POSEiDON

Pers**O**nalized **S**mart **E**nvironments to increase **I**nclusion of people with **DO**wn's sy**N**drome

Deliverable D5.1

Development framework

Call:	FP7-ICT-2013-10
Objective:	ICT-2013.5.3 ICT for smart and personalised inclusion
Contractual delivery date:	M6
Actual delivery date:	15.07.2016
Version:	V4
Author:	Lars Thomas Boye, Tellu AS
Contributors:	Andreas Braun, Fraunhofer Dean Kramer, MU Silvia Rus, Fraunhofer Alexandra Covaci, MU Unai Alegre, MU
Reviewers:	Andreas Braun, Fraunhofer Juan Carlos Augusto, MU
Dissemination level:	Public
Number of pages:	95

Contents

Executive summary

This deliverable presents the POSEIDON development framework, and documents our work with defining this framework. The developer framework is a collection of methodologies, infrastructure, middleware, tools, specifications, etc. These are things which are used in the development of the POSEIDON prototype system and part of it, but also provided to other developers interested in developing applications for people with cognitive disabilities and/or connect to the POSEIDON system. This is partly methodologies, open-source code and free tools of general use, and partly the POSEIDON infrastructure which other applications can make use of.

A goal of the POSEIDON project is to foster development of inclusive services for people with Down's syndrome, and a commercial POSEIDON solution needs to be extensible with new services. The development framework is the project's way of addressing these goals. It is associated to Task 5.1 - Establishment of a development framework. This task is closely linked to the other tasks of WP5, and the framework has been developed in tandem with the prototype system, with parts of the final infrastructure being formalised from prototypes. The framework task is also connected to the technology development of work packages 3 and 4, most importantly Task 3.1 on middleware for context awareness. Context awareness middleware is an important component of the framework.

Early in the project, it became clear that the initial plan from the project DoW – to have a framework finalised in the first six months of the project – was not feasible. To produce a solid and complete framework, the design needs to be an on-going process iterating with the prototype development. This process was mostly concluded as documented in the third version of the deliverable, which had little in common with previous versions of the document.

Version 4 update: A section 4.4 has been added, giving an overview of how the framework infrastructure supports personalisation. A description of the learning platform has also been added, as section 5.5.

The document first introduces its subject, giving an overview of what we mean by POSEIDON Development Framework. We have thoroughly analysed and discussed the framework requirements first listed in deliverable D2.1, in chapter 2. Chapters 3-6 presents the main parts of the framework. Methodologies are described in chapter 3. The framework architecture in chapter 4 describes the needed infrastructure and how applications are integrated in this system. The infrastructure is composed of a number of components, described in chapter 5. The framework also includes software and code libraries provided to developers to enable them to develop POSEIDON applications, and these are described in chapter 6. Architecture designs of previous versions of the framework are found in an appendix.

1 Introduction

A goal of the POSEIDON project is to foster development of inclusive services for people with Down syndrome. The project is developing a prototype system of applications and services, to support people with Down syndrome as well as those who interact with them on a daily basis. The hope is that the prototype system will be viable and useful enough that it can go on to be part of a commercial service for people with Down syndrome and their carers.

The development framework has multiple roles in this project vision. Firstly, it is methodologies and tools used to develop the prototype system, and the technology and infrastructure forming the basis of the system. The methodologies and tools must be sound and well-proven to allow developers to efficiently develop the complex system envisaged in the Description of Work. The technology and infrastructure must provide the necessary functionality in an efficient, secure and robust way. If a commercial POSEIDON solution is realised, the system needs to be maintainable and extensible with new services. So the development framework must insure an open architecture which can form the basis of further development. It is a goal in the project to provide a technological infrastructure to foster the development of services to support inclusion of people with Down syndrome and others in closely similar situations. The framework needs to be generally usable outside the project and its prototypes. The development framework is one of the key outcomes of the project. Figure 1 gives a simplified illustration of the role of the framework in the project, being the basis for the prototypes but also extending beyond the domain of the POSEIDON system.

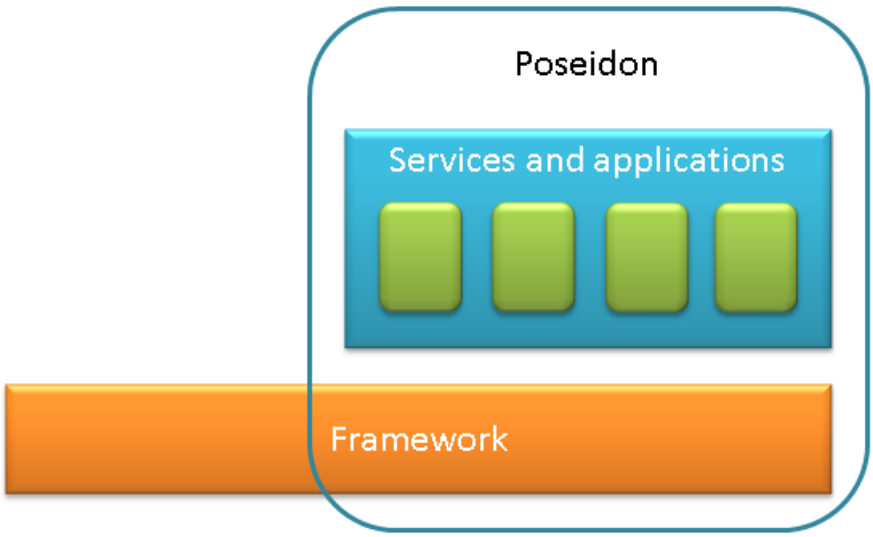

Figure 1: POSEIDON and its framework

The work has been to develop methodologies based on best practices in software engineering, and find a set of components, services and tools that together provide the needed basis for building assistive applications and services. This framework must be documented and made available to others. The hope is to attract interest and help others develop applications for this sector of the society. If others build on the same framework, it also enables better compatibility and synergies between applications. This means that we needed to think bigger than just the prototype requirements when selecting and designing the framework. As a framework, it needs to be generic. And those parts which will be needed by developers of new services and applications have to be made freely available (Open Source if source code is needed) as well as be properly documented. The

prototypes we have created in the project, in addition to providing value from their own functionality, serves as tests and illustrations for framework use.

1.1 Framework content

To structure the discussion and documentation of the development framework, we have organised the final version of this deliverable according to some main categories. Firstly we have system development methodologies, which specifies structured ways of creating ICT systems. Chapter 3 describes three methodologies: a User Centered Software Development Process, the ethics framework eFriend, and a methodology for requirements elicitation in Context-Aware systems.

For developing a complex system with multiple applications on multiple platforms, sharing a common set of services, an architecture is important. The framework architecture, presented in chapter 4, describes this infrastructure and how applications are integrated in this system. The main pillars of the infrastructure had to be sought in existing technology and frameworks, both because of the scope and size of our task, because it would be a waste of resources to create something new where technology meeting the requirements already is available, and because we need mature, tried and tested technology for the basis of our solution. The development of the architecture and infrastructure was very much guided by the wanted functionality for the prototype system, based on the Description of Work and the WP2 work of requirements and design. Rather than providing a final framework early on in the project, as was the original plan, much of the infrastructure has been created by formalising and generalising prototype work, making what we have created part of the framework. The final framework architecture therefore supports a number of functional aspects, with shared services which all POSEIDON applications can connect to. The infrastructure which implements the architecture is described in chapter 5.

Tellu's SmartPlatform (or SmartTracker service) is mentioned in the DoW, and is a server-side component which has been further developed in the project and which stores POSEIDON user accounts with user profiles, and also tracked data which should be available to POSEIDON applications or administrative personnel. Based on the needs for shared data, especially instructional content and media, a file server has been developed and included in the framework infrastructure, along with specifications for the data it holds. A calendar service is included, as management of time is an important functional aspect. As this is a research project and our work has been experimental, not all components investigated have been included in the final infrastructure. We especially had plans for the universAAL framework, described in earlier versions of this deliverable, but what it provides was found not to match what we needed for the POSEIDON infrastructure. Framework developments no longer relevant are described in the appendix.

The platforms for client applications are an important part of the architecture. The types of client hardware devices to be used in the POSEIDON prototypes were given from the onset, described in the Description of Work. We make a distinction between stationary and mobile application platforms. Stationary platforms are for using applications at home or in other stationary locations, and are based on laptop and desktop computers. These provide large screens and various input devices, and are suited to primary user training and secondary user management tasks. Here we have the interactive table as part of the infrastructure. It is a prototype of a new interaction device, which combines the size of a multitouch table with 3D hand position recognition. It is intended to be unobtrusively built into tables, allowing the user to control applications via hand gesture on or over the table.

Mobile platforms are smaller devices such as phones and tablets, suited to bringing with you wherever you go and using outside. These are needed to provide notifications and guidance to the

primary users. We have a goal of supporting multiple platforms for both stationary and mobile use. We specify a framework for web applications, responsive to run on all devices. For more specialised functionality, we provide additional support for stationary training applications developed in the cross-platform framework Unity, and mobile applications for Android devices.

The architecture includes middleware and application components for the supported platforms. Most importantly we have the Context Reasoner Middleware. It provides context awareness to mobile applications. Applications connect to it, causing it to perform acquisition of and reasoning over context, notifying the interested applications of changes. The context awareness part of the framework also includes a tool for producing context rules and a server-side for collecting data from the middleware.

The last main part of the framework documented in this delivery, in chapter 6, is the developer tools and components. This is software and code libraries provided to developers to enable them to develop POSEIDON applications. It includes tool support for the requirements elicitation methodology, a context modelling tool for creating new context reasoner rules, the Android and web developer frameworks, and code libraries and examples.

A final part of the framework considers user interface design. We provide specifications, information and graphical elements for user interfaces. This is documented in WP4, so not elaborated upon in this deliverable. The developer documentation includes guidelines for developing accessible user interfaces, and information about mobile and web application UI development. A user interface strategy is specified, considering user requirements. A colour palette and icons are provided, to create applications with a shared look and feel.

1.2 First version of deliverable

This deliverable was originally scheduled for month six of the project. The ambition in the project planning was to have a complete framework defined early on, and then use that for developing the three main prototype iterations. It soon became apparent that this was not feasible. Establishing a complete framework is a much larger task, and it needs to iterate with prototype iteration. Of the two main tasks completed for the original deliverable, the first was to specify and analyse framework requirements. This was presented in chapter 2, and the original requirements are retained in the final version, just augmenting each with a section discussing how it has been achieved. The second task was to get a first framework in place to support the first prototype. The components were those outlined in the DoW, and they were described in the deliverable. We also gave some plans for the framework development after the first prototype, such as investigating the use of universAAL.

Due to the limited time, the framework presented was a minimum needed to be able to rapidly build the first prototype, and not something which met all the requirements. In this iterative process of selecting framework components, building prototypes and assessing the components it also takes time to find the right border between framework and applications. So it was clear that the framework task would have to run through most of the project rather than be completed in the first six months, and that we would need to revise this deliverable to document the later results.

1.3 Second version of deliverable

The second version of this deliverable was updated to reflect the state of the framework as it was to be used for the second prototype iteration and first pilot. Chapter 4 from version 1 (Framework roadmap) had been split into two chapters, a chapter 3 describing various framework components and a chapter 4 describing the framework architecture as a whole, for the two prototype iterations. The executive summary, introduction and conclusion was updated to reflect the changes. As was

pointed out, the framework was still very much a work in progress at this point in the project, and it was not meant as a final version of the deliverable.

1.4 Third version of deliverable

Major work on the framework followed the second prototype iteration, formalising the functional aspects of the prototype system into the framework and further developing and documenting the infrastructure. Methodologies and tools have also been documented to a much larger degree. The work of defining the final architecture and how it relates to the prototype system has also affected other deliverables. The result is that D5.1 has been completely reworked compared to the earlier versions. The original framework requirements have been retained unchanged in chapter 2, but a section on adherence has been added to each requirement section. Parts of the SmartPlatform description in section 5.1 has also carried over, and parts of the previous architecture description has been reworked into the appendix. Other than this, both the structure and the content of this deliverable is new for version 3.

If you have read a previous version of the deliverable, we suggest that you start from chapter 3 and read from there, possibly skipping some of the technical details in chapters 5 and 6. Afterwards, go through chapter 2 and read the Achievement section for each requirement, if you would like to see the conclusions related to the initial framework technical requirements.

1.5 Fourth version – personalisation and learning

The fourth version of the deliverable is a minor update of the third, mainly to more clearly address personalisation as a separate topic. Personalisation is a very important aspect for the POSEIDON solution, which needs to be supported by the framework. While the mechanisms were in place, in the form of personal preferences in the user profiles and personalised content, the use for personalisation was not shown, which was also requested as a result of the review of the third version. Section 4.4 has therefore been added to give an overview of how the framework infrastructure supports personalisation.

The other addition is section 5.5, giving an overview of the learning platform of the infrastructure. For the rest of the document, only some minor corrections were made.

1.6 Framework documentation

This deliverable is the main part of the documentation, giving both the overview and collecting much of the technical documentation. D5.4 Databases for integration of services has been reworked together with D5.1, and is now considered part of the framework documentation as a companion deliverable. It collects developer documentation for connecting to the server-side part of the infrastructure. For the user interface design aspect of the framework, we refer to deliverables D4.1 and D4.5. D4.1 Interface strategy outlines the user interface strategy for the POSEIDON system, and is a resource for user interface developers. D4.5 HCI user and developer manuals contains documentation on user interface design and implementation. Other technical documentation is also relevant to the framework. Deliverable D5.6 gives the overview of the technical documentation produced in the POSEIDON project.

As part of the framework documentation process, the documentation is being published on the POSEDION project web: http://www.poseidon-project.org/

This website now has a developer section. The framework is presented with an overview here, and all documentation and code components will be published here as they are ready.

2 Technical requirements

A listing of technical requirements for POSEIDON and the framework was produced in the early stages of the project, and listed in the requirements report, D 2.1. Here we repeat the list of framework requirements, and analyse what each of them means for the framework selection and design. For this discussion, we have grouped the framework requirements into some categories. See the category column added to the table – the rest of this chapter is organised according to these categories.

The requirements themselves and their discussion have been kept unchanged from the initial version of the deliverable. A section on achievement has been added to each requirement in this version, where we look at how and to what extent the framework fulfils the requirement. As some of the prototype development took other courses than foreseen before starting the development, not all requirements ended up being relevant.

Label	Requirement	Category	Proto-type	Priority
Fr1	Support the development of the POSEIDON prototypes according to the POSEIDON requirements and constraints.	Overall	1	0 – Seen as mandatory at all times
Fr2	Enable connecting client devices with server-side services, as well as server-side services to each other, to form a POSEIDON system.	Interconnection	2	1
Fr3	Support various client devices. The required set in the project is tablets, interactive table and virtual reality set, but the framework should be open for any type of device running a major operating system or connected to a computer running a major operating system.	Client	3	10
Fr4	Support mobile use – mobile devices, and mobile networks such as 3G.	Interconnection	2	0
Fr5	When live, support the safety of the end users.	Functionality	3	20
Fr6	When live, support the privacy of the end-users. POSEIDON to provide optional user privacy settings to enable customization. Default settings will be provided.	Functionality	3	10
Fr7	Support context-awareness.	Functionality	2	7
Fr8	Support for optional interface customization to suit an end-user's needs. Several sets of default settings will likely be provided.	Functionality	2	5
Fr9	Storage of user data at the server-side with appropriate backup to safeguard. Optional user settings to customize data storage requirements with a default setting provided.	Functionality	2	2
Fr10	Provide a mature, stable and well-integrated software development environment for writing, testing and deploying POSEIDON software components.	Developer	2	0
Fr11	Documentation must be provided, to enable project participants and third parties to develop POSEIDON components.	Developer	2	3
Fr12	Allow new server-side services to be developed and plugged into a POSEIDON system by third party developers, giving a clean, documented and standards-	Developer	3	30

Label	Requirement	Category	Proto-type	Priority
	compliant API to do so without knowing the inner workings of the framework or the other components in the system.			
Fr13	Allow new client devices and applications to be developed and plugged into a POSEIDON system by third party developers, giving a clean, documented and standards-compliant API to do so without knowing the inner workings of the framework or the other components in the system.	Developer	3	30
Fr14	For tablet and smartphone client applications (“apps”), the framework must be able to support all platforms with a significant (15 %) market share. This means Android and iOS at the current time, but the framework should not exclude future mobile platforms.	Client	iOS at 3	30
Fr15	Software components for connecting an app to a POSEIDON system will be made available for Android and iOS, to support POSEIDON app development on these platforms.	Developer	iOS at 3	30
Fr16	Enable distribution of apps to tablet and smartphone end users, in a way that makes it possible for non-technical users to install and update the apps with little effort.	Client	2	10
Fr17	When live, framework components should have robustness and fault-tolerance comparable to non-vital commercial systems.	Non-functional	2	20
Fr18	Enable efficient implementations of client applications and communication, so that end users experience performance comparable to other applications running on the same hardware.	Non-functional	2	10
Fr19	Enable near real-time communication between components in the POSEIDON system.	Interconnection	2	5
Fr20	Enable asynchronous communication, where server-side services can initiate communication with a client application.	Interconnection	2	4
Fr21	When live, enable the POSEIDON system to scale to large numbers of connected end users without significant performance loss.	Non-functional	3	30
Fr22	Extensible, allowing integration of new functionality not yet foreseen without breaking existing functionality.	Interconnection	3	10

2.1 Overall

A category for requirements too wide to fit in the other categories.

2.1.1 Fr1 – Support POSEIDON

Summary: Support the development of the POSEIDON prototypes according to the POSEIDON requirements and constraints.

Discussion: While the framework must be generic enough to support services and hardware beyond what we consider in the project, the most important requirement is that it can support the

prototypes we want to build in the project. It must therefore support the prototype-specific functional and non-functional requirements listed in deliverable D2.1. Many of the rest of the framework requirements are also guided in part by Fr1.

Achievement: While a final evaluation of the prototype functional and non-functional requirements of the prototype system is outside the scope of this deliverable, and must wait until the final prototype iteration is done, the work so far has been on track to address at least the vast majority of them. The system architecture has been guided by the wanted functionality, and the technical infrastructure has been developed from the prototypes, as described in chapter 4.1.

See also: Fr22, Fr5-9

2.2 Interconnection

A system such as POSEIDON can consist of various server-side services and client applications/devices, and these must be interconnected to form a cohesive system. The requirements in this category specify requirements for interconnection and integration, including protocol requirements.

2.2.1 Fr2 – Interconnection

Summary: Enable connecting client devices with server-side services, as well as server-side services to each other, to form a POSEIDON system.

Discussion: Facilitating the interconnection of sub-systems to form a cohesive system is a core task for the framework. We will discuss the interconnection strategy after looking at the other, more specific requirements in this category.

Achievement: Facilitating interconnection is a core task of our infrastructure. We discuss the achievement of interconnection requirements in 2.2.6.

See also: Fr12-13

2.2.2 Fr4 – Mobile use

Summary: Support mobile use – mobile devices, and mobile networks such as 3G.

Discussion: Mobile use needs both a mobile device and a mobile network connection. Support for the first is more specifically handled in the client requirements Fr3 and Fr14. Considering network support, internet over cellular networks (3G/4G) will be part of the infrastructure the system needs for mobile use. All relevant mobile device platforms support such network connections. A mobile internet connection may be slow or lost in some cases. The framework should support efficient communication (Fr18), without unneeded overhead. It should also support robustness with regards to network fallouts – in the case where software components dealing with network transactions are provided for mobile client development, these should include local caching to handle intermittent network unavailability.

