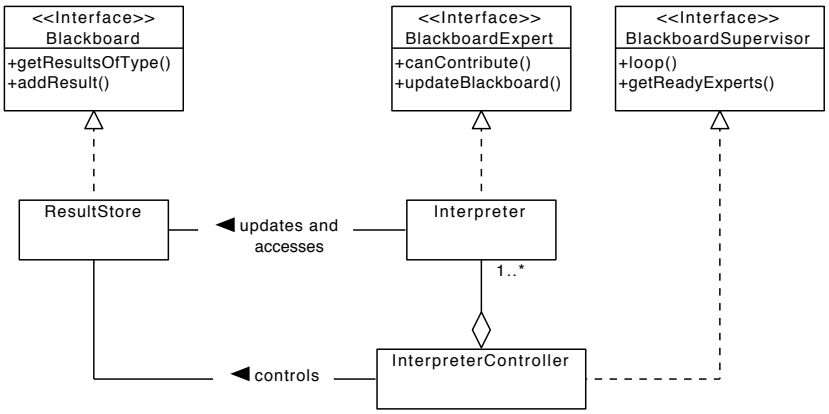


Figure 4.10: `Interpreters` as experts on a blackboard (UML class diagram)

Interpreters can add new results to the `ResultStore` by calling the `ResultStore.add_result()` method. Existing results are accessed with the `ResultStore.get_results_of_type()` method.

Interpreter order of execution `Interpreters` can depend on the results of other `Interpreters` to do their work. These dependencies must be resolved by determining an order of execution that allows `Interpreters` to run only when all of their requirements are satisfied. The requirements are stored in the `requirements` attribute of each interpreter as a list of strings that represent the classnames of the required `InterpretationResults`. The `InterpreterController` object determines the order of execution by calling each interpreter's `can_contribute()` method and then executes the interpreters that returned the boolean value `True`. Listing 4.5 shows how an individual `Interpreter` determines if its requirements are satisfied and Listing 4.6 shows how the `InterpreterController` determines the order of execution of all `Interpreters`.

Listing 4.5: Interpreter order of execution (`Interpreter`)

```
class Interpreter(object):
    #...

    def can_contribute(self, session_journal, results):
        for req in self.requirements:
            if not results.get_results_of_type(req):
                return False
        return True
```

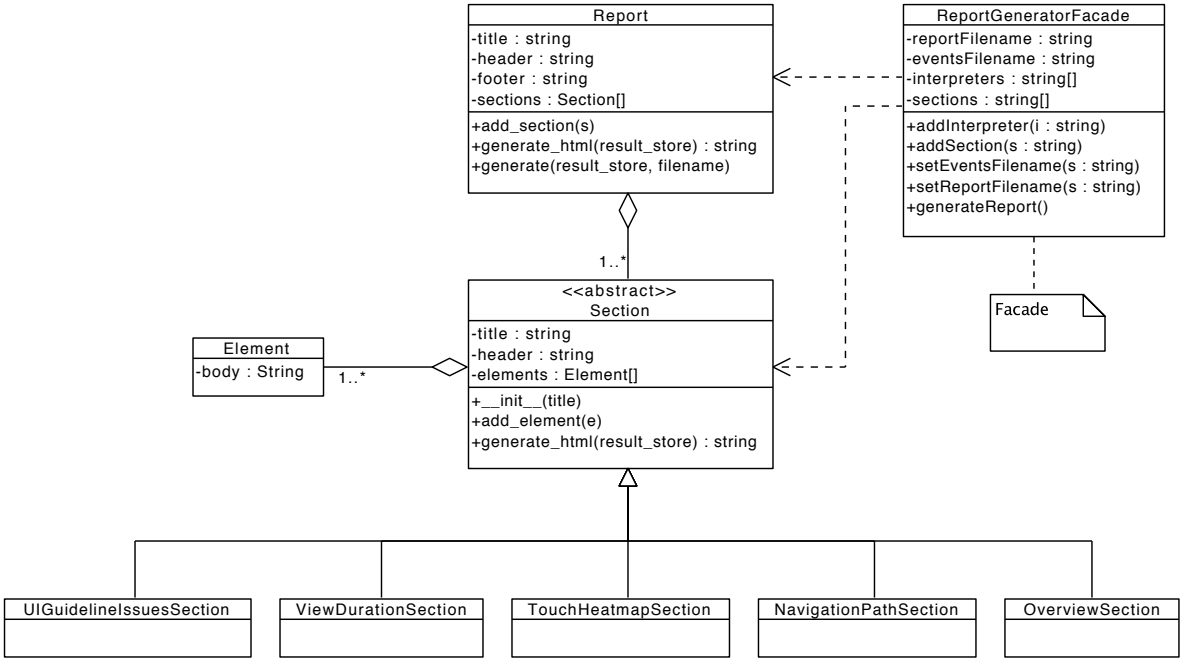
Listing 4.6: Interpreter order of execution (`InterpreterController`)

```
class InterpreterController(object):
    #...

    def get_ready_interpreters(self):
        result = []
        for interp in self.interpreters:
            if interp.can_contribute(self.events, self.results):
                result.append(interp)
        return result

    def loop(self):
        while True:
            ready_interpreters = self.get_ready_interpreters()
            if ready_interpreters == []:
                break # end the analysis phase
            for interp in ready_interpreters:
                interp.run(self.events, self.results)
```