Achievement: Internet over cellular networks is part of the infrastructure and supported by the mobile platforms. Our network transactions mainly use HTTP, and the data specifications are for light-weight JSON files. Some of these reference media files, which can be quite large. Such files should be downloaded once and stored on the device, but this is left to the application developer. It was decided that context reasoning should be done locally on the mobile device, with middleware provided as part of the infrastructure, as an internet connection is not always available everywhere.

See also: Fr14-15, Fr18

2.2.3 Fr19 – Real-time communication

Summary: Enable near real-time communication between components in the POSEIDON system.

Discussion: Near real-time communication between components means that when a state changes in one component, this change is communicated to all other components which are interested in such a change, within a few seconds. Not all communication will be in real-time, but the framework must include technology and protocols which support it. One necessary element of real-time communication in this type of system is asynchronous communication, so that a server-side service can push the state change to client applications right away. This is discussed in the next requirements (Fr20). Another element is speed: A message from one component must be processed and passed on quickly. This means there should never be more than a few intermediary points between two components connected to the system, and all of these must be able to process the message in real-time.

Achievement: This requirement was included as real-time communication was believed to be important, mainly for context awareness. However, it was decided that context reasoning should be handled on the mobile device, as we cannot rely on a network connection always being available. This requirement therefore became less valid and has not been specifically addressed. The SmartPlatform service of the infrastructure does support asynchronous communication, and the architecture does not feature more than a few intermediary points between two components, so the requirement has been achieved as far as it is relevant.

See also: Fr20, Fr18

2.2.4 Fr20 – Asynchronous communication

Summary: Enable asynchronous communication, where server-side services can initiate communication with a client application.

Discussion: The classic client-server pull model, where the client application requests data from services and gets a reply in return, is not sufficient for all POSEIDON needs. Firstly, we need to be to push state changes from server-side services to client applications as they happen. High-frequency pulling of the server is a possible way to meet the near real-time requirement, but would put a big load on servers even when there is nothing to communicate. One potential technology is HTTP push, such as WebSockets[1], which is part of the emerging HTML5 standard. HTTP push is available for all modern HTTP implementations, including web browsers for web applications. However, it requires the connection to be kept open.

For mobile applications, we want to be able to send a message to the app without requiring the app to use resources by running in the background and keeping a connection open; even without the app running at all. Both Android and iOS, the two primary mobile platforms for POSEIDON prototypes, have push services on the system level which can be used to start an app and deliver a message to it. For Android, the service is Google Cloud Messaging[2]. These are platform-specific communication channels. The framework can hide the differences behind a generic communication interface, with implementations plugged in as needed.

Another aspect of asynchrony is that it may not be possible to deliver a message right away. The message must then be stored by the framework, and delivered as soon as possible.

[1] http://tools.ietf.org/html/rfc6455
[2] http://developer.android.com/google/gcm/

Achievement: This requirement is related to Fr19, with we saw was based on the later revoked assumption that context information would flow through the system. The SmartPlatform has been given a subscription API, where clients can subscribe and updates are delivered with HTTP push, but this has not been exploited in the infrastructure so far, as this form of real-time communication was not needed in the prototype system. The architecture is therefore primarily based on the client-server model.

See also: Fr19

2.2.5 Fr22 – Extensibility

Summary: Extensible, allowing integration of new functionality not yet foreseen without breaking existing functionality.

Discussion: This is a key requirement to make the framework something more than a part of a POSEIDON prototype. It means that the framework must be generic, not built to handle just a specific functionality set. It should be pluggable, allowing new components to be plugged in without affecting unrelated components in the system. And the framework itself should be extendable, allowing modules to be added at the framework level.

Achievement: The framework architecture, described in chapter 4, does not in any way place restrictions on the addition of new components. Other requirements (Fr12-13) go into detail on the "pluggability".

See also: Fr12-13

2.2.6 Interconnection strategy

Having looked at the individual requirements for integration and interconnection, we can discuss a cohesive strategy to meet them all. This is basically a question of the communication infrastructure. The simplest model to build a system of client and server sub-systems is that of servers providing APIs, with clients and other servers communicating with a server through its API. But this means that each sub-system must be built for specific APIs. Components should have as few direct dependencies as possible. To provide a more fixed structure and framework functionality, we can use a central framework server. This will then act as a communication hub, and other components only need to communicate directly with this server. However, it is still not very flexible. As it bundles the infrastructure into a single component it is more difficult to modify and extend it. It is also a potential bottleneck, as all communication needs to go through this point.

For a more modular and less centralized approach, a bus or message broker middleware is a natural choice. Other framework functionality, such as data storage, can then be separate modules to be plugged into the bus as needed. New components can be plugged into the bus, and process relevant messages from other modules. Our conclusion in this section is therefore to seek a bus architecture and a generic, standardized message format for the communication infrastructure.

Achievement: Our framework architecture is a combination of the client-server model, with a number of server-side APIs, and the SmartPlatform service to some extent acting as a central server, holding account and data also used by other services. This is a pragmatic approach which serves the prototype system well. As we noted in the initial discussion, this is not the ideal structure. The plan was to use the universAAL framework to realise a more advanced interconnection strategy, as it has data buses for components to plug into. However, as is described in the appendix of this deliverable, universAAL was not found suitable for our needs.

2.3 Client

Here we look at some requirements on the framework's support for client application platforms.

2.3.1 Fr3 – Devices

Summary: Support various client devices. The required set in the project is tablets, interactive table and virtual reality set, but the framework should be open for any type of device running a major operating system or connected to a computer running a major operating system.

Discussion: For any form of computer with an internet connection, running a major operating system, it should be possible to develop the necessary software to connect it to the communication infrastructure. Major operating systems today are Windows (7/8), Linux, OS X, Android, iOS and Windows Phone. Since programming APIs with network functionality are available for all these platforms, we see no big barrier to the fulfillment of this requirement. If a specific bus API is chosen for the framework, it is also a question of whether this bus API has an implementation available for the platform. If not, connecting the platform to the framework requires either developing an endpoint implementation for the bus, or connecting to it through some standard protocol like HTTP. So it's important that at least one of these options is possible for all mentioned platforms.

Achievement: A large part of the infrastructure is server-side services, to which any client can connect with standard HTTP, so any platform with an internet connection can connect to these.

See also: Fr13, Fr14

2.3.2 Fr14 – App platforms

Summary: For tablet and smartphone client applications ("apps"), the framework must be able to support all platforms with a significant (15 %) market share. This means Android and iOS at the current time, but the framework should not exclude future mobile platforms.

Discussion: This requirement stresses that all major mobile platforms should be possible targets for apps connected to the framework, although this is a sub-set of the Fr3 requirement. There are generally two strategies for mobile app development today. One is to use the programming APIs available for each individual platform, to make so-called "native" apps (note that these are only native in a software sense, not necessarily in the traditional sense of being compiled for a specific hardware platform, as f.ex. Android lets the apps run on a wide range of processor architectures). With this strategy, all the implementation must be done separately for each platform. The other strategy is to use web technology (HTML, JavaScript and CSS), so that at least a significant part of the implementation can be used on all platforms. A framework such as PhoneGap[3] helps by automating the wrapping of the web code as an app for each platform. The web technology is mainly suited to implementing the user interface, light-weight logic and HTTP-based communication. For full access to device hardware and the rest of the system, or for better performance, a "native" back-end can be implemented separately for each platform in a hybrid approach.

Achievement: A more detailed discussion of platform strategies is given in chapter 4.2. We chose a combination of web technology for responsive applications running on all platforms and native applications for more specialised functionality. Although we have extended support for the Android platform, with context middleware only available here, there is nothing excluding other mobile platforms.

[3] http://phonegap.com/

See also: Fr3, Fr15

2.3.3 Fr16 – Distribution

Summary: Enable distribution of apps to tablet and smartphone end users, in a way that makes it possible for non-technical users to install and update the apps with little effort.

Discussion: This is an important aspect of client device support, as there is little value in developing apps for a device if the users aren't able to get and install those apps themselves. Installation could be done by a carer, but in any case it needs to be user friendly or it won't be successful. If the framework includes its own distribution system, and this is to be used, it must be available for all significant platforms (see Fr14), and it must be easy to use. Otherwise, all the mobile platforms have app stores, and these are a natural choice for distribution channel. These are user-friendly, mature and stable, and they tend to have good system integration, giving the developer installation statistics and crash reports.

Achievement: The app stores of mobile platforms are used for distribution of native apps, while web applications are available in the browser.

See also: Fr14

2.4 Functionality support

The requirements in this category deal with aspects of the services and applications to be built which the framework should support. Since it is the services and applications built on the framework which are ultimately responsible for the functionality provided to the end users, these requirements does not rest solely on the framework, and it is a question of how the responsibility for meeting them is distributed. As a minimum, the framework can't jeopardize their implementation. And functionality which is needed for most services and applications should ideally be implemented as part of the framework.

2.4.1 Fr5 – Safety

Summary: When live, support the safety of the end users.

Discussion: The safety of the end users is very important in our case, and the framework must play its part in supporting it. The system should be able to detect the level of safety of the user, so that unsafe situations can be avoided, or the user or carers can be notified if they do occur. This relates to context-awareness (Fr7). The notification of carers must be dependable, so this places requirements on the communication infrastructure. If one line of communication fails, there should be a back-up. And the general robustness and dependability of the system is important (Fr17).

Achievement: This requirement is mainly met through the related requirements Fr7 and Fr17. Other than providing a robust and dependable infrastructure, insuring safety has been placed in the application level.

See also: Fr7, Fr17

2.4.2 Fr6 – Privacy

Summary: When live, support the privacy of the end-users. POSEIDON to provide optional user privacy settings to enable customization. Default settings will be provided.

Discussion: For the framework, this requirement primarily means that the communication infrastructure and any data storage done by the framework must be secure, such as through

encryption. The framework should also avoid storing sensitive information, such as information to easily and uniquely identify the user or medical information. And what data to be stored should be customizable.

Achievement: This has been achieved as indicated in the discussion, with the framework infrastructure using encrypted communication (https) between components, a POSEIDON account required for authenticated data access. No sensitive information is stored. Privacy settings can be stored in the POSEIDON account.

See also: Fr9

2.4.3 Fr7 – Context-awareness

Summary: Support context-awareness.

Discussion: Context-awareness is one of the pillars of the POSEIDON project. Key user context aspects, such as location, time and what the user is doing, should be generally available to system components. A new component developed for the system should be able to rely on getting context information. This can be significant for APIs and protocols in the communication protocols. We may also want the deduction of context information and storage of such information to be included in the framework.

Achievement: Context awareness is primarily provided by middleware on the mobile device, described in chapter5.4. This currently only makes the context information available to applications on the device. However, the infrastructure has a good mechanism for making such information available to the rest of the system, in the SmartPlatform service which is made to receive such information, store and process it, and make it available to applications. This is currently used for position and some other device data, to make it available for monitoring.

2.4.4 Fr8 – Interface customization

Summary: Support for optional interface customization to suit an end-user's needs. Several sets of default settings will likely be provided.

Discussion: The user may have user interface preferences, such as font size, high-contrast and input and output modes. Rather than needing to configure this separately for each application in the system, the framework should store a general set of interface preferences. Various components can then access these and make use of them.

Achievement: This is achieved with the personal profile stored with the account on the SmartPlatform service.

See also: Fr9

2.4.5 Fr9 – Data storage

Summary: Storage of user data at the server-side with appropriate backup to safeguard. Optional user settings to customize data storage requirements with a default setting provided.

Discussion: We have seen some need for data storage at the framework level, such as for privacy settings, context and interface preferences. Other information that should be generally available within the system may also be discovered during the project or after. So the framework needs to include data storage, and this must be done safely and securely. This could be implemented with a database framework component.

Achievement: This is a requirement which became more important than anticipated, with the prototype development placing a focus on the need for shared data stores. Therefore the infrastructure has a strong server side where data storage is the main feature, including a file server. Details are described in sections 5.1-5.3.

See also: Fr6, Fr8

2.5 Non-functional properties

These are non-functional requirements for the framework itself, such as for the communication infrastructure.

2.5.1 Fr17 – Robustness

Summary: When live, framework components should have robustness and fault-tolerance comparable to non-vital commercial systems.

Discussion: It is essential for user acceptance that the system is dependable. This means that both the framework and the applications and services must be available and working very close to all the time. As some applications will be running on mobile systems, important resources such as network and GPS may be unavailable from time to time. This is not an error in the sense this requirement addresses, being outside our control, but rather a case we must plan and design for. Mobile applications must be designed to fail gracefully, taking care to work as well as they can with the resources available and not crash or give the user a technical error message. This aspect only relates to the framework as far as app components delivered as part of the framework are concerned (Fr15). The main concern for the framework, as it is the basis for the system, is that all framework components must be robust and free of serious bugs. They should be able to run with an uptime of at least 99 %. This means that framework components should be mature, well-tested technology. As the services and applications to be developed will be prototypes, and they will depend on the framework, it is important that the framework is never the weak link. In addition to contributing to dependability for the end user, the framework should be dependable for developers, so that when a new application is failing during development and testing, the developer can be fairly certain the fault lies within his own code.

Achievement: We have made sure to only use mature, well-tested technology for our infrastructure, except where it is of an experimental nature being developed in the project, in which case we have made sure to only use well-tested aspects in the pilots. This requirement is one of the reasons universAAL was not utilised, and a more traditional client-server approach was chosen for the architecture.

2.5.2 Fr18 – Efficiency

Summary: Enable efficient implementations of client applications and communication, so that end users experience performance comparable to other applications running on the same hardware.

Discussion: Firstly, efficiency is needed in the communication infrastructure, including any nodes the data must pass through before reaching its destination. This is needed for the responsiveness in the system, and to meet the real-time requirement. It means that communication protocols and processing should impose minimum overhead. Secondly, framework components such as database and mobile app components must be efficient to achieve responsiveness. And in addition to itself being efficient, the framework must support the development of efficient applications. Efficiency is especially important for mobile applications, as the hardware is relatively weak, and battery use is

also an issue. This means we do not want to enforce a heavy, high-level framework on mobile platforms.

Achievement: Our infrastructure mainly consists of middleware and server-side services, which provides their services to applications in a direct manner and themselves have little processing overhead. Communication between modules over the internet mainly consist of lightweight JSON data. The framework does not enforce any specific technologies for client applications, and those favoured by the framework – Android and Unity – allow efficient implementations.

See also: Fr19

2.5.3 Fr21 – Scalability

Summary: When live, enable the POSEIDON system to scale to large numbers of connected end users without significant performance loss.

Discussion: This requirement doesn't mean that a single server needs to be able to support a huge amount of simultaneously connected users, or that the initially deployed system must be able to support whatever amount of users we can hope for in the future. Server-side hardware and infrastructure will always need to be scaled to fit the user base – the requirement means that such scaling must be possible. The framework must have the potential to support any size of user base we can imagine for it, through enabling scaling with multiple server instances and load-balancing to maintain good performance for all users. However, our efficiency requirement (Fr18) means that the system should support a reasonable number of simultaneously connected users seen in relation to the deployed hardware.

Achievement: Having the context reasoning deployed on the mobile device, rather than requiring processing on a central sever, supports this requirement. The server-side infrastructure consists of traditional server applications with HTTP APIs. Such applications can be scaled using load-balancing. In addition, work is ongoing to partition the SmartPlatform into modules for a more scalable cloud deployment. To conclude, the chosen architecture lends itself to scaling when it becomes necessary.

See also: Fr18

2.6 Developer support

Finally we have requirements dealing with how the framework should support developers creating new applications and services for the system.

2.6.1 Fr10 – Development environment

Summary: Provide a mature, stable and well-integrated software development environment for writing, testing and deploying POSEIDON software components.

Discussion: Developers need well-working tools to be able to produce software components for the system. This is an important part of the framework, though it is for the most part not something we should produce in the project. We need to ensure that the necessary tools exist for whatever platforms and APIs we include in the framework. The tools should be mature and relatively bug-free – if the developers need to struggle to use the necessary tools, this will affect the productivity in the project and we can't expect the framework to be used outside the project. It is also a big plus for tools to have a large user base, so that if there is a problem, a solution or someone to ask can be found online. In addition to writing and compiling the software, the development environment must support testing and deployment.

Different platforms require different tools. For instance, Android has an extensive package of Development Tools freely available for the different desktop operating systems, which also integrates well with the major Software Development Kits. Eclipse[4] is a highly configurable and extendable software development platform with a lot of available components, including integration for the Android tools. It is a natural basis for a development environment. New Eclipse components can also be developed as part of the POSEIDON framework if necessary.

Achievement: Favoured platforms and frameworks are web, Android and Unity, which all have mature, feature-rich developer environments and large developer communities. We also provide tools based on Modelio (see sections 6.1 and 6.2), a mature platform for developer tools.

See also: Fr15

2.6.2 Fr11 – Documentation

Summary: Documentation must be provided, to enable project participants and third parties to develop POSEIDON components.

Discussion: Whether a framework component is reusing existing technology or produced in the project, the developer documentation needed to make use of it must be available. For instance, a communication API through which applications can integrate with the POSEIDON system must be well documented.

Achievement: Comprehensive documentation has been produced for all framework components produced in the project, and all other components chosen for inclusion have developer documentation. Produced framework documentation is found in this deliverable and D5.4, and publication on the project website is ongoing as this deliverable is finalised. A listing of technical documentation is maintained in delivery 5.6, which will be final at the end of the project.

2.6.3 Fr12 – Server-side pluggability

Summary: Allow new server-side services to be developed and plugged into a POSEIDON system by third party developers, giving a clean, documented and standards-compliant API to do so without knowing the inner workings of the framework or the other components in the system.

Discussion: This and the next requirement specify how the framework and resulting system will be extendable. It is not enough that it can be extended; it must be feasible to do so for third-party developers without prior knowledge of the system, and without needing to learn about the inner workings of the framework. To easily be able to plug in a new component, the developer must be given a well-defined interface for integration, and be insulated from the rest of the system. It is very challenging to make services pluggable to this extent – to be useful a service must be used by other components, but at the same time we don't want anything to depend on the service specifics directly. This is where a bus architecture will work well. A service can subscribe to messages based on what type of data it can process, and send messages on the bus for any interested clients.

Achievement: As described when discussing interconnection strategy in 2.2.6, a bus architecture has not been realised. This requirement is therefore only achieved in part – new server-side services can be added, but client applications need to know their APIs to use them. Integration of new server-side services is still very much facilitated, as the services of the infrastructure have well-documented APIs

[4] http://www.eclipse.org/

and data specifications, and authentication with the POSEIDON account can be added to any service by contacting the SmartPlatform instance.

See also: Fr13, Fr22

2.6.4 Fr13 – Client-side pluggability

Summary: Allow new client devices and applications to be developed and plugged into a POSEIDON system by third party developers, giving a clean, documented and standards-compliant API to do so without knowing the inner workings of the framework or the other components in the system.

Discussion: This is the client-side version of Fr12, and the same comments apply, except that pluggability is more straight-forward than for server-side services. As long as there is a well-defined API, and this is generic enough to support an open-ended set of services for the domain, this pluggability is achieved.

Achievement: This is achieved by having well-defined APIs on middleware and server-side services, as well as specifications for the data exchanged.

See also: Fr12, Fr15, Fr22

2.6.5 Fr15 – App components

Summary: Software components for connecting an app to a POSEIDON system will be made available for Android and iOS, to support POSEIDON app development on these platforms.