We found that it is often required that an `Interpreter` only executes once during the analysis phase. A `ViewChangesInterpreter`, for example, only aggregates all of the view change events in the `SessionJournal` and does not depend on other `Interpreters`' results. Whereas an `AnalysisStatisticsInterpreter` must run multiple times to update the statistics during the analysis phase. Because we found that `Interpreters` are only executed once in many cases, we added the `ContributeOnceInterpreter` subclass that makes it simple to implement such a behavior.

4.2.3 Critique subsystem

The `Critique` subsystem creates a HTML document that summarizes the insights from the analysis phase. This HTML document is called a `Report`. During the object design some modifications were made to the subsystem. Now, the individual `Sections` are not produced by a `SectionGenerator` object but instead exist as subclasses of the abstract class `Section`. Therefore, the content of a section and the programming logic that creates the content are both represented by the specializations of the class `Section`. Furthermore, the `ReportGenerator` was changed to represent a facade for the whole subsystem (`ReportGeneratorFacade`). We believe that these changes simplify the `Critique` subsystem, allow for more code reuse and thus make it easier to extend the system with new section types.



ReportGenerator facade We use the Facade pattern to hide implementation details of the `Analysis` and the `Critique` package and to present a clean interface. The `ReportGeneratorFacade` represents this facade object. It includes methods to configure both the analysis phase and the critique phase. The developer configures the analysis phase by specifying a file that contains the persisted `Events` that should be analyzed. And the developer also specifies a list of `Interpreters` that should be run during the analysis phase. The developer configures the critique phase by setting an output filename for the report and adding a number of `Sections`. The report is then generated by calling the `generateReport()` method.

Section HTML generation `Sections` are the building blocks of a `Report` and therefore also consist of HTML code. This HTML code is in turn made up of many individual components called HTML *elements* (or "tags"). To better reflect this structure in the `Critique` subsystem we added the `Element` object to represent the building blocks of individual `Sections`. This makes it easy to output textual information as a vertical list in a `Section`. Listing 4.7 and Listing 4.8 show how the HTML code for a `Section` is generated using the `OverviewSection` object as an example.

Listing 4.7: Section HTML generation (`Section`)

```
class Section(object):
    # ...

    def generate_html(self, result_store):
        s = self.header
        if self.elements:
            s += '<ul type="square">'
            for e in self.elements:
                s += '<li>%s</li>' % e.body
            s += "</ul>"
        return s
```

Listing 4.8: Section HTML generation (`OverviewSection`)

```
class OverviewSection(Section):
    def __init__(self):
        super(OverviewSection, self).__init__("Overview")

    def generate_html(self, result_store):
        overviews = result_store.get_results_of_type('OverviewResult')
        if overviews:
            data = overviews[0].data
            self.add_element(Element("Session started at %s" % data['sessionStart']))
            self.add_element(Element("Session ended at %s" % data['sessionEnd']))
            self.add_element(Element("%d events were recorded" % data['numEvents']))
        return super(OverviewSection, self).generate_html(result_store)
```

5 Prototypical implementation

In this chapter we present the prototype of the `muEvaluationFramework`. We show how the prototype handles the capture, analysis and critique phases.

5.1 Overview

We implemented a prototype during the writing of the thesis to fulfill several goals. We wanted to improve the design of the framework in an iterative process. To do so, we explored how different approaches work and what kinds of usability data can be gathered on the Apple iOS platform. At the beginning of the thesis it was not clear how the collected data could be analyzed and presented to the developer. We also used the prototype to explore this area.

The prototypical implementation was tested with two open source iOS applications: *Wordpress for iOS*, a companion tool for the popular PHP-based content management system *Wordpress*; and *PlainNote*, a minimalist note-taking application. Wordpress for iOS was more difficult to work with because it requires a steady internet connection to connect to a Wordpress account. During development of the prototype this requirement was sometimes difficult to fulfill and therefore we switched the host-application and worked with PlainNote instead. This offered the benefit of testing the prototype with an entirely different application and also ensured that all setup steps were verified once more.

5.2 Capture support

The prototype provides support for capturing usability data inside an host-application, to transmit these data over a wireless network connection and to store them into a file for later analysis.

The `CaptureLibrary` is implemented in the Objective-C language because this is a requirement to interface with host-applications on the iOS platform. The `CaptureServer` is implemented in the Python language.

Network support The prototype of the `CaptureServer` announces its service via Zeroconf and for this we used the *pybonjour* library [29]. The `CaptureLibrary` automatically finds and connects to the `CaptureServer` as soon as it is available. Not requiring any configuration, such as entering IP addresses and port numbers, proved very to be useful during the development of the prototype because the system "just works" in almost every network setup. The network connection uses standard sockets for TCP/IP-based data transmission. The protocol is very simple and stateless, and `Events` are transmitted as plaintext XML-strings. In the `CaptureLibrary`, the transmission of events is performed in background threads using Apple's *Grand Central Dispatch*, a framework which implements parallel execution of tasks using thread pools [3]. This ensures that data transmission does not hinder the performance of the host-application too much. Indeed, we found that the subjective application performance was unchanged with the two applications we tested.

Implemented sensors The prototype contains three sensor types:

- `MFTouchInputSensor`: a sensor that attaches to the `sendEvent:` method of the `UIWindow` class via the *method interception* technique we described in Chapter 4 and captures `UIEvents` of the type `UIEventTypeTouches`. This allows the prototype to track all kinds of finger gestures: quick taps, drag movements and scrolling.
- `MFUIControlSensor`: this sensor attaches to the `UIControl` class, a base class for most user interface elements on iOS. By intercepting the `sendAction:to:forEvent:` method the prototype can capture press events for all subclasses of `UIControl`, such as `UIButton` or `UISwitch`.
- `MFScreenSensor`: this sensor is used to detect changes to what is displayed on the mobile device's screen. It attaches to `UIWindow`, `UIViewController` and `UINavigationController` to detect when `UIViews` are added or removed from the view stack. This sensor also takes a screenshot and a snapshot of the complete view hierarchy of the topmost `UIWindow` whenever the active view changes. Screenshots of a running application are taken using the private framework `UIGetScreenImage()` function or by iterating through the view hierarchy and rendering each view into a `CGImage` (UIGetScreenImage is not available on the iOS simulator, for example). Screenshots are compressed using the PNG format and then transmitted over the network. To keep the host-application responsive the compression is performed in a background thread.