Discussion: In addition to making sure that all relevant platforms can be connected to the framework (Fr3, Fr14), we want to produce framework software components for Android as a primary app platform and for iOS as a secondary app platform. Framework software components for app platforms will take care of the connection to the communication infrastructure of the framework, and handle tasks all such apps will need to include, such as framework session/authentication and data retrieval and storage. This will provide a basis for developing new apps, giving the app-specific code a well-defined API to the provided framework implementation. A platform strategy must be selected (see Fr14). We can choose to either provide separate, native components for Android and iOS, or provide web technology components usable on both platforms (or a combination of both). If for native Android, the components will be implemented either as a jar-file or as an Android library project source code, and this will be freely available for third party developers. If using web technology, the framework can also include user interface components, so that apps developed by different entities as well as for different platforms can share a common look and feel.

Achievement: For Android we provide components for connecting to the SmartPlatform APIs, including authentication (described in section 6.5), in addition to the context middleware being available on this platform. We have also defined a framework for web applications (6.4), which is the only explicit support for other mobile platforms such as iOS. Native iOS components have not been a framework priority as the prototypes have focused the effort of native development on a single platform, and providing more extensive support for multiple platforms would have required much resources for little gain to the prototype system and pilots.

See also: Fr13, Fr14

3 Methodologies

In this chapter, we describe and discuss the development methodology defined for POSEIDON compatible applications. This methodology includes a requirements methodology, development process, ethical framework, and modelling tools for context-awareness. Tools based on the methodology are described in chapter 6.

3.1 UC-SDP User Centered Software Development Process

Through the decades in the software engineering community there has been a shift from the more structured and organized approach to more flexible ones that lately emphasize on speed. The creation of intelligent environments is different from traditional systems and the research is moving towards the best engineering techniques for developing this kind of systems. We consider that in this area there is a need of a more tailored process. Our approach does not have necessarily to change the way software is developed in the area; however it offers a guide for developers that will remind along the process of the important stages which define the ethos of the area. To increase the chances that ethics is embedded in the product being developed the U-C SDP explicitly includes it as part of the process. Underlying all this development there should be an ethical framework which should be taken into consideration from beginning to end of the process and at all stages there should be specific actions taken to ensure the ethical layer of the system is translated accurately from one stage to the other. The process can be supported by the ethical framework presented in Section “Ethical Framework”. We following introduce the different elements in Figure 2.

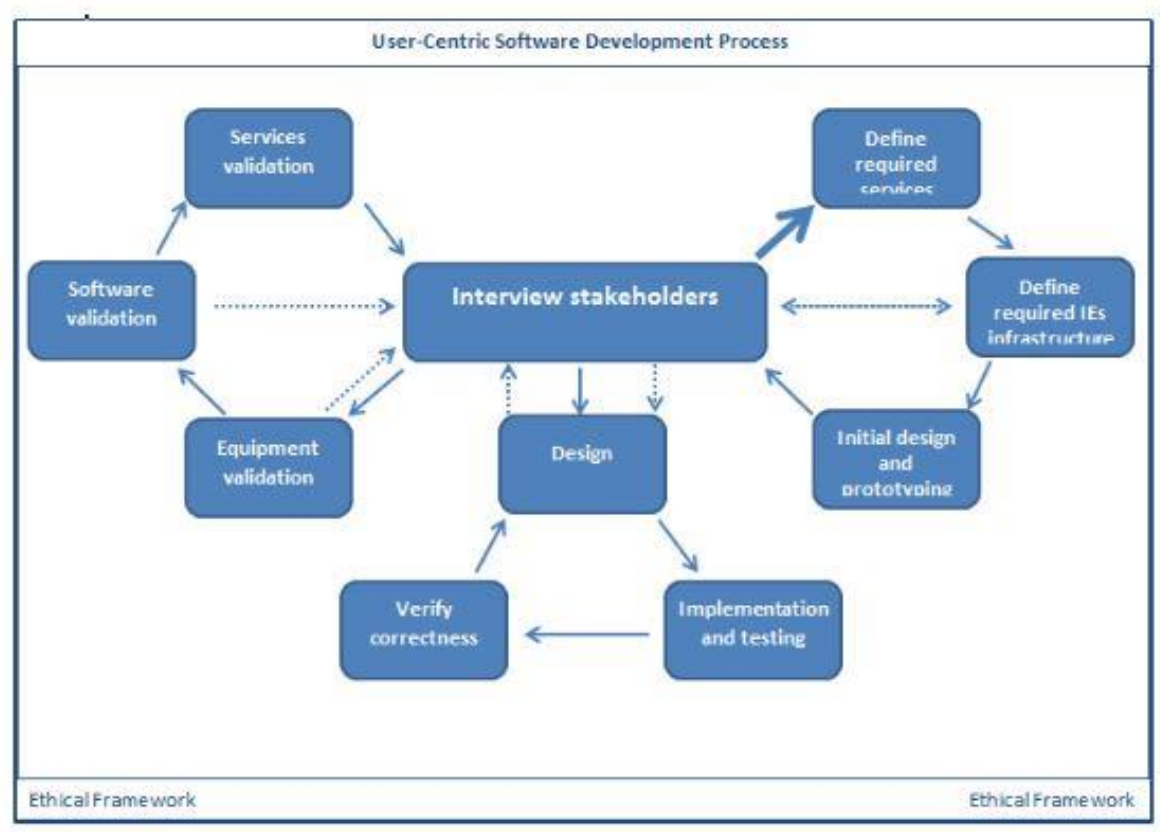

Figure 2: Main activities in the User-Centric Software Development Process

3.1.1 Initial development

Below we explain the initial development stages in the development of the UC-SDP (right side Figure 2):

1) Interview Stakeholders: each project should start by gathering the expectations of the stakeholders. The very essence of systems in this area is to serve humans. This principle shapes U-C SDP as through the different stages they can continuously monitor and influence the development with their opinions.

2) Define Required Services: the technical teams translate the information gathered from stakeholders into services the system will aim to provide.

3) Define Required IEs Infrastructure: the technical teams select the sensors, actuators and other devices and interfaces which will allow the materialization of the IE in the real world. The figure shows dotted arrows indicating it is suggested the technical team checks whether the proposed infrastructure to materialize the services is acceptable to the stakeholders and the stakeholders can accept or make alternative suggestions (which should be taken on board by the team).

3.1.2 Main development

Below we explain the main stages in the development of the UC-SDP (Low centre loop Figure 2):

1) Design: having the approval of the stakeholders at this stage implies a more detailed design analysis which should be strongly connected with the other stages of this cycle (i.e., create technical material which can feed testing and verification). Notice there is also a smaller loop between design and stakeholders indicating the desirability of making this step not an isolated stage but an interactive one with the stakeholders.

2) Implementation and Testing: this is about coding and testing that code. Testing should consider software, hardware and human-computer interfaces.

3) Verify Correctness: verification (e.g., through model checking) is one the most rigorous soundness checks available which can [8] and should be performed on systems in this area. Notice the dotted arrows to indicate verification and testing are complementary and should be used in conjunction to make sure the system built is correct.

4) Interview Stakeholders: stakeholders should be also involved on testing and approving the final functionality obtained through the interfaces and other aspects of the system they will experience.

3.1.3 Intelligent Environment Installation

Below we explain the main stages in the development of the UC-SDP (Left hand side loop Figure 2):

1) Equipment Validation: deploying starts with the infrastructure (hardware, network, devices and interfaces) and his step should have a separate safety and reliability check. Users can check if the infrastructure deployed is acceptable for them (location, maintenance required, and other practical aspects of its presence).

2) Software Validation: software is deployed on the infrastructure and the behaviour of the system can be experienced and tested by users. As in equipment testing the user can object on individual parts of infrastructure, here users can do the same on specific functions of the systems.

3) Services Validation: this involves the stakeholders experiencing the system for significant periods of time on a continuous basis (e.g., through Living Labs).

4) Interview Stakeholders: Their feedback after equipment or software testing goes to the development team. A problem in any of those system components at this stage can lead to redesigning and redevelopment of the system by going back to any of the other main loops.

After the system has been improved it comes back to another installation exercise. These main loops and secondary loops can be executed as many times as needed. The different combinations of paths can be exercised in a specific rigid order and with the full breath of services in mind from the very beginning, more like in the Waterfall model, or with smaller objectives in faster cycles more like in incremental and agile models. However, as most systems in this area are safety critical, the slower and more careful approaches are strongly advisable.

3.2 Ethical Framework

When developing AAL related applications, like POSEIDON , we recommend and expect applications to adhere to the eFriend Framework (Jones *et al.*, 2015). This framework was informed by the *Intelligent Environments Manifesto* (Augusto *et al.*, 2013) supporting the following principles:

- P3: Deliver help according to the needs and preferences of those who are being helped.
- P5: Preserve the privacy of the user/s.
- P6: Prioritise safety of the user/s at all times.
- P9: Adhere to the strict principle that the user is in command and the computer obeys.

The eFriend framework relies on the developer following these simple principles:

1. **Non-maleficence** and **beneficence:** Developed systems should be designed and developed to not cause any harm, particularly to primary users. Systems should aim to bring social benefits to users, by the increase in their quality of life.
2. **User-centred** and **multiple user groups:** During system design and development, views of all stakeholders, particularly primary users should be a central consideration. This process should help identify and accommodate the preferences and requirements of the user groups.
3. **Privacy:** Settings related to privacy need to be considered throughout all stages of design and development. The users of such systems should retain the ability to exercise control over monitoring, tracking, and recording activities in the systems. Users should be able to adjust privacy settings for different POSEIDON compatible services.
4. **Data Protection** and **Security:** All data collected in the process of running POSEIDON compatible services must comply with relevant data protection legislations in the territories in which they are consumed e.g. the UK Data Protection Act 1998. Users should also be capable to choose what personal information can be accessed, and how it can be used. Security is a responsibility you should take seriously, including maintaining safety and security of any collected, processed, or stored data.
5. **Autonomy:** Another important foundation for user trust. Primary users should be enabled to specify and adjust their level of autonomy. This can include the reconfiguration, customization, and overriding of components in the POSEIDON system, allowing the user to take control.
6. **Transparency:** It is important that primary users of POSEIDON compatible systems know and understand how different services can affect their lives in positive and negative aspects. This can be handled by making background tasks including surveillance more visible to the user.
7. **Equality** and **Dignity:** Developers should try to ensure the accessibility and affordability of devices, systems, and services to the primary user. Systems designed to be POSEIDON compatible should ensure social inclusiveness by accommodating different levels of cognition, competence and technical ability. These systems should under no except undermine user dignity, including stigmatising the user.

3.3 eFRIEND applied to the POSEIDON Project

The process of informing the project with the eFRIEND framework is an ongoing one. It started at an early stage with the initial requirements analysis, resulting in a specific set of requirements which map onto the eFRIEND principles, both directly and indirectly. The mapping of principles to requirements was supported by ad-hoc discussions within the group, some by email, some during development workshops. The project evolves through three prototypes of increasing sophistication. At each stage, these prototypes are assessed against criteria which include checking how faithfully the product fulfils these key requirements. These included basic Framework Requirements [FR], Functional [Fun] and Non-functional [NF] requirements, and various Hardware [H] and Design [D] constraints. Some of the relevant requirements and constraints are noted below.

Non-Maleficence / Beneficence

POSEIDON aims to enhance the welfare and quality of life of its target users by enhancing their autonomy, independence and social inclusion. It incorporates measures to avoid any risk of harming the user. The project has an Ethics Advisory Committee, comprised of experts on ethics, data protection and on the target users (representatives of Down syndrome associations in the participating countries). POSEIDON is developed and evaluated in accordance with the Charter of Fundamental Rights of the European Union (ECFR) and the Convention on the Rights of People with Disabilities (CRPD).

Multiple Users

POSEIDON is specifically designed for a multi-user environment and incorporates the needs and requirements of various stakeholders, including primary users (people with DS), secondary users (parents / carers) and tertiary users (for example, personal assistants, support workers, specialist teachers, healthcare professionals, employers and local authorities). The project acknowledges that these requirements and preferences may need to be balanced and/or prioritized, and that they may change dynamically over time.

User-Centricity

The POSEIDON project pursues a user-centric approach. It aims to develop assistive technology in joint collaboration with primary users and their carers at every stage of the development process. Primary and secondary users' wishes, values and needs are taken into account through detailed requirements gathering and analysis via surveys and face-to-face interviews with secondary and primary users. From this information, a clear understanding has been gained of primary users' living situations and daily living competencies, levels of proficiency using existing technology, together with the range of physical, sensory and cognitive difficulties they experience, including areas such as motor skills, speech, writing and learning disabilities. The POSEIDON system aims to address these challenges by providing context-specific help, information and intelligent assistance which is appropriate for different situations.

Reliability

Given that users may be dependent on the POSEIDON system outside the home, it must be robust, stable and reliable.

Fr17—When live, framework components should have robustness and fault-tolerance comparable to non-vital commercial systems

NF11—The system should be available 24/7, except for short periods of downtime for maintenance such as system upgrades

NF12—When live, system should be reliable enough so that its services are working and available at least 95% of the time

NF13—When live, maintainability should be such that the time to get the system restored after major failure is less than 1 day

NF14—When live, technical support should be available

Fun6b—Should provide comprehensive outdoors navigation services

Safety and Security

The use of a tablet device in a public setting by vulnerable users raises potential safety issues. Location and context awareness features will help the user to tackle difficult situations where they feel insecure or unsafe. Interfaces should provide a quick and reliable communication channel in order to call someone for help. Location-tracking via GPS and emergency connectivity enable carers to know the current whereabouts of their protégés, their previous locations, and enable them to check that they had reached their destination safely. Primary users, will be able to contact the carer if they get lost or have problems finding their way and need help.

Fr5—When live, support the safety of the end users

Fun2—System should provide immediate access to phone call

Fun5—System should keep track of user's position when travelling outdoors

Fun27—Carers should have possibility to request location of primary user

Fun28—Carers should be able to contact the primary user

H9—Device level access security should be present

NF5—Network level security for mobile component

Privacy

The results of the requirements analysis confirm that privacy is of high importance to potential users of POSEIDON and must be guaranteed in usage outside the home. POSEIDON accordingly aims to ensure that no user's privacy is violated. Users and their carers will have the ability to adjust privacy levels and to specify which personal data can be accessed and for what purposes it can be used.

Fr6—When live, support the privacy of end-users. Provide optional user privacy settings to enable customization

Fun10—Users should be able to decide on, and vary, the level of privacy at a specific point in time

Data Protection

While the effective use of POSEIDON makes it necessary to collect and analyse personal data to provide appropriate tools for different situations, data protection principles will be adhered to regarding informed consent for data collection, security of data transfers, controlled access to secondary uses of personal data, and storage of (un)necessary data according to specified time limits.

Fr9—Safeguard user data at the server-side with appropriate backup. Optional user settings to customize data storage requirements

Fun12—When live, users' information security should be protected

NF10—Context-related data should be stored for no more than 6 months

Fun11—Users should be able to decide the type of information stored in the devices used

Social Inclusion

One of the most important requirements to emerge from survey data was the facilitation of communication and socializing with others, in order to reduce the risk of social isolation that people with DS face, and increase their independence. Social inclusion was in turn found to be closely related to mobility and travel independence, a major factor in feeling independent and less reliant on others. The ability to travel independently has a direct impact on the opportunity to build up and maintain friendships beyond school or work into leisure settings. Despite some variation in levels of travel independence, people with DS tend to rely on relatives for transport with only a minority using public transport, or walking, alone. Particular difficulties that were faced include using maps, reading bus timetables, dealing with unexpected events while travelling (such as delays or cancellations in public transport) and finding alternative or unfamiliar routes.

POSEIDON aims to reduce this travel insecurity by providing support for independent, safe travel. Potential areas of assistance include: customised multimedia-rich navigation systems based on GPS to negotiate public transport, real-time context-related information including timelines, video clips, photos, recorded messages with reminders, suggestions, signals and warnings and trip-planning assistance. POSEIDON provides support for communication through access to various customised communication tools and platforms. It aims to foster social inclusion by facilitating and augmenting inter-personal communication rather than replacing it. The integration of the three main POSEIDON architecture components, and access to the system by tertiary users (in education, for example) will enable it to support educational needs, such as lesson planning, homework preparation, target setting and behavioural issues. For those in work, assistive technology will help by facilitating time management, organizational skills and the learning of job roles.

Fun7—System should be proactive (instead of reactive) in the following situations: issuing reminders in the areas where the primary user has indicated more help is welcomed (candidates: planning trips, during travelling from A to B)

Fun15a—First User-level contexts to be considered are: travelling, communicating

Fun15b—Second User-level contexts to be considered are: studying, working, well-being

Fun16—When live, ‘safety net’ plan for foreseeable situations (e.g. bus does not arrive and no connection)

Fun18—System should provide support for further social integration at leisure time

Fun24—Include photos of faces for known people in the communication tools

Fun25—Organize photos collection for social interaction and sharing, and support for other functionality, e.g. notable landmarks while in transit to aid orientation

Fun29—Carers should have possibility to request emotional state of primary user for two-way reassurance that there are no problems and to ensure primary user feels supported

D5—Give priority to plans involving public transport

D6—Trip planners should focus on the next few steps and use familiar landmarks to guide

Autonomy

People with DS can perform many everyday tasks and competencies, but often only with help which reduces their independence and autonomy. One particular area of difficulty is *time management*, including understanding time intervals, time orientation, remembering and keeping appointments, processing large amounts of information at once, following and remembering complicated instructions. Other recurring challenges include handling new situations or changes in routine, managing diet and making healthy eating choices, and remembering to take medication. Another area of difficulty is *handling and managing money*, in particular, understanding the relative value of

different amounts of money (which affects the ability to buy a product in a shop without help) and understanding the value of products when shopping. These difficulties are related to more general problems with numeracy and mental arithmetic, which in turn can affect confidence in going out with friends, and potentially further compounding social isolation.

The survey and interview data suggested a strong wish on the part of the target users to be more independent, and less reliant on carers and relatives. A high priority for POSEIDON, therefore, is to provide context-specific assistance to support autonomy and independence in the above areas. Enabling tasks to be completed independently without the need for assistance will potentially boost users' self-esteem and confidence.

Autonomy, however, as previously discussed, also means users being able to control technology. POSEIDON will be adjustable to individual preferences and personal needs. Users will be able to customize the system, within their framework of capabilities or with the help of their carers. While default settings will be provided, POSEIDON will include the ability to override those defaults. The system will allow for some functions to be switched on or off, in line with different needs and competencies. Functions do not have to be used all the time, or in situations where support is *not* needed. It is recognised that too many choices and functions working at the same time could make it difficult for the user. Users, ultimately, will thus have the ability to scale back, or turn off the system if they feel bothered by it.

NF1—System should promote user's autonomy and independence

Fr8—Support for optional interface customization to suit the end-user's needs

Fun9—The system functionality should be customizable

Fun19—The system should assist with activities supporting independence and integration

D7—Special consideration given to the way time is represented and communicated

Fun15c—Third User-level contexts to be considered are: socializing, healthcare, managing money

Transparency

To be in the control of the system, users needs to understand it's (re)actions, feedback and possible uses.

Potential weaknesses, limitations and vulnerabilities in the POSEIDON system will be made transparent to users, including system operations, data collection and use, and surveillance activities. An open source development environment will be created to attract the interest of different organisations, and enable compatibility with a wide range of applications and devices.