5.3 Analysis support

Implemented interpreters

- `OverviewInterpreter`: calculates when the evaluation session started (the timestamp of the first `Event`), when it ended (the timestamp of the last `Event`). Additionally, it shows the number of `Events` that were recorded.
- `ScreenshotInterpreter`: searches the event stream for screenshot images and collects them all in a `ScreenshotResult` list. Inside this list the screenshots are ordered by the time they were taken at. This ordering is the used by other `Interpreters` to reference the screenshots, e.g. the screenshot with index *3* maps to image data *A*. This ensures that no duplicate images are stored in the report.
- `TouchHeatmapInterpreter`: gathers `UserInputEvents` and maps them to screenshot images, therefore the interpreter requires the results of the `ScreenshotInterpreter`. Each touch event is mapped to a screenshot image by comparing the timestamp of the event to the timestamp of each screenshot and then selecting the last screenshot that was taken before the event happened. This ensures that all events are mapped to a screenshot and that touch events for different views do not all end up in the same screenshot.
- `ViewChangesInterpreter`: iterates through all view change events, generates a string representation of the view stack for each change and maps them to screenshot indices. This interpreter requires the results of the `OverviewInterpreter` and the `ScreenshotInterpreter`.
- `ViewDurationInterpreter`: gathers the timestamps of all view change events and generates a statistic that shows for how long each view was active. This interpreter requires the results of the `OverviewInterpreter` and the `ViewChangesInterpreter`.
- `UIGuidelineChecker`: performs a depth-first search on the nodes of each `UIState` event and checks if the elements in the view hierarchy have dimensions smaller than 44 pixels. This heuristic is given in Apple's *iOS Human Interface Guidelines* [6] and used as a proof-of-concept that the prototype can detect usability problems in a host-application.

5.4 Critique support

HTML reports The system generates reports in the form of a single HTML file that references a number of screenshots that are stored as separate files in the PNG format. The report document uses *Cascading Style Sheets* (CSS) to provide a consistent layout for section headers. `Reports` are non-interactive in the prototype but individual `Sections` can include JavaScript code to provide such functionality.

Implemented section types The report generator of the prototype supports the following five section types:

- `OverviewSection`: provides general information about the session, that is the time when the session started or ended, and the number of events that were recorded.
- `ViewDurationSection`: displays a table that contains the time the user spent on each view in the host-application. The table is sorted by time and absolute as well as relative timings are given for each view.
- `NavigationPathSection`: shows how a user navigated through the views of a host-application. For each step, a timestamp is given and the state of the view stack is printed out. Optionally, this section can show screenshots for each view change and also indicate the kind of view change, i.e. whether the view is a regular view or a modal view.
- `TouchHeatmapsSection`: shows screenshots of the host-application and draws colored circles to indicate finger touches that were performed by the user. The visualization of the touches is done in the browser via the HTML5 `<canvas>` element. This lays the foundation for interactively changing the presentation of the touch events and also allows the same set of screenshots to be used elsewhere.
- `UIGuidelineIssuesSection`: shows the detected violations against Apple's *iOS Human Interface Guidelines* [6]. Violations are shown as a textual list that is typeset in red font color.