NF8—System should be open and transparent to users re: expected system functionality and weaknesses

Fr11—Documentation must be provided to enable project participants and third parties to develop POSEIDON components

Fr22—Extensible, allowing integration of new functionality not yet foreseen

Fun31—Provide confirmation that system has processed a request so user knows what is going on

Equality of access

POSEIDON upholds the principles of equality and fair access to technology. POSEIDON will be designed simply enough so that it can be used by the widest possible range of users with different potential levels of competence and cognitive ability. The system kits will be financially affordable and

available in various price categories with different payment options. Accessibility and inclusiveness will also inform design and usability. In tune with user requirements, the system will avoid the need for fast reactions, fine motor skills and manual dexterity. It will be generally symbol-based, rather than text-based, using gestural interaction where appropriate. POSEIDON will have an attractive design and user interface that is fun and simple to operate.

NF9—The system should provide help regardless of age and technical ability

H1—Cost of tablet should be less than €300

H2—Cost of virtual reality set less than €600

H3—Cost of adding interactivity to the table less than €300

NF17—Motivating to use

D2—Interface preferably based on symbol, icons and animations

D3—Take into consideration aesthetical features (colours, fonts, contrast, etc.)

D4—Consider design heuristics

It will be evident that some of the above requirements crossover more than one area and have an impact on several ethical principles simultaneously (context awareness, for example). What this does show, however, is that these principles can feed into, and map onto, concrete technical features and functions in any system. Having completed the requirements analysis, it is envisioned that this framework will permeate the remaining stages of POSEIDON's development. In the *design* phase, this will include the embedding of these ethical principles in the system architecture and functional specifications. This will be followed by their incorporation into formal methods, behavioural properties, and system agents in the *implementation* phase. Following the installation of the system, formal *verification* and *validation* of the system will ensure that its behaviour is consistent with the key ethical principles. In the *testing* phase, pre-pilot studies will be conducted so that the capabilities of the hardware and software used are fit for purpose and fulfil the requirements. This will be accompanied by usability testing with different prototypes, involving field trials and pilot tests from which detailed feedback will be gathered from users through a combination of workshops, diaries, reaction sheets and usage statistics.

3.4 R4C-AS – A Methodology for requirements elicitation in Context-Aware Systems

Context-aware systems (C-AS) are different from traditional systems to be created. Developers have to foresee the context in which the system is going to be executing in order to program services that can be useful to the users because they relate to their needs and preferences. The set of situations in which the system can offer services can be endless or unforeseeable. We have been exploring the best ways for eliciting requirements for C-AS taking into account the special needs of these kind of systems.

3.4.1 Methodology

The first aim of the methodology is to identify the stakeholder goals, preferences, needs and activities. The initial stages of the methodology can be described as a form of workout model, illustrated by Figure 3, whereby we introduce six core requirements gathering categories against which dedicated requirements elicitation activities proceed. They are not meant to be mutually exclusive or follow a synchronous process ow. Rather, they can take place concurrently/iteratively and are covered by two further activities that play a vital role: “operational support” and “harmonisation”. The first one is intended to determine technical support requirements for all aspects of the operational load: Delivery/set-up, customization training, system monitoring, review and potential upgrade requirements. The second one involves managing the overall requirements process and requirements specifications (including future review), particularly with the reference to a multi-disciplinary team that may or may not be distributed across different centres. In order to

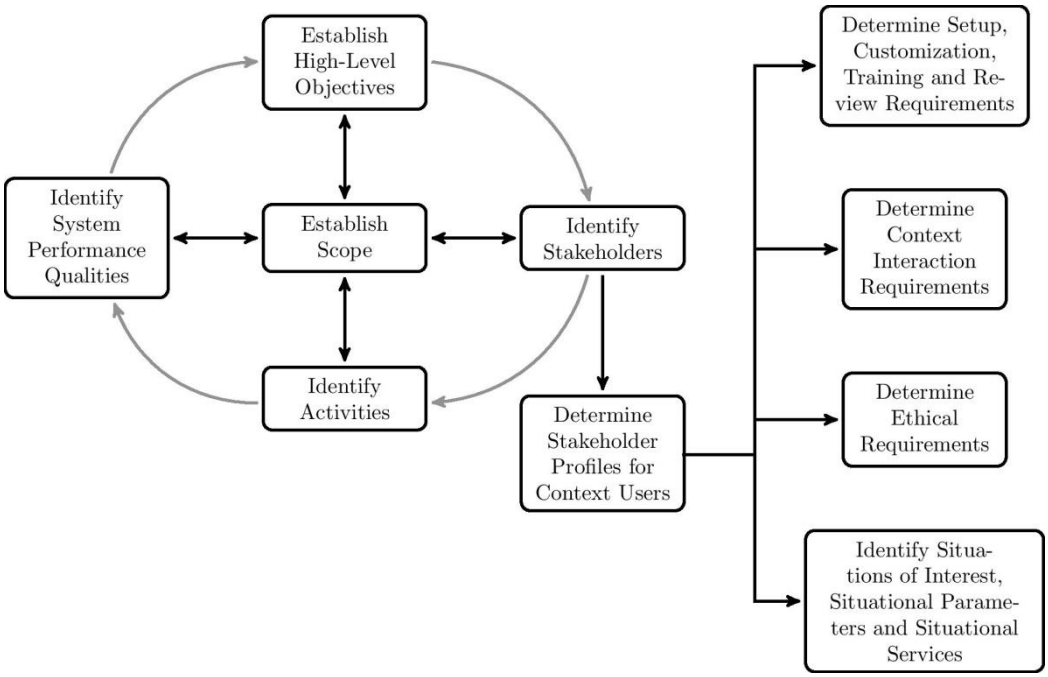

Figure 3: Core activities for the R4C-AS Methodology

illustrate the explanations, we will follow the example of the POSEIDON Navigation application. The reader has to understand that the activities described in this section are meant to be iterative, and do not follow a linear process. They all influence each other in an iterative way.

3.4.1.1 Establish Scope and High Level Objectives

These activities essentially refer to the initial stages of the requirements gathering phase. The aim is to specify the main objectives of the project so that all parties understand them. When it comes to establish the scope, the purpose is to determine the system boundaries. The developers decide what the system can and cannot do, specifying what is inside it and immediately external to it. As shown in Figure 3, establishing the scope is an activity affected by the main iterative circle. Table 1 shows an example of the scope and the high-level objectives for the Navigation Application example.

Scope	**High-Level Objectives**
Mobile Application	Foster the independence of People with Down’s Syndrome when displacing outdoors
Based on Maps	

With tailored notifications to guide users depending on context	

Table 1 Scope and High-Level objectives in the example of the Navigation Application

3.4.1.2 Identify Stakeholders & Profiles

The following step is to identify the stakeholders of the project. Table 2 illustrates the stakeholders involved in the Navigation Application.

Stakeholder	**Description**
Primary Users	People with Down syndrome
Secondary Users	Parents or carers of people with Down syndrome
Tertiary Users	Teachers or supervisors of people with Down syndrome
Calls Provider	Company that provides phone calls and SMS to the mobile device
Internet Provider	Company that provides internet to the mobile device
Device Manufacturer	Company that manufactures the device
Operating System Developers	Group involved in the development of the operating system of the device
Maps Library Developers	Group involved in the development of maps libraries

Table 2 Stakeholders involved in the example of the Navigation Application

3.4.1.2.1 Profiles

To complete this activity we need to create different profiles of stakeholders in order to have a better perspective on what services can be offered by the system. The reader should bear in mind that not all the stakeholders will require context profiles. In order to identify different profiles that there might arise; there is a need to identify the requirements for a certain stakeholder by using the following four sub-activities:

1. Set-up, customization, training, monitor the system and review its use: It is about determining the technical support a specific profile will receive. Examples of these requirements can be: 1) When live, maintainability should be such that the time to get system restored after major failure (such as re at a server location or database corruption due to server failure) is less than 1 day; 2) When live, technical support should be available; 3) When live, first set up time by supplier should be less than 1 hour; 4) When live, system upgrades should be achieved in less than 2 hours

2. Interaction design issues: The context interaction requirements for the primary and secondary users. Psychological questionnaires can be used to assess individual skills (cognitive, physical and perceptual). In addition, prototyping will be a key approach to ensure the context interactions are carried out and work as intended.

3. Ethical requirements: It is about determining the ethical requirements for a specific profile. This is done by using the e-FRIEND framework.

4. Identify Situations of Interest, Situational Parameters and Situational Services: It is related to identifying the situations in which the system can provide services, using the information about the profiles and the activities of the different stakeholders. This sub-activity is further explained in Identifying situations of interest, situational parameters and situational services.

3.4.1.3 Identify Activities

During this stage the aim is to determine the activities that stakeholders perform in their daily lives, in order to identify how can the system assist them in those activities. The process can be supported by scenario-based methods. For example, Figure 4 illustrates the activities that are generally involved when moving from one place to another by bus.

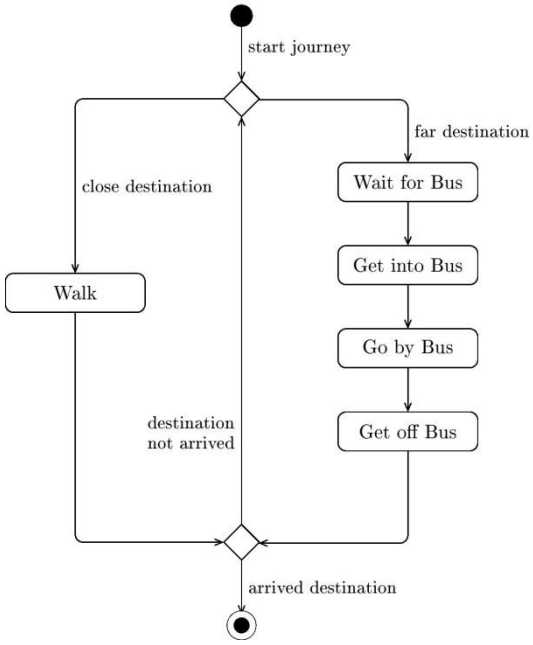

Figure 4: Main activities involved in displacement by bus

3.4.1.4 Identifying System Performance Qualities

System Performance Qualities are related to the gathering of requirements that can be used to judge the operation of a system, describing how it has to work, rather than citing specific behaviours which describe what the system should do. This idea is similar to the classical non-functional requirements, but it explicitly considers them as quality attributes, recognizing that they could stem from functionalities of the system. The process can be supported by analyzing the high-level objectives as it can be observed in Table 3 an example of how to identify system performance qualities by using the high level objectives of the stakeholders..

Stakeholder	Goals	Sub-Goals	Requirements
Primary User (PU)	Increase their independence from SUs when moving from one place to another (outdoors)	Guide the PUs through reminders of what they need to do next	Receive instructions that are able to understand, taking into account possible visual and auditive insensivities
Secondary User (SU)	Reduce the attention that PU require on them when they have to do displacements outdoors	Show the SU that the PU is safe	The system has to enable the communication between the PU and the SU anytime
			The SU can request the location of the PU

		Show the SU that the PU is doing what is meant to do	The SU can know when a PU has finished a task

Table 3 an example of how to identify system performance qualities by using the high level objectives of the stakeholders.

3.4.1.5 Identifying situations of interest, situational parameters and situational services

In this subsection we will be using the following concepts:

- Situation of Interest (SoI): Circumstance in which the system can potentially offer a situational service.
- Situational Service: A system functionality that is triggered in a certain circumstance and is directly or indirectly related to the preferences and needs of the stakeholders in that specific circumstance.
- Situational Parameter: Is a measurable factor forming one of a set that can identify a certain situation of interest.

During the previous stages of the methodology, the aim is to identify the stakeholder activities, goals and needs. This information eases the identification of Situations of Interest (SoI). The first step analyses the previously identified activities of the stakeholders in order to discover different SoI. The process can also be complemented by scenario based techniques. The SoI do not have to be relevant to all the stakeholders. They might be relevant only for one of them or even to a single stakeholder profile. The concept of SoI, enables a deeper analysis on: 1) how the situation is going to be detected (Figure 5-a) and 2) what actions can the system take in those situations (Figure 5-b).

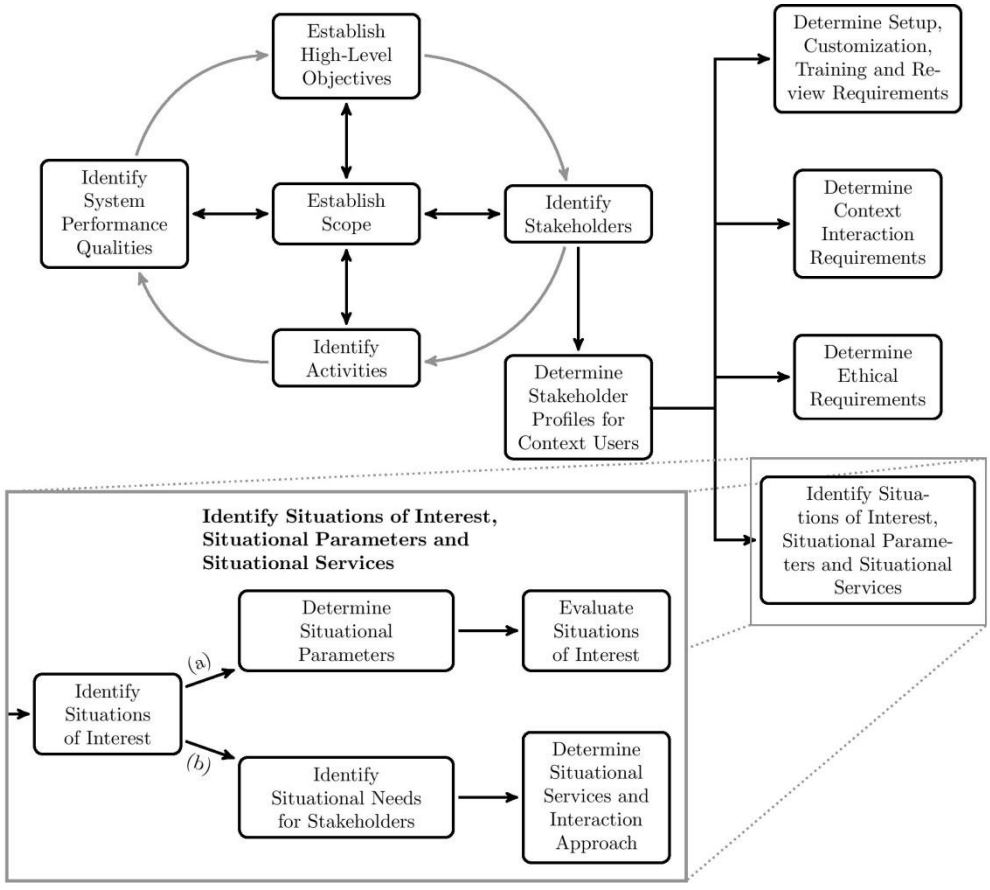

Figure 5: Figure describing the inner workings of the activity "Identify Situations of Interest, Situational Parameters and Situational Services"

3.4.1.5.1 Identifying Situations of Interest

The first step inside the extension is to use the knowledge obtained in previous stages for identifying Situations of Interest (SoI). During this stage scenario based techniques can be used. We will illustrate the example by using the activities identified in Figure 4 about travelling by bus. Table 4 summarizes the situations of interest in that example.

Activity	**Situation of Interest**
Waiting for bus	The user has just arrived to the bus stop
	The bus arrives to the bus stop
Going by bus	The bus arrives to the desired bus stop
	The bus arrives one stop before the desired one

Table 4 Situations of Interest for the ``Waiting for bus'' activity from Feil! Fant ikke referansekilden.

3.4.1.5.2 Identifying Situational Services

Once the situations of interest have been identified, the first step is to think about the needs of the stakeholders in those situations. This can be complemented by the information gathered about the stakeholders in previous stages as well as by the use of questionnaires. The next step is to determine which services or functionalities will be executed in each of the identified situations, taking into account the preferences and needs of the stakeholders (**Feil! Fant ikke referansekilden.**5-b). Once the service is identified there is a need also to de ne how the service will interact with the specific profile of users. There are two ways in which a service can be provided: A) Active, where the system changes its content autonomously; B) Passive, where the user has explicit involvement in the actions taken by the system. For example by asking permission for a service to execute, or showing a list of options. Table 5 illustrates the situational services and their interaction with the users that can be provided in some of the situations of interest identified.

Activity	**Situation of Interest**	**Situational Need**	**Situational Service**	**Interaction Type**
Waiting for bus	The user has just arrived to the bus stop	Know how much time they have to wait	Notification showing the time remaining for the next bus	Active
	The bus arrives to the bus stop	Know if the bus is the one that they need to embark	Notification informing the user that the bus that has arrives is the one that needs to embark or not	Active
Going by bus	The bus arrives to the desired bus stop	Know when to disembark	Notification reminding the users that they need to get down at this stop	Active
	The bus arrives one stop before the desired one	Know when they need to press the stop button	Remind users that they need to press the stop button	Active

Table 5 Situational services and interaction types for the situations of interest identified in Table 4 Situations of Interest for the ``Waiting for bus'' activity from Feil! Fant ikke referansekilden.

3.4.1.5.3 Identifying Situational Parameters

After the identification of the SoIs, the requirements engineers need to agree upon which parameters could make the system detect a certain SoI. First, a text based description on how the system will detect that situation will be provided. Based on this, the situational parameters will be identified which at the same time provide information about the sources of information that the system will use to identify such situation. The use of descriptions and parameters enables the participants in the project to communicate among themselves using a language that can be understood by all, without describing the inner workings of the system. The next step, as it can be observed in Figure 5-a, is to evaluate the feasibility of the system identifying the SoIs. For this purpose, there is a need to perform a preliminary evaluation of how the system is going to measure the different parameters, trying to identify their source. This evaluation has to be done for each of the SoIs identified. The idea is to perform a light analysis that can help to evaluate, until a certain extent, the feasibility of the system detecting a SoI. During this evaluation some situations of interest might be discarded if requirements engineers consider the situation not worthy to implement for different reasons. The reasons could be (among others): 1) A considerable error in the way the SoI is detected; 2) The cost of the implementation versus the value added by the system taking the actions proposed in those situations. The objective of this evaluation is not to accurately determine all the possible sources but to make engineers think about the feasibility of implementing what they are proposing. A deeper thought about the sources of each parameter will happen in further stages of the design. Rejecting to implement a SoI at requirements elicitation stage of the development can save time and money. The later the team realizes that a situation cannot be implemented the higher the costs will be. Table 6 gives an example of the parameters and sources for the Navigation Application example.

Activity	**Situation of Interest**	**Identification Description**	**Situational Parameter**	**Source**
Waiting for bus	The PU arrives to the bus stop	The PU location coincides with the bus stop location	PU Location	PU Mobile device's coordinates
			Bus Stop Location	Bus stop coordinates
	The bus arrives to the bus stop	The time is approximately the time when the bus was scheduled to come	Bus Arrival Time	Bus line arrival time schedule
			Current Time	PU Mobile device's time

Table 6 Situational parameters for some of the situations of interest identified in Table 4 Situations of Interest for the ``Waiting for bus'' activity from Feil! Fant ikke referansekilden..

3.4.2 Diagrammatical Support

3.4.2.1 SysML/UML

UML is the standard from OMG [5], modelling language used in the software community. Even though UML should be sufficient to address Systems Engineering needs through its wide range of notations, it is recommended to adapt it with extensions defined with "UML profiles". SysML is based on UML and involves modelling blocks instead of modelling classes, thus providing a vocabulary that's more suitable for Systems Engineering. A block encompasses software, hardware, data, processes, personnel, and facilities. As specified on the following diagram, SysML reuses a subset of UML2 (UML4SysML), and defines its own extensions. Therefore SysML includes nine diagrams instead of the thirteen diagrams from UML2, making it a smaller language that is easier to learn and apply. SysML can be easily understood by the software community, due to its relation with UML2, whilst it is accessible to other communities. SysML makes it possible to generate specifications in a single language for heterogeneous teams, dealing with the realisation of the system hardware and software blocks. Knowledge is thereby captured through models stored in a single repository, enhancing communication throughout all teams. In the long term, blocks can be reused as their specifications and models enable suitability assessment for subsequent projects.

3.4.2.2 SysML Requirements Diagram

According to the OMG, both Systems Engineering and the software industry use requirements to formalize the stakeholders' needs, which will be realized as functionality and constraints, satisfied by the delivered application or system. They present a model based approach that makes use of requirements through dependency associations with elements from the model such as use cases, blocks, or test cases, establishing the model traceability. SysML specifications define the requirements modelling, by taking current features from the tools currently available on the market. Hence SysML define a visual and graphical representation of textual requirements, specialised associations between themselves or with other elements of the model, and how they can be managed in a structured and hierarchical environment. SysML defines new types of associations (stereotyped dependencies):

- Derive : one or several requirements derive from a requirement
- Satisfy: one or several model elements fulfil a requirement
- Verify: one or several model elements (e.g. a test case, verify that the system fulfils a requirement)
- Refine: one or several model elements (e.g. a use case, further refine a requirement)

SysML defines new types of comments, introducing stereotypes, enabling the association of explanations with associations or model elements:

- Problem : comment which description specifies the identified problem or need, following a deficiency, limitation, or failure of one or more model elements
- Rationale: comment which description provides the reason or justification on the decision related with the association or element

[5] Object Management Group (OMG) is an international, open membership, not-for-profit technology standards consortium. They present SysML requirements diagram. Source: http://www.omgsysml.org/SysML_Modelling_Language_explained-finance.pdf

3.4.2.2.1 SysML Requirements Diagram Extension

We have extended SysML with two new types of relations:

- *ContextualRqt*: Refers to a relation between a requirement and another requirement which is related to the situational service explained in Section “Identifying situations of interest, situational parameters and situational services”.
- ContextDependency: Refers to the relation between a situation and a requirement

We have also extended it with one new type of element:

- Situation: Is an element similar to a requirement that describes a situation of interest as explained in Section “Identifying situations of interest, situational parameters and situational services”.

Figure 6 shows an example of our extension.

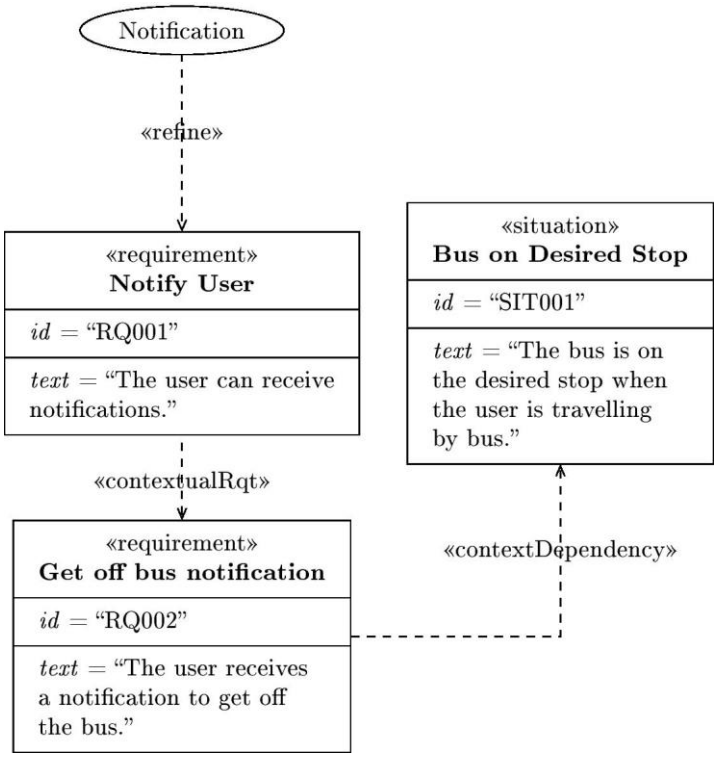

Figure 6: Example of the extension for the SysML requirements diagram

3.4.2.2.2 Context Dependency Diagram

We have created a new diagram that relates the situations of interest to how the system is going to identify the context as explained in Section “Identifying Situational Parameters”. We introduce a new element “Context Variable”, an extension of a UML Class. We also introduce a new extension of a relation called “identify” which joins a situation of interest with a context variable. All the extensions can be observed in the meta-model (Figure 8).

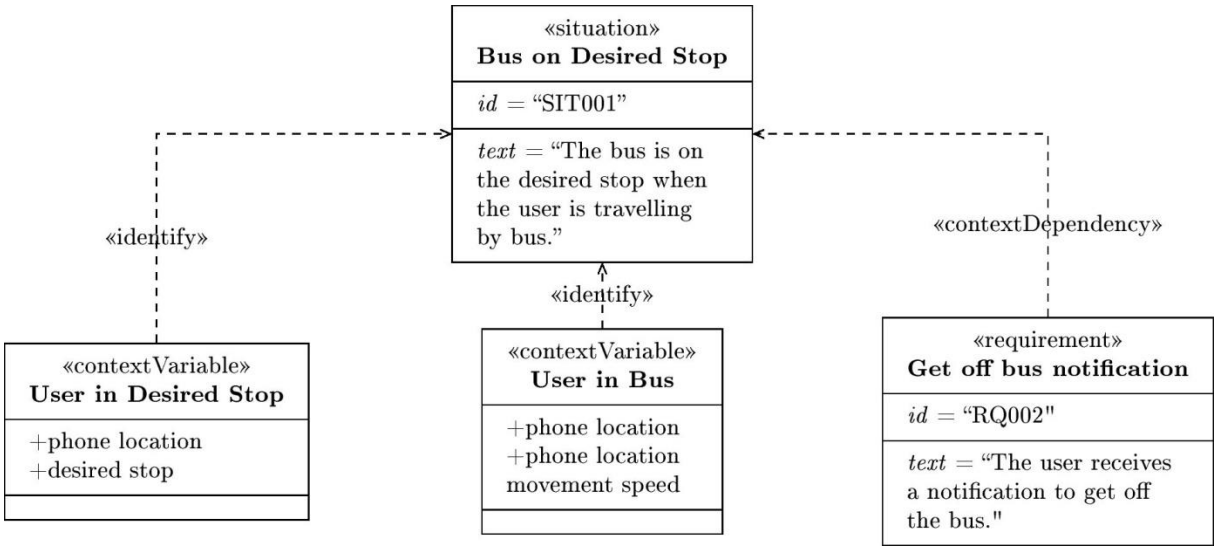

Figure 7: Example for the extension of SysML for the context dependency diagram.

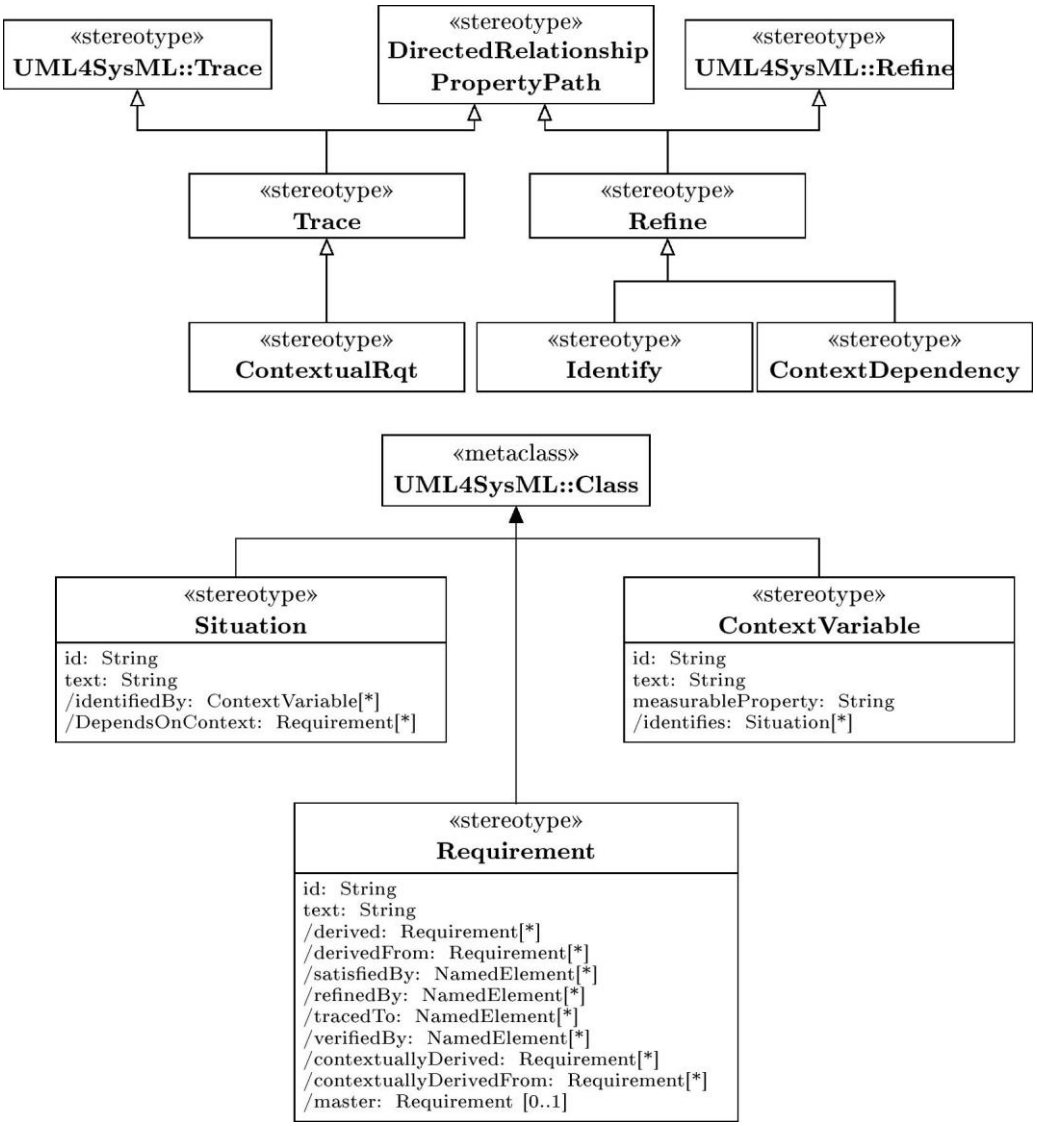

Figure 8: Metamodel of the SysML Extension.

4 System architecture

This chapter describes the technical infrastructure of the POSEIDON system, platforms for client applications, and how this framework is used to build a system of applications to help people with Down syndrome and their secondary and tertiary users. Fundamental to the architecture is the functionality it needs to support (section 4.1), where the framework mainly provides server-side infrastructure, and the client platforms (section 4.2). The resulting architecture is described in section 4.3. As personalisation is a key aspect of the POSEIDON system, section 4.4 gives an overview of how personalisation is supported by the framework.

This chapter presents the final architecture, which is the result of several iterations throughout the project. The earlier iterations are described in appendix 1.

4.1 Supported functionality

The process of defining a technical infrastructure for POSEIDON has been an iterative one, iterating with the prototype development. As it was not clear from the outset what the framework would be, the chosen strategy was to set in place the most essential infrastructure and start the prototype development, to later generalise parts of the prototype system into the framework for the benefit of other developers. So the functionality supported by the framework is based on the prototypes developed in the project, which in turn is based on the end user requirements. These requirements came from the preparatory work and the Description of Work, where needs for activities of daily living (ADL) are summarised in a scenario, and were further refined in the early work in the project (Work Package 2).

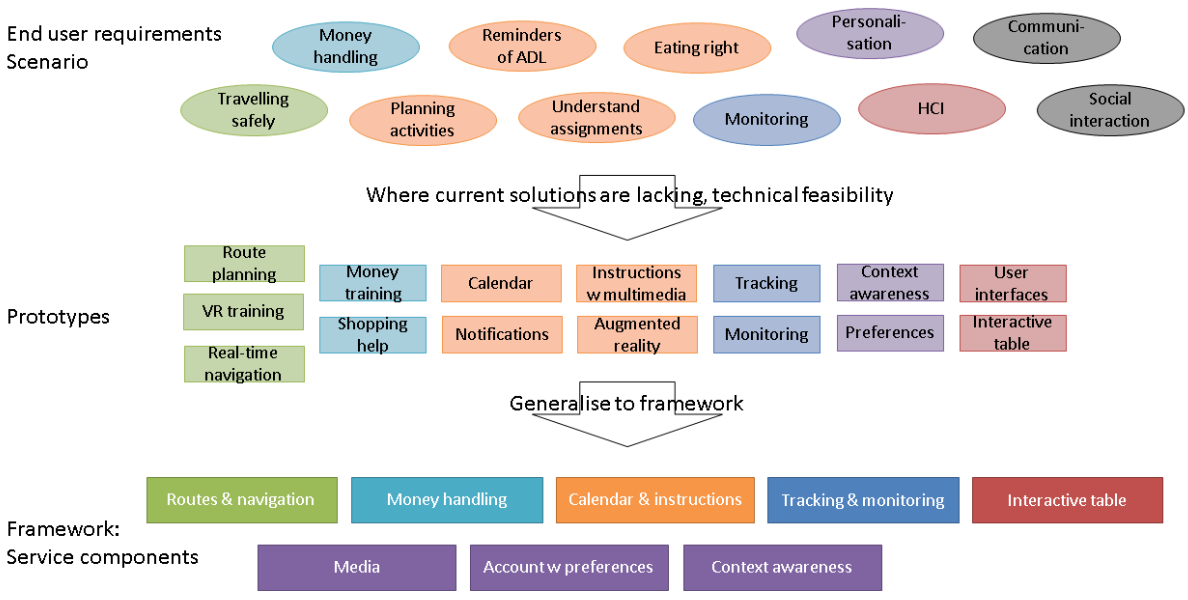

Figure 9: Framework functionality development process

The process is illustrated in Figure 9. The top third of the figure are the most important requirements found to be useful for supporting the user group in the ADL scenario. Below this we have a summary of the prototype functionality which has been developed in the project to address the requirements. We have developed prototypes providing functionality that covers most, but not all, of the requirements. Our focus has been where current ICT-based solutions are lacking, and where it was

technically feasible for us to provide better solutions. In particular, we chose not to focus on communication, because most DS users have no problems using a phone for this purpose. Other social interaction is also only supported indirectly by POSEIDON functionality developed so far, such as through the aid in travelling safely and handling money.

A technical infrastructure of service components has been defined and implemented based on the prototype work, extracting general aspects from the prototypes into something which is shared between the applications and which will enable other applications to integrate with this system. The following sub-sections describes the various functional areas and the infrastructure needed to support them. This is part of the basis for the architecture described in 4.3, with the resulting infrastructure components described in chapter 5.

4.1.1 Routes and navigation

One area of the scenario we have worked on extensively in the POSEIDON project, is to help the primary user travel safely. We selected this area because mobility is fundamental in having social interactions and in participating in anything outside the home in an independent manner. It is also an area where current solutions are lacking, as route planners such as Google Directions and the applications using such services have low usability for people with Down syndrome. And it is an area where we had the necessary resources and skills within the project to develop interesting prototypes.

Our prototype solutions have three main parts. Firstly, the secondary users need to be able to plan routes, adding personalised instructions for their primary user. The primary user can do VR training, moving through the route at home to get to know it. Finally, the system provides real-time navigation assistance on the road.

One fundamental question we have explored in the project, is what a route is and how it is created. For the different applications to interact, and for third-party applications to be able to interact with this part of the system, the framework must at least define a format for route data. The most simple route specification we can imagine is to define a destination, so that we can point out the general direction of movement to get there. Then we can add intermediary points, because we can rarely travel in a straight line and because instructions are needed along the way. One question is how fine-grained the route needs to be, such as if it needs to follow every curve in every street.

Generally, route data can be created in two ways: (i) manually (drawing the route on a map and maybe add textual instructions at key points, or go through the route with a position tracker and use the recorded track), (ii) by using a route planning service which can find routes between given points based on map data. Both approaches have advantages and disadvantages.

Configuring a route manually requires effort and a tool for this purpose. A main benefit is that a person will be able to create a custom route that suits the primary user. A main disadvantage is that the route is static, and only works from the pre-defined starting point. On the other hand, the advantages of automated route planning are that it requires minimal effort from users, and it is able to create a route from and to any place connected to known streets. The secondary user could create a list of destinations of interest for the primary user, who can start navigation based on the data the planner server generates from the current location. Thus, we can easily adapt and provide a new route to the destination every time the user deviates from the route.

As creating and managing an advanced, automated route planner was outside the scope of the project, we looked at existing route planners. The quality of a route planner depends both on the algorithms it uses and on the data it has available. The best route planners can also use public

transport route information to produce multi-modal routes, which include public transport with time schedule.

Directions, one of these route planning services, is implemented by Google as part of its Maps suite[6]. Google Directions can create routes for various transport modes. Google Directions is freely available through a web API, so it is possible to be used in own applications displaying the routes. However, their terms of use state that they do not allow usage of their routes for real time navigation or guidance. OpenTripPlanner[7] (OTP) is an open-source platform for route planning web applications. An OpenTripPlanner server instance can be used through a web GUI with map, or through a web API. For map data it uses OpenStreetMap. OTP appears to be functional and mature enough for real-world use. It can do multi-modal planning with public transport schedules, based on data in the same format as that used by Google (GTFS). Providing the route planner service with OTP requires setting up and managing a server with the OTP system where the data must be kept up-to-date. The OTP instances in use are usually run by transport companies and only cover their city/area. There are many trip planner services available online which are provided by transport companies and others to stimulate use of their services. We didn't find any OTP or other open planner instance we could use.

Google Directions was the best route planner service available. The two main problems with using routes produced by this service, is that they lack any personalised instructions and that the terms of service does not allow usage in real-time navigation. We tested a hybrid approach: the route planning of the secondary user used the Google Directions route as a basis, adding personalised instructions with images. After pilot testing, the conclusion was that routes produced by Google Directions do not always take the path the secondary user would select for the primary user. The automated service may choose an unsafe street, and does not know about shortcuts and other paths which doesn't follow the street map. We therefore concluded that manual route creation is needed for safe, personalised routes. This means that the person creating the route must select a set of intermediary points and provide instructions for each. However, we do not preclude all use of automated route planners. If the user deviates from the planned route and needs help getting back to it, a route planner may provide data which can be used to help the user in this limited case.

The resulting framework infrastructure needed is a clearly defined data format for routes, and a place to store route data so that all applications (route planning, VR training and real-time navigation) can access the same data.

4.1.2 Money handling

Being able to pay for goods and services is another fundamental activity of daily living. In addition to going to shops, restaurants, etc., being able to pay for transport can be necessary for travel. As the training aspect was seen as the most important, this was the first focus for prototype development. The interactive table was selected as an input device for this, as physical money can be moved around on the large surface. For the full solution, the same three factors as those for navigation is needed: setting up by the carer, training at home, and offering assistance "in the field".