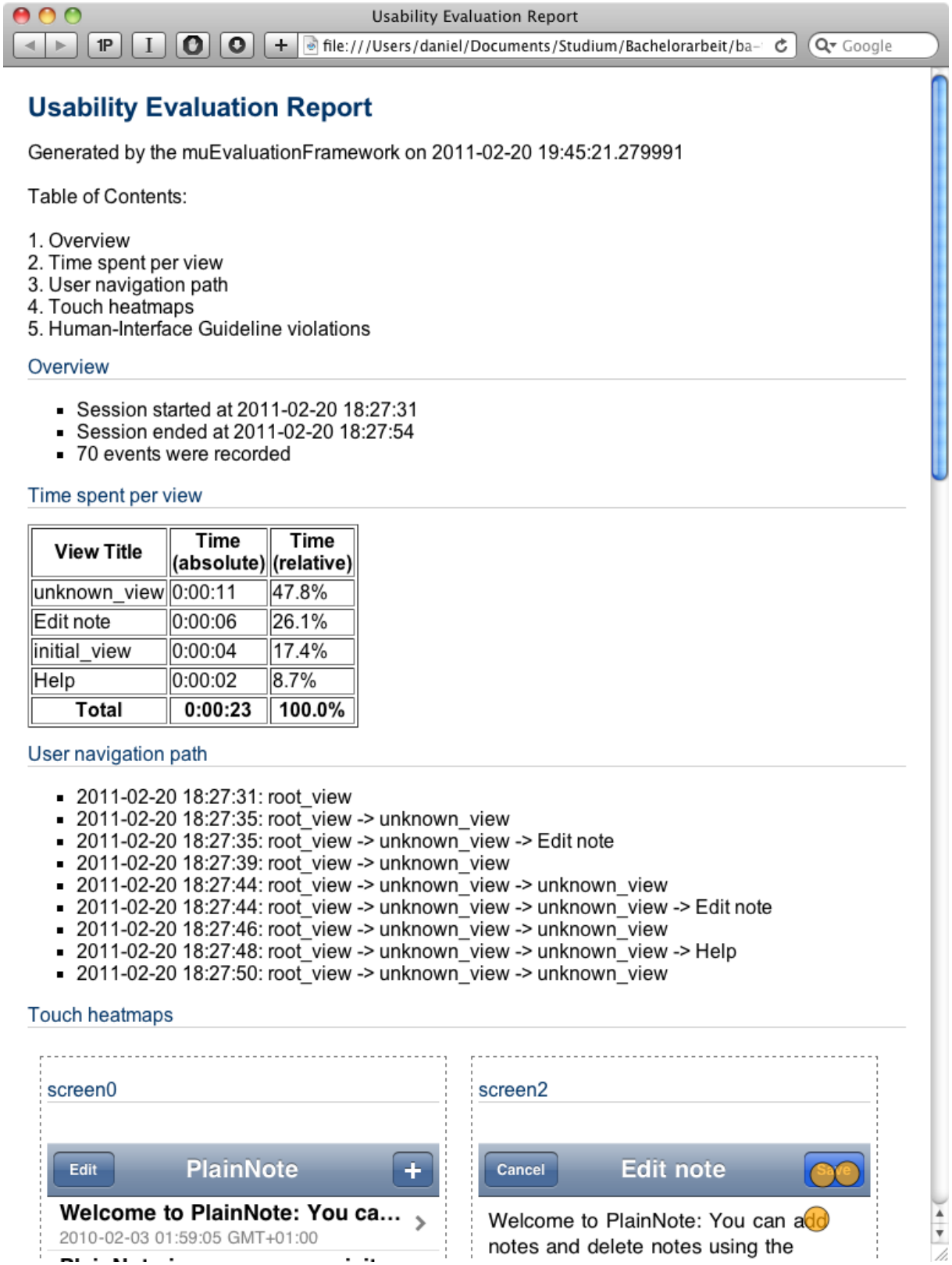


Figure 5.1: A finished report document (I) (Prototype screenshot)

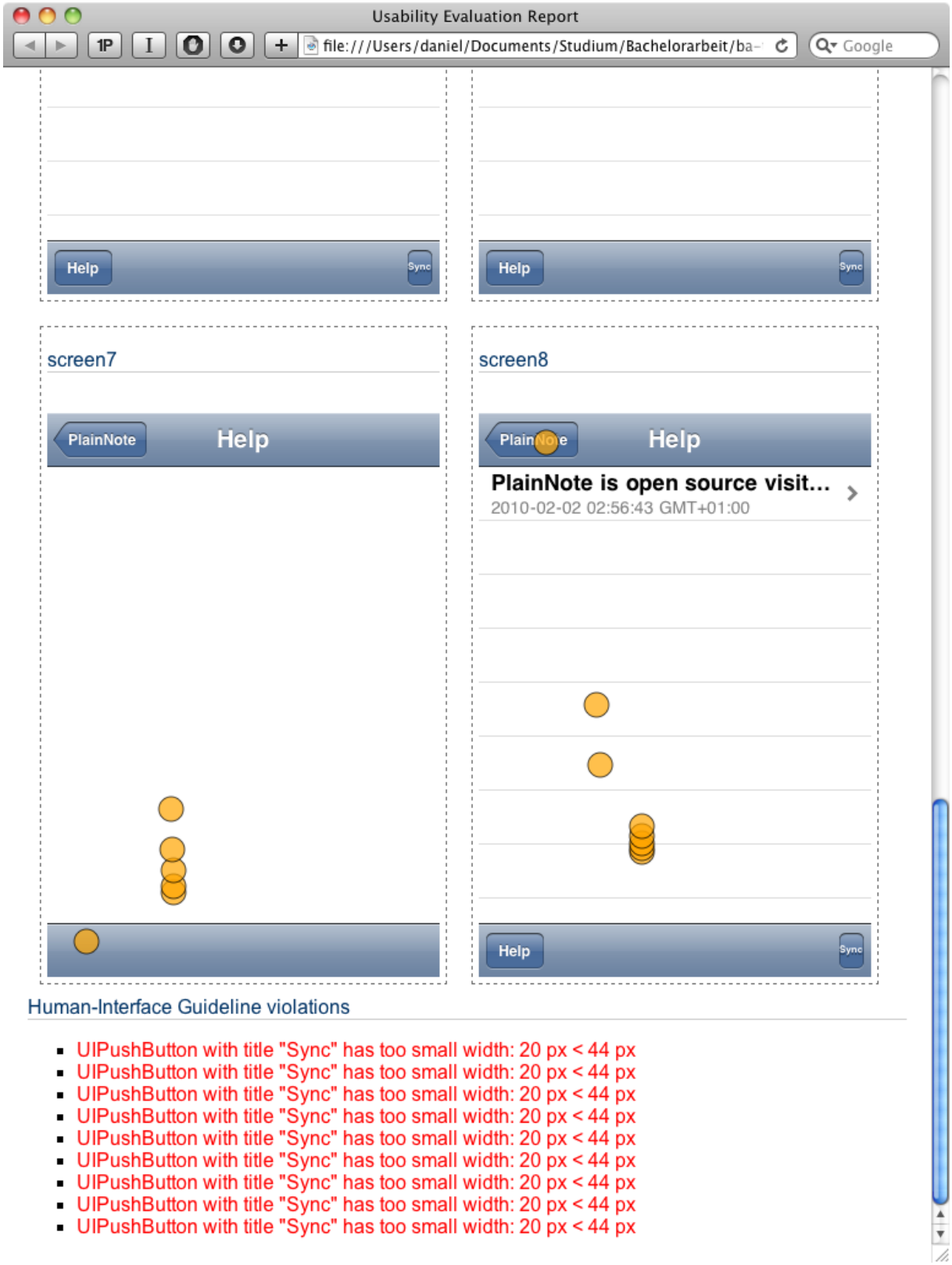


Figure 5.2: A finished report document (II) (Prototype screenshot)

6 Future work

During the development process we encountered several interesting issues for the future development of the present approach and we summarize them in the following sections.

6.1 Evaluation of a real product

To further improve the framework's support for automated usability evaluations it should be used in a test case with a real product and real test users. Although we used the prototypical implementation of the framework with two applications that are available in the Apple iOS AppStore[1] (*Wordpress for iOS* and *PlainNote*), this would be a chance to further refine the framework and to test it in a real world scenario. Based on the feedback of the developers we could add new sensors and section types to further enhance the use of the framework.