The selected underlying model to support these activities is that of a shopping list – a list of products with prices. There needs to be an application for secondary or tertiary users to define products and create shopping lists, primary user functionality for training to pay for the products in the shopping list at home, and assistance in buying the products on the mobile device. The infrastructure must connect the three together. The framework must provide specification for how to define products

[6] https://developers.google.com/maps/documentation/directions/
[7] http://www.opentripplanner.org/

and shopping lists, as well as a place to store the definitions and images of the products. The interactive table is also a useful component for training applications.

4.1.3 Calendar and instructions

Two other important aspects of supporting ADL is to be able to plan activities in time and then get notified at the right time, and to provide instructions and other information for various types of tasks. We have grouped this together, as instructions will often be linked to planned activities, and a notification for an activity may need to explain the activity to the user. Indeed a sort of calendar service is essential, and there should also be possible to link it to other functionality, so that starting navigation by a route can be scheduled for a specific time or the primary user can be notified to prepare the correct amount of money for a shopping list.

Our prototypes include calendar interfaces for both primary and secondary users. Again the secondary user is mainly responsible for providing content – entering events in the calendar. The primary user needs reminders. But primary users should also be able to enter and change events themselves. In any case, both primary and secondary user applications need access to the same calendar data.

We have added the concept of an instruction list to the calendar event. This is a sequence of instructions, where each one can have text, image, recorded sound and video. These should be presented to the primary user at the time of the event, but also be available at other times.

In addition to instructions tied to calendar events, easy access to a personalised set of instruction videos was wanted. One prototype functionality is therefore for the secondary user to define a list of videos, and for the secondary user to have this list available on their mobile device.

An augmented reality prototype was developed as part of the first, exploratory set of prototypes. Here the primary user is aided in preparing what they need to bring when they go out, based on a calendar event, by overlaying information on what the device sees through the camera. This idea was not developed further, as other functionality was prioritised.

Calendars are well known in ICT, and we have used Google's calendar service as cloud storage for events in our prototypes. iCalendar[8] is the closest thing to a standard for exchange of calendar data, and it embodies the conceptual model of planned events in time with notification etc. However, this basic calendar event model is not by itself sufficient for our needs, as we need to include multi-media instructions and links to other concepts in our framework such as routes for navigation.

Our infrastructure needs a calendar event format, based on existing formats but with additional POSEIDON information such as instruction list. It needs a cloud storage of the events, and of associated instructional media, so that these can be exchanged between applications. And we need a data format for the video list.

4.1.4 Tracking and monitoring

The monitoring aspect in Figure 9 means that a secondary user should at times be able to check where the primary user is, and also check other information from the mobile device of the primary user, such as battery level. It entails on the one hand that the primary user must be tracked, and on the other that the information is available to an authenticated secondary or tertiary user. This is related to travelling safely – someone who may get lost can more safely move independently if they can be tracked. It is a functionality which benefits both primary and secondary user – the secondary

[8] http://en.wikipedia.org/wiki/ICalendar

user has less reason to worry and can perform a form of care remotely, and the primary user can be allowed more independency since they can be located if a problem arises.

Some form of tracking has been seen as fundamental from the start. Tellu's SmartTracker service, which provides a complete server-side infrastructure for tracking applications, was described as part of the framework in the Description of Work. Its role in the technology infrastructure is detailed in chapter 5. The framework should also enable primary user applications to send data into the SmartTracker service, and for secondary user applications to access the information.

Privacy and security are important features of the framework when it comes to tracking. It should be possible for primary users to control when a carer can see where they are. And the information must of course only be made available to the correct user, based on authentication of identity.

4.1.5 Supporting functions

What we have described so far are the main categories of end user functionality provided by the prototypes, to be supported by the framework. The rest of the framework components indicated in Figure 9 can be seen as support functions, needed to support the functionality or wanted to enhance them. These are:

- Interactive table: The interactive table is a new interactive device, described in the Description of Work and further developed and tested in the project. It is used for interacting with the money and navigation training applications. We consider the table itself and software components needed to connect to it as part of our infrastructure.
- Media: We have seen that all of navigation, money handling, calendar events and instructions need media. The infrastructure must therefore provide storage of and access to media. Images, sound and video is of particular importance to this user group, as not all are good readers. Media other than text is important to provide instructions.
- Context awareness: It is a goal in the POSEIDON project to add context awareness to the system, so that primary user applications can take the context of the user into consideration when offering assistance. Support for this also needs to be part of the infrastructure.
- Accounts & preferences: For a system where users access private data in multiple applications, with data stored in the cloud, there needs to be user accounts for defining who the users are. Infrastructure is needed for users to authenticate themselves. Personalisation is one of the requirements, and in addition to instructions and other content being personal our prototype system allows primary and secondary users to set preferences for how the applications work. Preferences also need to be stored for the user account, so that they can be shared between different applications.

As our prototypes provide user interfaces specifically designed for the target group, we will also mention user interface guidelines and graphical components are part of our framework. These are not discussed here as they are not part of the architecture or considered technology infrastructure components.

4.2 Client platforms

This chapter discusses the platforms our architecture supports for client applications. The project has a goal of supporting services on multiple platforms. Firstly, we categorise platforms as either stationary or mobile. Stationary platforms in our case is laptop and desktop computers, with their input and output devices. With mobile platforms we mean devices you may use while out walking. This includes smartphones and tablets, although there is no clear distinction between tablets and laptops today, and indeed not all will be comfortable using tablets in this way. We make this

distinction not just because it is convenient and a common way to categorise computing devices, but because we distinguish between the stationary use at home or in other fixed locations, and the mobile use when moving about. The first allows screens of some size and special input devices such as the interactive table, while the second is always available. For the functionality we have described in the previous section, the preparation and training aspects should primarily be available on stationary platforms, which are primarily done at home and benefit from a larger screen and input devices such as keyboard and interactive table. The guidance and notification aspects must be available at all times, and therefore need to be available on mobile platforms. So while it is a goal for the prototype applications that all data and all primary and secondary user functionality should be available on both mobile and stationary platforms, different aspects of the primary user functionality can be available on stationary and mobile devices.

On another level, what is available on stationary platforms should be available on as many stationary platforms as possible, and the same applies to mobile platforms. This is a question of multi-platform strategies, which is an area we have investigated in the project and want to address with the framework. Platforms will be discussed separately for mobile and stationary in the following sub-sections, but first we give an overview of what we see as possible strategies for supporting multiple platforms:

- **Native applications:** This is the traditional way of building an application for a specific operating system, which means that supporting multiple platforms means targeting each platform individually. It is the only way to fully exploit the possibilities in each platform, but it means building a separate codebase for each. One way to support this strategy with the framework is to supply framework components for multiple platforms, such as for connecting an application to the SmartTracker service. Application developers could then use the supplied components to develop their own POSEIDON-enabled applications.
- **Web applications:** A web application is an application which runs in a web browser rather than in a specific operating system. It is built on the web technologies of HTML, JavaScript and CSS. Built the right way, a web application should be able to run in the browsers of all modern platforms. A responsive web application is one which can adapt to different screen sizes, so that the same application can be used on both stationary and mobile applications. Web applications are seen by some as the future of multi-platform applications, but they are still limited to mainly being a user interface layer, connected to online servers for data. So it is not a strategy for all types of applications.
- **Java applications:** This strategy is to write applications in the Java programming language, for running on the Java Virtual Machine (JVM). Java was designed specifically to support a multi-platform strategy, but with the advent of web applications and the lack of JVMs on mobile platforms, it is rarely used for consumer applications. It remains a possible strategy for stationary applications, and it should also be noted that the Java programming language and basic APIs are used for Android applications. While Android has its own alternative to the JVM not compatible with regular Java bytecode, and its own APIs on top of the basic Java ones, Java code can be reused for both JVM and Android applications.
- **Web-native hybrid:** Web technology can be combined with native components for a hybrid strategy, where at least the user interface is platform-independent. The mobile platforms allow wrapping a web application as a traditional app which is installed locally. It can then be connected to back-end, native components, such as for background processes which are independent of the web UI. There are also frameworks to facilitate developing this type of

application, such as Apache Cordova (formerly PhoneGap)[9]. Cordova can be seen as a multi-platform framework in its own right, also fitting in the last category of this list, as it defines its own platform and then maps this to the various target platforms using native-specific components for each platform. While a hybrid strategy may at first seem promising, it is as much a combination of drawbacks as it is of strengths. Both web technology and frameworks built on it have their limitations compared to native applications, so you either have to stay within the boundaries or make your own platform-specific adaptations.

- **Multi-platform frameworks:** Such a framework constitutes its own platform and then map that onto each of the set of supported platforms. The developer writes source code for this framework platform, and its tools compile this single source to run on various platforms. Cordova is one such framework, though we put it in the previous category as it is based on web technology. Other general-purpose multi-platform frameworks include Xamarin[10] and Qt[11]. These are more powerful than Cordova, but we do not see them as feasible for the POSEIDON framework, primarily because they require costly licenses. We also want to mention Unity[12], although that is primarily targeted at games development and also has limitation on the free version. Selecting a multi-platform framework is also a form of platform lock-in, as even though your resulting application may be able to run on multiple operating systems you lock yourself into a proprietary platform for its development, so you need to be very sure it is the right long-term strategy, both in terms of features and in terms of economy.

4.2.1 Stationary platforms

Stationary platforms are laptop and desktop computers and the input and output devices connected to these. Training activities for primary users should run on such platforms, as well as management and preparation activities for secondary/tertiary users. Major operating system families are Microsoft Windows, OS X and Linux. We want to support at least Windows and OS X, as these cover just about all consumer laptops and desktops.

We have used two platform strategies for prototype stationary applications, web and the multi-platform framework Unity. The main secondary user prototype application is a web application, featuring monitoring and input of all content except routes. This is a responsive web application, so it also covers the mobile platforms. The web framework used to develop the application is part of the POSEIDON framework, and described in chapter 6. Much of the POSEIDON infrastructure is server-side components with http APIs, and these are well-suited to use in web applications.

Unity was used for the training applications. Unity allows compiling the applications for all the major operating systems of stationary systems. It also allows compiling applications for mobile platforms, but our stationary framework for training applications include the interactive table. The full stationary platform for training applications is a PC attached to a large screen and a set of input devices that allow a natural interaction of the system, based on different forms of gesture input.

A conceptual sketch can be seen in Figure 10. The PC (1) is connected to the large screen (2) and handles all processing requirements. An interactive table (3) is attached to or integrated into a living room table and captures hand gestures performed above it. Therefore, the stationary system provides the following inputs, outputs and data connections:

[9] https://cordova.apache.org/
[10] http://xamarin.com/
[11] http://www.qt.io/
[12] http://unity3d.com/unity

- Capture of hand gestures for interaction
- Output of graphically rich user interfaces
- Connection to POSEIDON platform

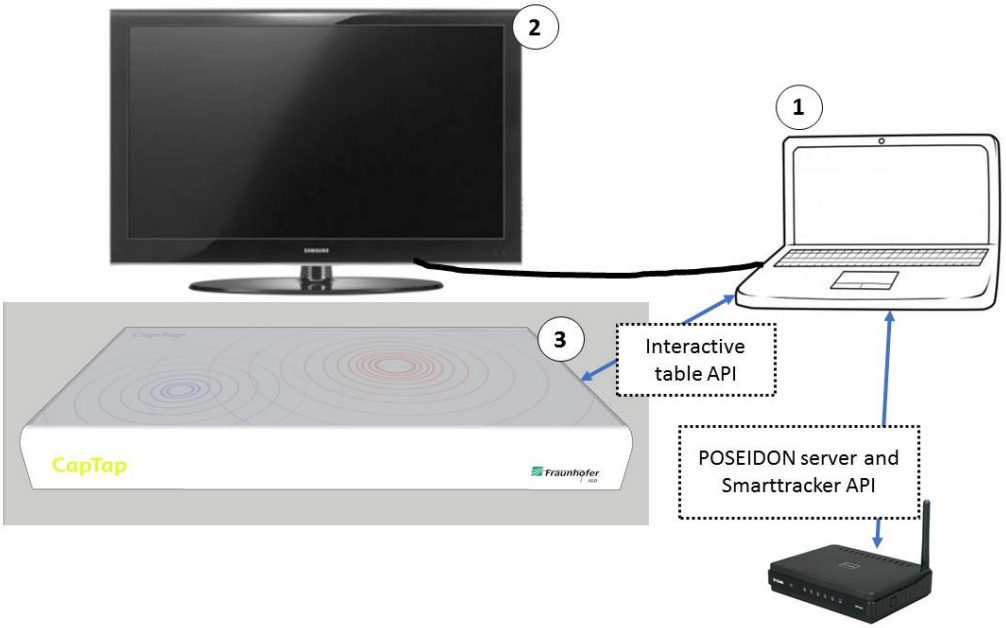

Figure 10: Stationary system including large screen device (2), PC (1) and interactive table (3)

As shown in Figure 11, the different input systems are interfaced by the POSEIDON infrastructure components using their respective APIs. The platform is not comprised only of the PC, but also includes information acquired from the server-side infrastructure and from the interactive table. Finally, this is translated to commands to the Output APIs that control the content of the large screen. This could be 3D APIs such as X3D or Unity, which is used in our prototype implementation, but also include 2D contents that might be better suited to transmit some content.

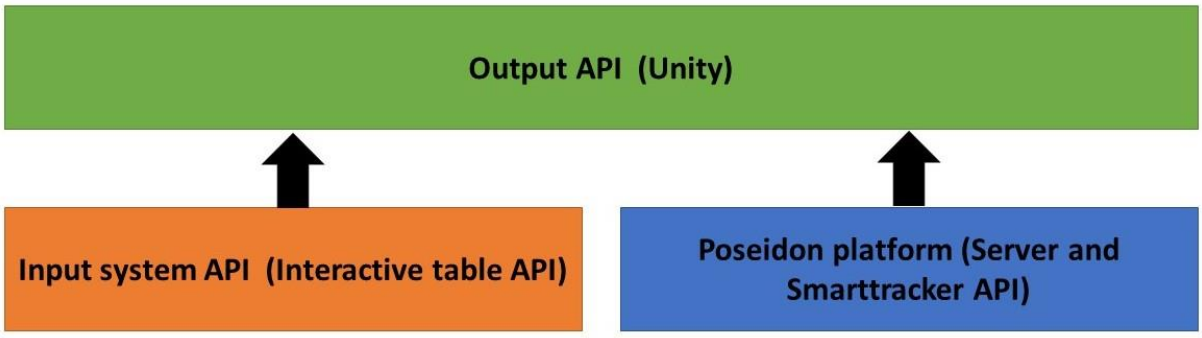

Figure 11: Communication flow of Unity-based stationary system

4.2.2 Mobile platforms

Primary user functionality such as notifications, guidance and instructions should be available on mobile platforms so that the users have access to the functionality at all times. In addition, the secondary user functionality of monitoring and managing calendars and instructions should be available on mobile phones because most people have a smartphone with them at all times.

Two different platform strategies were selected for mobile platforms. A web application is preferable where the wanted functionality can be implemented with web technology. The web strategy is a shared strategy for stationary and mobile platforms. The main secondary user prototype application is a responsive web application, usable on all platforms. It was clear that some of the things we wanted to do in POSEIDON, such as using the sensors of the devices to track context and provide navigation, even when the screen is not on, could not be done with a web application. The native application strategy is selected where a web application does not provide the desired functionality, i.e. where we want an application which can run in the background on the device at all times and make full use of all device resources. Web and native are the two extremes of the possible strategies, and cover the full range. We have also considered the hybrid approach which combines them, and the Cordoba framework. Based on previous experiences with Cordoba, there is a big gap between what you can do in it and what you can do natively, and so we wanted to stick with either web or native. As the technology matures, a hybrid approach may become more feasible.

As discussed in our framework requirements, Android is the primary platform for native applications, with an intention to also support iOS. Prototype native applications, as well as framework components, have been developed for Android. We have not developed native iOS versions of the same functionality in the project. This is due to considerations of effort versus gain. It would have taken considerable effort to develop and test the components on iOS. There would be no gain within the project, as all the prototype functionality can be tested with the Android applications. And it would not be wise use of resources to develop iOS framework components until we see a need for them. Our framework therefore gives specific support to the Android platform, with components for connecting to POSEIDON services. The infrastructure of services is available to be exploited by applications on all platforms, using server APIs that are part of our framework.

We also exploit the fact that Android uses the Java programming language. Framework components for connecting to the SmartTracker service are available in both regular Java and Android versions, so that they can be used to build stationary Java applications in addition to Android applications. We are also working in the last part of the project to generalise more of the Android codebase, to be able to provide more open-source Java code.

4.3 Resulting architecture

The key features of the POSEIDON architecture are the infrastructure of services and data specifications, and how client applications connect to these services. Figure 12 shows an overview of the architecture. Server-side services are a prominent feature of this architecture, and they are shown along the bottom of the figure. All data which may be shared between different applications and/or between primary and secondary/tertiary users, must be stored on the servers. Five APIs allow client applications to connect to these. We provide a file server for storing instructional content including multi-media, and this has an API for client applications to upload and retrieve files. For calendar events we use the Google Calendar service and its API. Tellu SmartPlatform, which implements the SmartTracker service, has two APIs. The sensor input API is for input of raw data to be tracked, such as positions. The REST API allows client applications access to the data model and stored data of the service, which includes user accounts with preferences and current and past tracked data. The learning module also does data collection, to be able to do analysis of user behaviour. Data is posted by the context middleware, and the query API allows applications to retrieve information. In addition to documenting the APIs, we specify the data to be stored in these services. By using the services and adhering to the specifications, any third-party application can connect to the POSEIDON system. The infrastructure is described in detail in chapter 5.

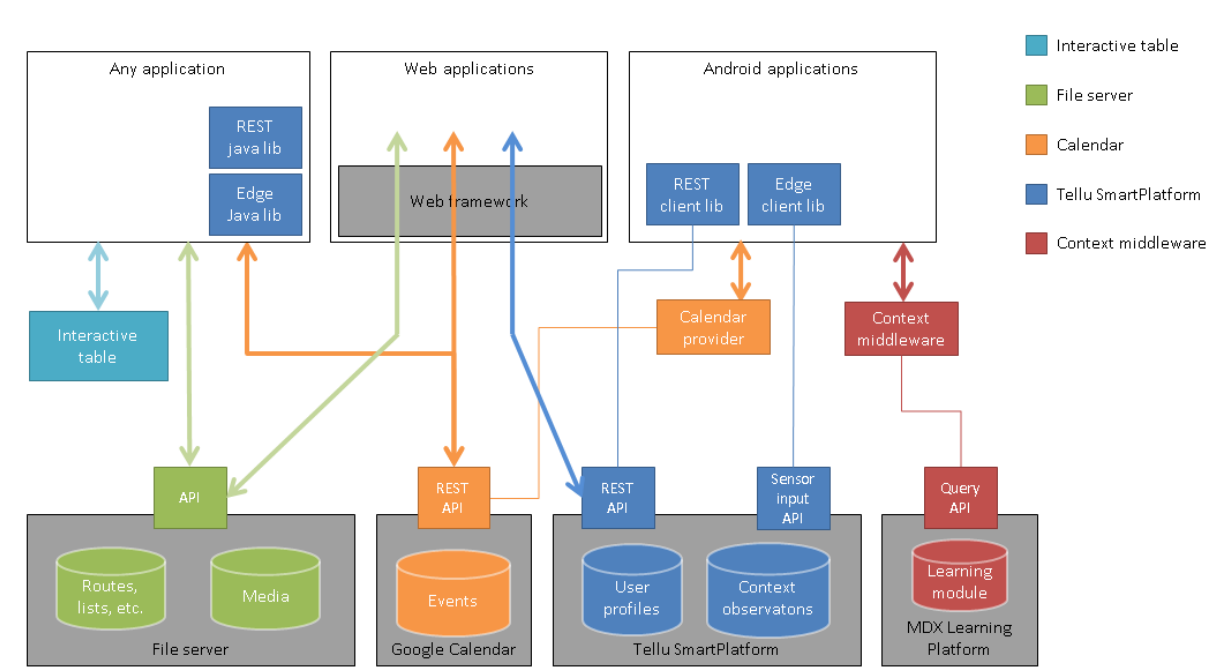

Figure 12: POSEIDON framework architecture

Client applications can connect to the service APIs in two ways. Any application can connect directly, using the specifications we provide. In addition, the framework provides some middleware and application components for select platforms, which handle the communication with the server-side API and can be used to ease the development work on these platforms. The Android platform has a framework of data providers to synchronise local data with servers, and Google provides a data provider for its Calendar service, so Android applications are recommended to use this interface rather than work directly with the server-side API. We provide Java and Android components for using the SmartPlatform APIs, taking care of the network transactions. In Figure 12, thin lines indicate connections handled by the framework, while the thick lines with arrow-heads indicate a connection between an application and the infrastructure provided by the framework. Note that not all possible connections are shown in the figure.