6.2 Web-based automated usability evaluation

The framework could be extended to include the abilities that were mentioned in the visionary scenario. We could add a number of sensor and section types that enable advanced functionality such as recording a video of the test user's face or analyzing the user's speech for strong emotions. In additions to these improvements, the `muEvaluation-Framework` could be made available as a web service. This means that developers who want to test a host-application would simply upload their application binary to a server and then select a few options, such as the number of test users, the active sensors and sections, and tasks for the test users. The framework would now automatically distribute the evaluation to a suitable group of test users, then wait for them to finish the evaluations, and return the results (i.e. the finished reports) back to the developers.

[1] The iOS AppStore is an online service provided by Apple Inc. that allows users to purchase and download applications ("Apps" for iOS-based devices).

6.3 Google Android support

Next to the iOS platform, Google's Android is another important platform for the smartphone market. Because of the similarities in the usability concept of both platforms, it makes much more sense to extend the `muEvaluationFramework` to the Android platform than, for example, to Nokia's Symbian. Android applications are mostly implemented in the Java language. This means that the `CaptureLibrary` subsystem of the framework would have to be ported to the Java language and be enabled to work with the Android UI toolkit. Because Java provides very good support for reflection we are sure that this is a viable option.

7 Conclusion

In this thesis we developed the `muEvaluationFramework`, an application-independent software for remote usability evaluation on mobile platforms. The framework supports automation in the capture, analysis, and critique phases of an usability evaluation.

First, during the capture phase, the framework collects usability data such as user input or application events while a test user interacts with the host-application. This data collection is performed by an extensible set of software-based sensors. Second, during the analysis phase, interpreter algorithms analyze the usability data that were collected in the capture phase. This analysis step is performed automatically. It summarizes the evaluation session and reveals usability issues in the analyzed application. Third, during the critique phase, a report is generated that contains multiple sections with the results of the analysis phase for the developer. Reports are generated in the HTML format and can include interactive elements and embedded multimedia content.

We developed a prototypical implementation of the framework for the Apple iOS platform. This prototype was tested with two open-source applications, *Wordpress for iOS* and *PlainNote* that are available on the iOS AppStore. The prototypical implementation shows how usability data can be collected on the iOS platform and how it can be automatically analyzed. We therefore believe that the prototype demonstrates that the software architecture for the framework is viable, and that it can be used as a basis for future research in the area of automated remote usability evaluation.

Bibliography

[1] Andreasen, M., Nielsen, H., Schrøder, S., and Stage, J. What happened to remote usability testing?: an empirical study of three methods. *Proceedings of the SIGCHI conference on Human factors in computing systems* (2007), 1405–1414. 7

[2] Apple Inc. Coding guidelines for Cocoa. `http://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/CodingGuidelines/CodingGuidelines.pdf`, May 2010. 55

[3] Apple Inc. Grand Central Dispatch (GCD) reference. `http://developer.apple.com/library/mac/documentation/Performance/Reference/GCD_libdispatch_Ref/GCD_libdispatch_Ref.pdf`, May 2010. 72

[4] Apple Inc. Objective-C runtime reference. `http://developer.apple.com/library/mac/documentation/Cocoa/Reference/ObjCRuntimeRef/ObjCRuntimeRef.pdf`, June 2010. 59

[5] Apple Inc. Apple iOS platform homepage. `http://www.apple.com/ios`, Feb. 2011. 47

[6] Apple Inc. iOS human interface guidelines. `http://developer.apple.com/library/ios/documentation/UserExperience/Conceptual/MobileHIG/MobileHIG.pdf`, Jan. 2011. 19, 20, 21, 73, 74

[7] Ash, M. MAObjCRuntime library. `https://github.com/mikeash/MAObjCRuntime`, Sept. 2010. 59

[8] Au, F., Baker, S., Warren, I., and Dobbie, G. Automated usability testing framework. *AUIC '08: Proceedings of the ninth conference on Australasian user interface 76* (Jan 2008). 3, 7