Two other parts of the infrastructure are shown in the figure. The interactive table is considered part of the framework for stationary applications for primary users. And middleware for providing context awareness is provided on the Android platform. The middleware is a component which can be installed on Android devices. It has its own interface which applications can use to communicate with it. Applications can subscribe to updates in specific contexts, and the middleware deliver these to the subscribing applications.

The white boxes in the top part of the figure indicate the types of applications which can be created with the framework and connected to the infrastructure. We indicate the specific support for web and Android applications. For web applications, we specify a web framework. Android applications are developed using the Android developer tools and technology stack, and we provide Android-specific components for the communication with the SmartPlatform server. Other than this, applications on any platform with an internet connection can connect to our server-side services. For stationary applications other than web we recommend either Unity or Java, where we provide some components.

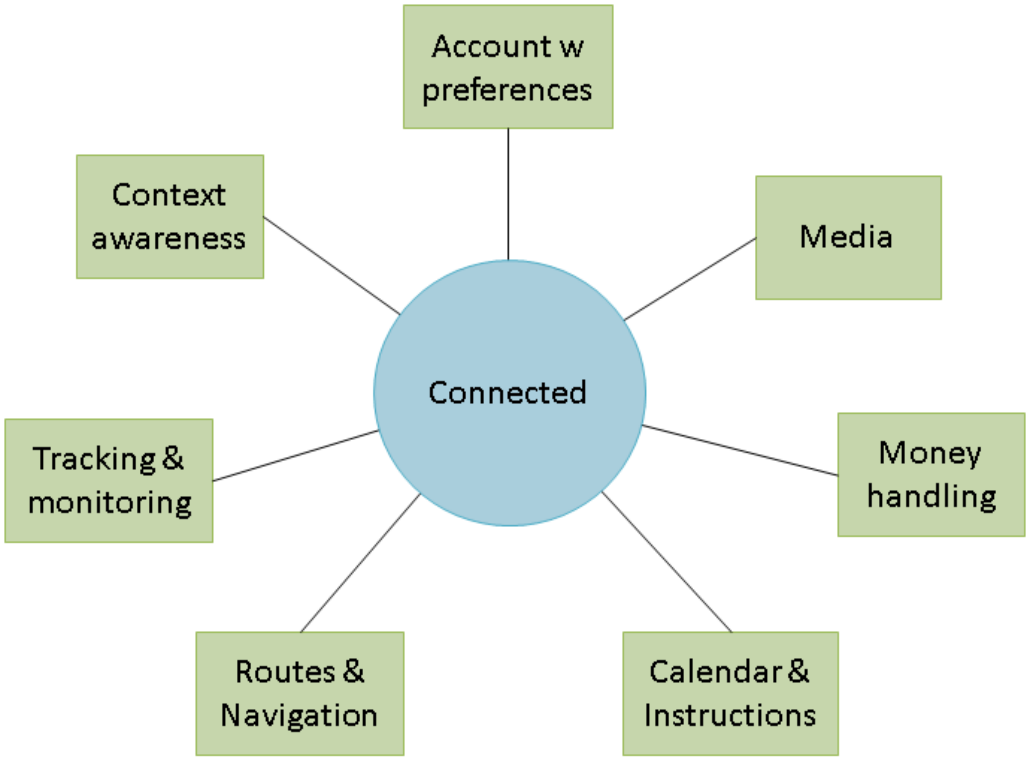

Figure 13: A service view of the infrastructure

Figure 13 indicates the service aspects provided by the POSEIDON infrastructure. These are connected together by the concept of a POSEIDON user, and when used together they provide a good foundation for a comprehensive ICT service for people with Down syndrome and their carers. Here is the mapping of these service aspects to the infrastructure components which provide them:

- Account w. preferences: Stored on the SmartPlatform server and available through its REST API. Java and Android component for REST API.
- Media: Stored on the file server, referenced in the data formats of the various types of instructional content.
- Money handling: Data specifications for product and shopping list, stored on file server.
- Calendar and instructions: Calendar events stored in the Google Calendar service, available through the calendar provider Android middleware or directly through the server-side API. Extended data specification for calendar events including instructions. Specification for video list stored on file server. Media for instructions stored on file server.
- Routes and navigation: Data specification for routes, with instructions for navigation, stored on file server.
- Tracking and monitoring: SmartPlatform service for collecting and storing tracked data. Sensor input API and Java and Android component for this.
- Context awareness: Android middleware, connected to its own back-end learning platform server.

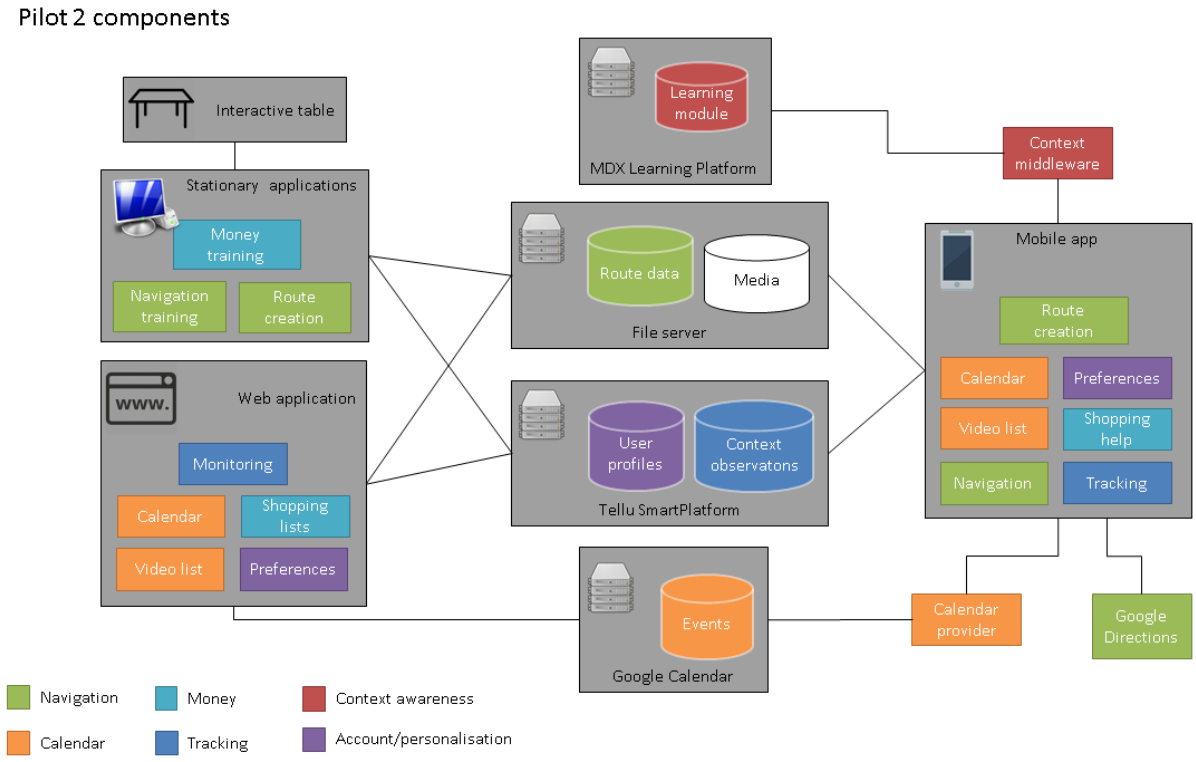

Figure 14: Overview of prototype 3 system for pilot 2

The technology infrastructure is described in the next chapter, followed by the developer tools and components in chapter 6. The POSEIDON prototype applications are built on this framework. As an example of how the architecture is used, Figure 14 shows the third iteration of the prototype system. The system is described in the third iteration of the D5.2 deliverable. Here we note that the server-side infrastructure components are placed in the middle part of the figure, while stationary, web and mobile (Android) applications are connected to this. This high-level figure shows the functionality provided on each platform in this prototype system.

4.4 Personalisation support

Individual personalisation of services is always important in the assisted living domain, to make sure each user gets the appropriate functionality and content. It is especially important for people with Down syndrome, as there are big differences between individual capabilities and needs. The framework needs to support personalisation. Although it is presented as one of many items in Figure 9, personalisation is an aspect needed in most components of the system. Here we give an overview of the main personalisation mechanisms currently provided by the infrastructure and used in the pilot applications. This support is two-fold: providing a shared store of personal preferences for applications, and providing shared storage and standards for content. The infrastructure is described in more detail in the next chapter. Keep in mind that the infrastructure is there to enable application development – it is up to the applications to use these mechanisms and others to provide a personalised service to the end users.

4.4.1 Preferences

The POSEIDON account, stored on the SmartPlatform service, includes a user profile which can store any number of preferences as key-value pairs. All POSEIDON applications should connect to the SmartPlatform service, logging in to the account and retrieving this user profile. Preferences for the user and the applications should be stored here. This common store of preferences allows them to

be reused between applications, as illustrated in Figure 15. The preferences are retrieved and updated through the SmartPlatform API. The technical details of this API is given in deliverable D5.4, chapter 2. See section 2.5 for the relevant information and example code.

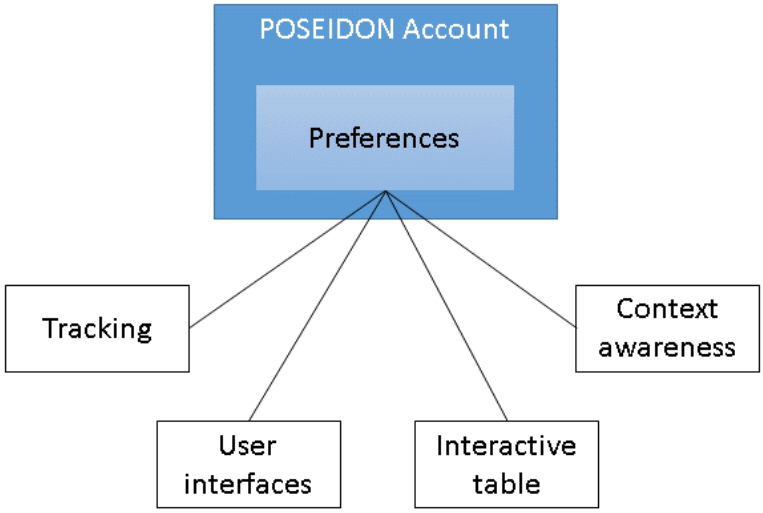

Figure 15: Preferences used by multiple applications

In the prototype system, most of the preferences can be edited in the web application. Examples of preferences used by prototype applications are visual theme for primary user interfaces, and whether or not the position of the primary user is tracked (the last one is not available in the web interface, as this web is mainly for secondary users while the primary users should have full control of this preference). Section 2.5 of D5.4 has a list of preferences and other properties currently used, which may be of interest to multiple applications. So far, the approach has been for applications to check if there are any existing preferences they can reuse, otherwise add new preferences as needed. With more maturity and applications, it will be possible to define a bigger set of standard, general preferences.

4.4.2 Content

Figure 16 shows an overview of the personal content supported by the infrastructure for the pilot applications. Storage is provided by two services, the file server and calendar service. Each of these has a well-defined API for applications to use to upload and retrieve content, and well-defined data formats. These are described in more detail in the next chapter, and the technical specifications for developers are given in deliverable D5.4. Here we give an overview of the data, with a focus on personalisation.

Three forms of instructional content is created by secondary users and stored on the file server:

- Routes: Routes for navigation training and real-time navigation. The route format is based on that used by Google, but with the addition of custom text, images and recorded sound for each instruction step. In the prototype 2 system, routes were generated by Google's service and then personalised by the secondary user. Now the route is fully personalised, with the secondary user defining the route geometry as well.
- Shopping lists: Products are registered with prices and images, and then used to create shopping lists for money training and shopping assistance. This is all defined by the secondary users.
- Video list: A personal list of videos, intended for instructional videos, can be defined.

Note that the filer server provides general purpose storage of files, and so can be used to store other content, but those described here are the data formats defined by the framework to insure the necessary functionality for the prototypes and interoperability between applications.

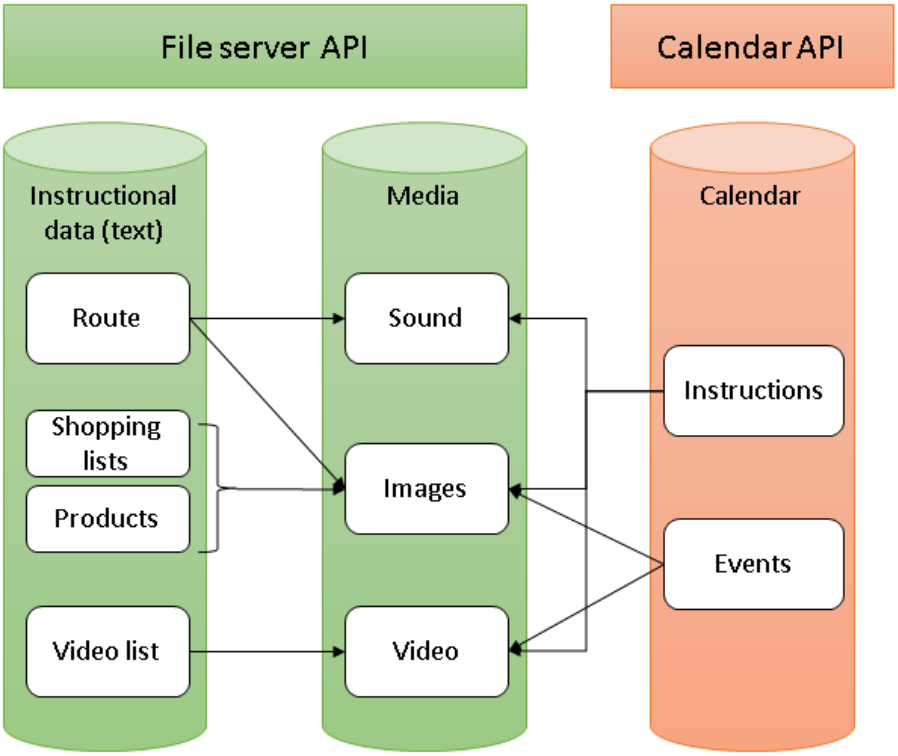

Figure 16: Personal content in the infrastructure

On the calendar side, content is organised as events in a time schedule. This is based on the normal calendar event model, with such fields as title, description and alert time, but support for images and video is added for further personalisation. In addition, a list of instructions can be added to any event, and each instruction can have text, image, sound and/or video. Personalisation of how the user is notified of events is a combination of account preferences and event content. Notification times can be defined for the individual event, to make sure the user is notified a specific amount of time ahead of the event. Preferences control how the calendar application notifies the user. Currently, two preferences are used. One preference specifies if the user should be notified at the start time of the event, which then happens for all events, in addition to any notifications before the start time specified by the event. The other preference is whether the notification (sound and vibration) should be continuous until acknowledged or less insistent.

The figure also shows which forms of media can be included in which content items. Media are files stored on the file server.

4.4.3 Other mechanisms

There will in some cases be personalization mechanisms outside the infrastructure. One example is the configuration of Android devices. The Android system, used on mobile devices for the prototypes, has a large number of settings which affect the applications running on the device. The user interacts with these settings through the Settings application of the device. Depending on the setting, it is either not possible or very bad practice for an application to change such settings directly. The user should be in control, through the Settings interface of the device.

One example of a relevant setting is font size. POSEIDON users may want to tweak this based on their eyesight. The font size in user interfaces made on Android's API can either be controlled by the global device setting, or not at all. It is not possible for a single application to change the font size only for itself, and the user should make the change through the user interface provided by the device. This is disruptive with respect to providing a complete and user-friendly service to people with special needs, but it is the price we pay to use generic devices.

5 Technology infrastructure

This chapter gives brief descriptions of each of the infrastructure components identified by the architecture. For server-side data storage and APIs, the developer documentation is found in deliverable D5.4.

5.1 Tellu SmartPlatform

Tellu SmartPlatform is a generic and highly configurable platform for data collection and processing. Its core is the storage and processing of data from sensor devices. Sensor data can in the broad sense be any type of data received as observations of limited size.

SmartPlatform is in active development, and will continue to be for the foreseeable future. However, it is currently used commercially in a variety of domains, where it must meet strict demands on uptime, security and error-free operation, so it has a high degree of maturity. It is usually offered as Software-as-a-Service, with the main commercial instance hosted by a professional hosting company providing 24-7 monitoring and support. Tellu offers this service to companies which act as service providers in their respective markets, and it is these partner companies which deal with end users. Tellu has also delivered server instances as disc images for virtualization. In relation to our framework requirements, it is important to note that SmartPlatform is closed source, with the exception of the open-source components it uses.

Note that in the DoW and early project work, only the term SmartTracker is used, as that was the name used for the platform and Tellu's service at that time. Tellu SmartPlatform was later adopted as the name of the platform, while the service is currently branded TelluCloud. In POSEIDON we use the term SmartPlatform for the platform, while the original SmartTracker name is still used for the service. The distinction between these terms is usually not important.

This section gives an overview of the platform as a POSEIDON infrastructure component, with relevant technical documentation for developers found in D5.4. A general presentation of the service is found on Tellu's web site: http://www.tellucloud.com/

5.1.1 Platform overview

Figure 17 gives an overview of the platform. Refer to the figure for the following presentation. On the bottom of the figure is the input to the platform – the connected sensor devices. The Device Adapters connects the heterogeneous sensor devices with the core. It consists of *edges* which implement the different protocols of the different devices, translating between the device-specific protocols and SmartPlatform's internal format. In addition to receiving data from the devices and passing it on to the core, an edge typically support commands to the devices, such as for configuration, so there is communication both ways. In the commercial applications of SmartPlatform, most connected devices are small purpose-built sensor devices such as GPS, communicating using mobile network connection and SMS. Edges have been developed for many such devices, as well as for some sensor gateways. For the cases where we have control of the communicating device, such as for SmartPlatform-specific mobile apps, there is also an edge more directly based on SmartPlatform's internal format. Otherwise, a device-specific edge needs to be developed to support a new type of device or protocol. As long as the protocol is known and the device has an internet connection, we can connect it to the SmartPlatform.