[9] Balagtas-Fernandez, F., and Hussmann, H. A methodology and framework to simplify usability analysis of mobile applications. *ASE '09: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering* (Nov 2009). 1, 10

[10] Balbo, S. Automatic evaluation of user interface usability: Dream or reality? *Proceedings of the Queensland Computer-Human Interaction Symposium* (1995). 7

[11] Bevan, N. Measuring usability as quality of use. *Software Quality Journal 4*, 2 (1995), 115–130. 3

[12] Bruegge, B., and Dutoit, A. H. *Object-Oriented Software Engineering: Conquering Complex and Changing Systems*. Prentice Hall, 2000. 48

[13] Buck, E. M., and Yacktman, D. A. *Cocoa Design Patterns*, 1st ed. Addison-Wesley Professional, 2009. 57, 58

[14] Cheshire, S. Zero configuration networking. `http://www.zeroconf.org`, Feb. 2011. 58

[15] Duh, H., Tan, G., and Chen, V. Usability evaluation for mobile device: a comparison of laboratory and field tests. *MobileHCI '06: Proceedings of the 8th conference on Human-computer interaction with mobile devices and services* (Sep 2006). 1

[16] Google Inc. Android platform homepage. `http://www.android.com`, Feb. 2011. 47

[17] Google Inc. Developer's guide - google analytics for mobile. `http://code.google.com/mobile/analytics/docs`, Feb. 2011. 11

[18] Hartson, H., Castillo, J., Kelso, J., and Neale, W. Remote evaluation: the network as an extension of the usability laboratory. *CHI '96: Proceedings of the SIGCHI conference on Human factors in computing systems: common ground* (Apr 1996). 6

[19] Hipp, D. R. SQLite homepage. `http://www.sqlite.org`, Feb. 2011. 12

[20] Holzinger, A. Usability engineering methods for software developers. *Communications of the ACM 48*, 1 (Jan 2005). 3, 5

[21] Ivory, M., and Hearst, M. The state of the art in automating usability evaluation of user interfaces. *Computing Surveys (CSUR 33*, 4 (Dec 2001). 5, 7, 8

[22] Kaikkonen, A., Kallio, T., Kekäläinen, A., Kankainen, A., and Cankar, M. Usability testing of mobile applications: A comparison between laboratory and field testing. *Journal of Usability Studies 1*, 1 (2005), 4–16. 1

[23] Kjeldskov, J., and Stage, J. New techniques for usability evaluation of mobile systems. *International Journal of Human-Computer Studies 60*, 5-6 (2004), 599–620. 1

[24] Nielsen, J. *Usability engineering*. Academic Press, Boston, 1993. 3, 5

[25] Nokia. Symbian blog | Symbian at nokia. `http://symbian.nokia.com`, Feb. 2011. 47

[26] Object Management Group. UML superstructure specification, v.2.1.2. `http://www.omg.org/spec/UML/2.1.2/Superstructure/PDF`, Nov. 2007. 3

[27] Paternò, F., Russino, A., and Santoro, C. Remote evaluation of mobile applications. *TAMODIA 2007* (2007), 155 – 169. 2, 7, 9

[28] Shneiderman, B., and Plaisant, C. *Designing the User Interface: Strategies for Effective Human-Computer Interaction, Fifth Edition*. Pearson Higher Education, 2009. 5

[29] Stawarz, C. The pybonjour library homepage. `http://code.google.com/p/pybonjour/`, Feb. 2011. 72

[30] van Rossum, G., and Warsaw, B. Style guide for Python code. `http://www.python.org/dev/peps/pep-0008`, Feb. 2011. 55

[31] Vasudeva, V. *Software Architecture*. Dorling Kindersley, 2009. 33

[32] Zhang, D., and Adipat, B. Challenges, methodologies, and issues in the usability testing of mobile applications. *International Journal of Human-Computer Interaction 18*, 3 (2005), 293–308. 3, 4