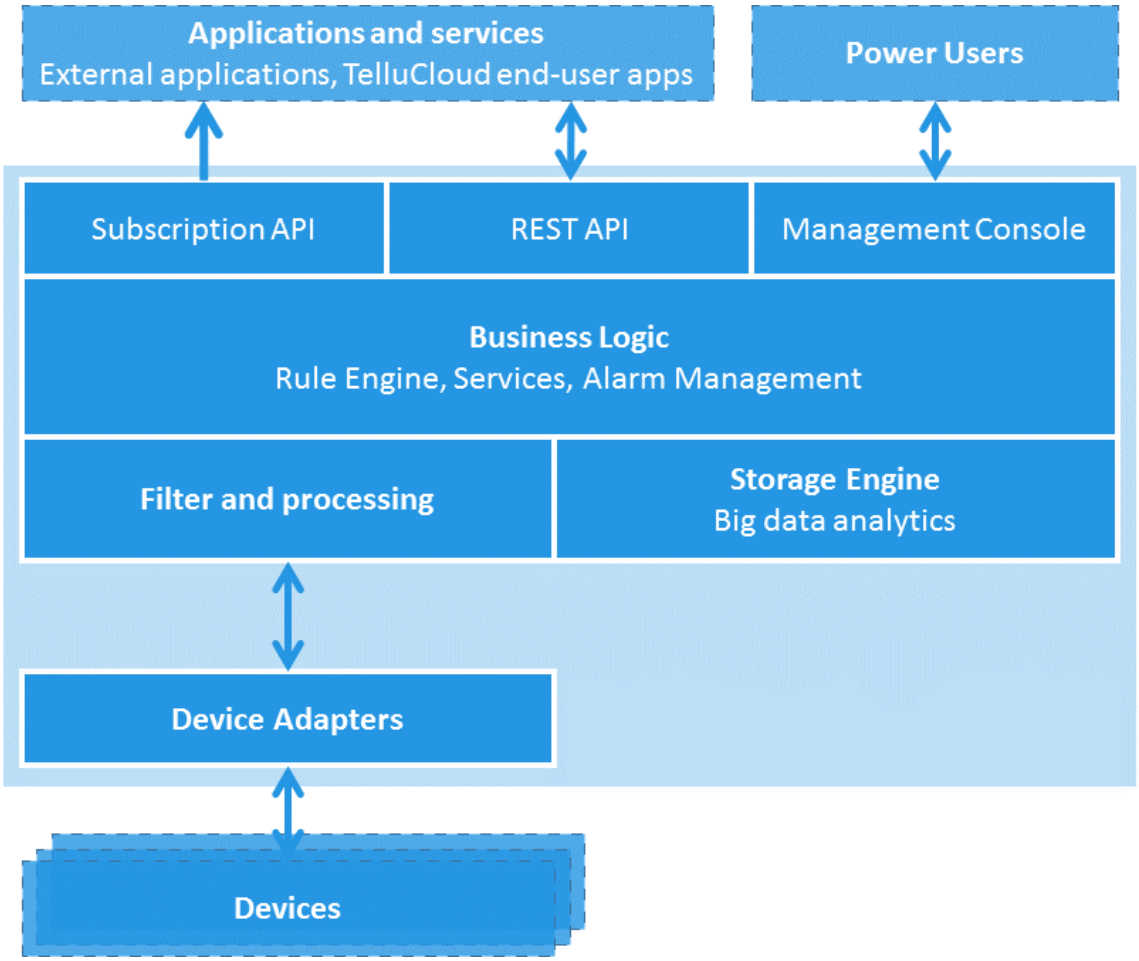

Figure 17: Tellu SmartPlatform architecture overview

Moving on to the main part (core) of the system, data from devices first go through Filter and processing, before being stored in databases (Storage Engine) and fed to a rule engine (Business Logic). As part of the processing, the device data is integrated in the platform's own data model. The Storage engine has a database for the current state of all tracked entities as well as a database for long-term historical storage. The data objects of the SmartPlatform data model are available through a rich REST API. Such an API gives access to the data to external systems, for reading it, and also for adding, updating and deleting where it makes sense to do so. A push-based API for subscribing to data updates has also been developed. Management Console is a web interface for administration and access to the data. This is highly configurable. The available functionality depends on the permissions of the logged in user, and it can be white labeled to suit a service provider.

5.1.2 Data, rules and security

The most important concept in the SmartPlatform data model is that of the *asset*, which is a tracked entity, i.e. the thing we care about. In our case it is the primary user, but it could also be a car or a house. A sensor device is usually associated with a specific asset, and all data from this device will then pertain to that asset. Core data entities such as assets and the incoming sensor observations are processed by the rule engine. This important component of the platform uses Drools[13] by JBoss, an open-source production rule system with an inference engine at its core. The rule engine is a place to implement server-side logic. The action of a triggered rule can be one of the external actions of the

[13] http://drools.jboss.org/

platform, such as sending an SMS or email to notify someone of an abnormal condition, or it can be to update the internal state of an asset. More advanced logic such as machine learning and artificial intelligence can be achieved by designing larger rule systems and state sets. The management console adds a layer on top of the rule code, to allow for user-friendly configuration of rule logic. It involves parameterized Drools template code, with placeholders for values to be filled in when instantiating a set of Drools rules, and a form-style web user interface for instantiating such rules.

A core concept for many of SmartPlatform's use cases it that of *alarm*. In traditional use, the rule engine is mainly used to detect abnormal conditions or other conditions that needs human attention. A result of rule triggering is then to raise an alarm with a specified level. Alarms are available through SmartPlatform's web application, or they are passed on to alarm centres through the platform's APIs. Messaging actions such as SMS and email are used as additional notification channels.

Data access is of course based on authentication and permissions. SmartPlatform has a role hierarchy for user entities and a detailed permission system to fine-tune what data such a user entity can access. We can have secondary users with access only to the data pertaining to the primary user they care for, as well as tertiary users with access to multiple primary users but a more limited data set for each one. Communication is secured with HTTPS. The data model, data storage and access policies are described in more detail in Deliverable D5.4 – Databases for integration of services.

5.1.3 SmartTracker service in POSEIDON framework

The SmartTracker service implemented with the SmartPlatform has been part of the infrastructure since the Description of Work of the POSEIDON project proposal. It is a central server-side component of the framework, holding user accounts and data. Here is a summary of what functionality it brings to the POSEIDON system:

- Accounts and authentication: SmartPlatform users, with username and password, are used as POSEIDON accounts. Client applications and other server applications must authenticate themselves with the SmartTracker server to access relevant data.
- Preferences for personalisation: With properties stored as part of the user account, SmartTracker offers a cloud storage of preferences for personalization and other settings, so that these can be shared between client applications.
- Receival and storage of tracked data: Device data such as position and battery level is sent to SmartTracker and stored there. Applications can also send logging events here, and in the pilots it is used to collect feedback from users. The stored data can be made available to secondary and tertiary users through the POSEIDON web. While it is only the POSIEDON mobile application prototype which sends such data to SmartTracker in the prototype system, this can be extended by connecting other applications. SmartTracker comes with device adapters for various sensor devices and can be extended with more, so that any type of input can be integrated.
- REST API for integration: The REST API gives access to the SmartPlatform data model and the history of tracked data. The POSEIDON web application is built on this API. It allows for other services and applications to integrate with POSEIDON.
- Management console: The management console is used by administrators and other personnel to configure and monitor the service.
- Rule engine: With the rule engine, SmarTracker offers a place to implement server-side logic. Rules can trigger on data from client applications, such as entering/leaving a location or low battery on the device. Actions include SMS and email, as well as contacting other services.

This is an area which is not prioritised in the research project, but it offers many possibilities for the service to grow and for integration with other services.

5.2 File server, multimedia and data formats

A file server is included in the framework infrastructure for applications to store and share any data not handled by the other framework infrastructure components. A generic file server application has been developed in the project to meet our data sharing needs. It can store any file, but there are two main types used in the prototypes. One is media files – images, sound and video uploaded by carers. The other is the definitions of instructional content, such as routes and shopping lists. These are structured data such as XML or JSON[14]. Instructional content typically includes media. The structured data will then contain references to the media files. This way a media file such as an image can be reused for several instances of instructional content without duplicating the file.

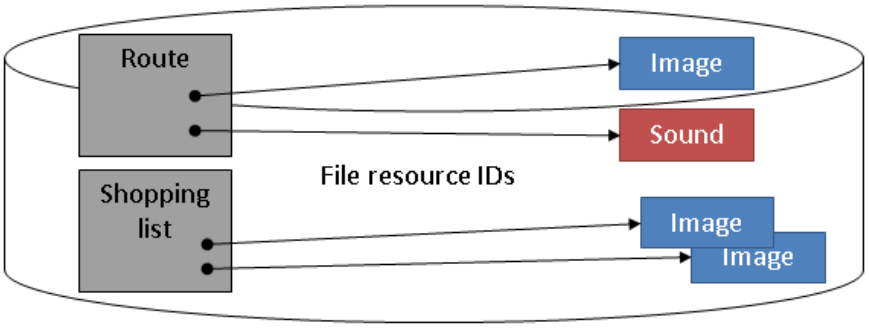

Figure 18: File server, with data files referencing media files.

5.2.1 File server

The file server has a basic HTTP API. A client application can upload a file, and it is given a resource ID unique in the server. This resource ID is used to get the file, delete it or replace it with a new version. The file server can also return a list of files, with resource IDs.

Access is authenticated with a SmartPlatform user, so the POSEIDON SmartPlatform account must be used to access the server. A login with username and password can be done either directly to the SmartPlatform server, or to the file server which then contacts the SmartPlatform server. An access token is returned, needed to authenticate all transactions. All files are associated with a specific SmartPlatform asset (primary user). For a SmartPlatform user with a single asset (this is the case with our secondary users), this asset is used. For users that have access to multiple assets in the SmartPlatform instance, the asset has to be specified when communicating with the file server.

Since client applications will typically need to list all files of a specific type or category (such as all routes or shopping lists), every file has a category which is specified when uploading the file. Conceptually we can think of this as a file storage with a folder for each SmartPlatform asset (primary user), and sub-folders for each category. Listing of files are done based on category.

An initial file server was developed for prototype iteration 2, and used in the first pilot. The final version was developed for prototype iteration 3, and is more generic, allowing for storage of any type of file. This version also added authentication and integration with SmartPlatform, using the same authentication token that is used in SmartPlatform. It is used by all client applications in the

[14] http://www.json.org/

prototype: the POSEIDON web, the mobile application, the home navigation system and the money-handling applications.

The detailed file server API documentation is found in D5.4, chapter 3.

5.2.2 Media files

For multi-media instructions, client applications can use any form of media which can be displayed/played on the device. To support this, media files can be uploaded to the file server. More specifically, the instruction formats specified by the framework support text, images, sound and video. While the text is included in the instruction files, the other are separate files which may be uploaded to the file server.

5.2.3 Data files

These are structured files in specific formats and following framework specifications, defining various forms of instructional content, entered by the secondary user for the benefit of the primary user. The following are defined as part of the framework:

- Routes: A route defines a way to get from a starting point to a destination, and includes instructions for the user along the way. Each step may refer to an image and/or a sound file as part of the instruction. The data format is based on that used by Google Directions, the most-used route planning service, so that navigation clients can easily be compatible with such routes (but we add our own fields). The route data specification is found in chapter 4 of D5.4.
- Products and shopping lists: For supporting the money handling aspect of the framework, a list of products with prices can be defined, and shopping lists with such products. Data specifications for these are found in chapter 5 of D5.4.
- Video list: A playlist of videos can be defined for the primary user, intended for instructional videos. The videos themselves can be stored either on the file server or on an external service such as YouTube. The specification for the list is found in chapter 6 of D5.4.

In addition we have specifications for calendar events and instructions tied to these. Calendar events are stored by a different infrastructure component, and described in the next section.

5.3 Calendar

In the POSEIDON system we use the free Google Calendar service for cloud storage of events. This means the events are stored by Google, and a Google account is used for authentication. Media used to present the event and associated instructions are stored on the POSEIDON file server. Figure 19 shows the architecture for this part of the system. We see the event in Google Calendar storage, with a POSEIDON-specific instruction list extension as part of its data. This data references media files found on the file server.

Two types of application is indicated in the figure, with two different ways of accessing the calendar data. The web application interacts directly with the cloud storage by using the Google Calendar HTTP API[15]. Any application can access the calendar data in this way, but it requires authentication connected to a Google account. Android devices have middleware for synchronising calendar data from cloud services to local storage. Our framework recommends using this middleware for Android applications, accessing calendar data through the Calendar Data Provider API[16]. This Data Provider

[15] https://developers.google.com/google-apps/calendar/
[16] http://developer.android.com/guide/topics/providers/calendar-provider.html

system of Android adds a layer between the client app and the calendar storage service, and the API is generic and not Google-specific. Apps retrieve and update events through this generic API, and communication with the server is handled by a Data Provider implementation for the particular server API. Any calendar storage provider can develop a Data Provider for their calendar, thereby adding support for it on Android devices. This means there is nothing Google-specific about the calendar implementation of our app, and it by default supports other calendar services as well.

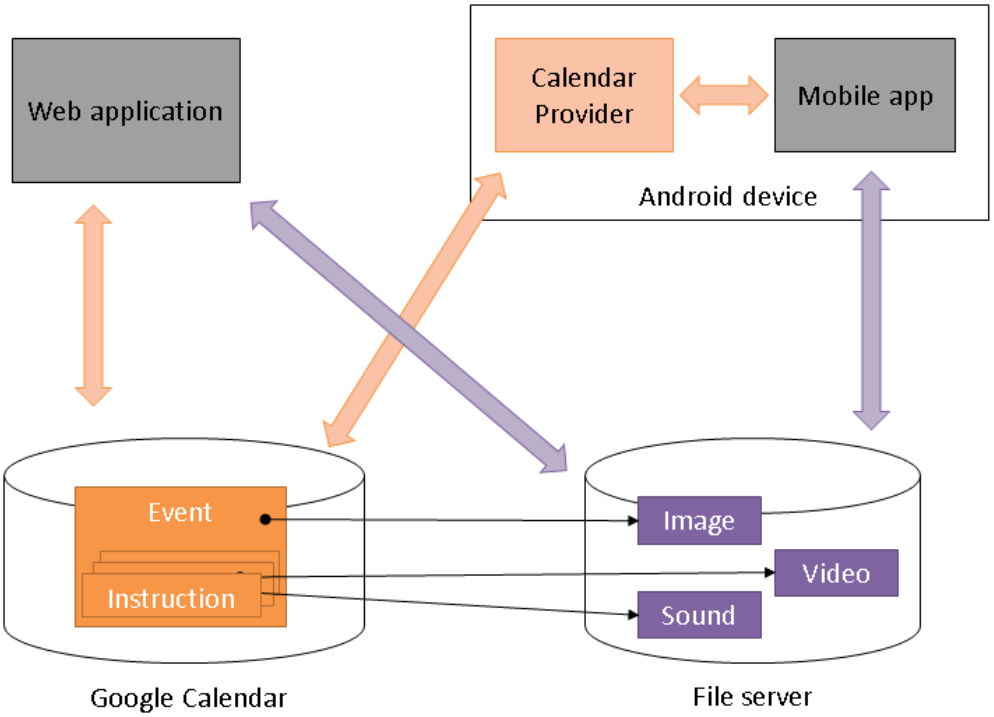

Figure 19: POSEIDON infrastructure for calendar events with instructional media.

Note that as we use the Google Calendar service for storage, the data will also be available in the Google Calendar web interface and other calendar applications. However, we add POSEIDON-specific data to the events, and accessing them through non-POSEIDON interfaces will not make sense. All calendar interaction should be through POSEIDON applications, where the user interfaces support the POSEIDON-specific data and media.

Both types of application need to access the POSEIDON file server, storing media files there when creating an event with media and retrieving the files referenced from existing events. The POSEIDON prototype system has two applications directly using calendar data. The POSEIDON web is the primary interface for managing the calendar, with the full range of options for creating and editing events. The mobile application is where the calendar events are delivered to the primary user, with reminders, notifications and instructions.

The calendar data specification is given in deliverable D5.4, where the file server API is also documented. Here we explain the conceptual model and how the data is used in the system. This conceptual model is based on the calendar APIs used – those of Google Calendar and the Android calendar data provider – but with restrictions and additions for POSEIDON use. This information should also be communicated to the (secondary) end users in some form, to help use the system correctly.

5.3.1 Calendars

All events belong to a *calendar*. Any Google account has a calendar, and it may have several. An Android device has access to the calendars of Google accounts entered in the device. There may be any number of calendars in an Android device. It is important, both for the applications and the end users, to make sure the same Google account and calendar is used in the various calendar-enabled applications in the POSEIDON system. While it is possible to support multiple calendars, this greatly increases the complexity for the carer setting up the system. Therefore, the prototype system should only be used with a single calendar for a primary user, and it is important that the Google accounts and devices are set up with only a single calendar.

5.3.2 Event scheduling

The main entity of the calendar model is the *event*. The main attribute of an event is its scheduled date and time. This can get quite complex, as events for most calendar services can be recurring based on some pattern, there can be exceptions to the recurrence, events can have a duration or not and they can be for full days (no start or end time). We do not support all the possibilities of the Google or Android calendar models in the POSEIDON framework. Here is a summary of our rules:

- An event can have a start and end time, or just a (start) time (no duration).
- We do NOT support all day events, or events spanning multiple dates. An event must have a start time, and if it has an end time this must lie on the same date. The reason for these restrictions is to keep the user interface of primary user applications simple, listing events sorted on time for a specific day.
- We support recurring events, as specified in the Google Calendar API. This means there can be a potentially open-ended amount of occurrences of an event. Note that any additional attributes of the event, such as description or route, will be the same for each occurrence.

An event may also have a reminder scheduled a specific number of minutes before the start time.

5.3.3 Other event attributes

These are the other attributes we currently support.

- Title/summary: This is the “name” of the event, and is the primary attribute for identifying the event. It is what is shown in lists, notifications and short descriptions, and should preferably be short enough to fit on a single line on a phone screen.
- Description: This text can be whatever the user wants to convey about the event. It can be left empty, if the title gives enough information, but will typically give more details about the event. There is no explicit limit in the system, but the length should be limited by the reading ability of the specific primary user and more generally by the screen size of mobile devices. A good rule of thumb is to not exceed half of a mobile phone screen.
- Image: An image can be used to give the event a visual presentation. It should be an icon or other image capable for giving a clear representation of the event in a small amount of screen space, as it may be used together with the title in lists and other places where a compact view of the event is given. For primary users who do not read well this is very important, and should be included for all events.
- Video: While the image is a visual supplement to the title, we also support inclusion of a video to supplement or replace the description text, especially for people with poor reading ability. For more media content, the instruction list is used.

- Route for navigation: In the POSEIDON system an event can be linked to the navigation functionality by specifying one of the routes planned for the primary user. A device supporting navigation should offer to start navigation at the start time of the event.
- Instruction list: While the description can be used to give instructions, the framework also supports an ordered list of multimedia instructions. Each item of the list can have text, image, sound and video. Instructions should be divided into simple steps, so that applications can show one step at the time and the user can advance the steps at his or her own pace. The text of an item should be kept short – typically a single, short sentence. Each sound or video clip should also be brief. The set of instructions is to be shown from the notification at the start time of the event.

5.3.4 Event notification

Events should be communicated to the primary user in two main ways. A calendar user interface should show events planned for a day, highlighting any ongoing or upcoming event, so that the user can actively seek out information about his or her schedule. A calendar application for the primary user should also provide notifications at the scheduled times, actively pushing information to the user. It is important that secondary user knows and understands how the events are communicated to the primary user, to specify events correctly.

The start time of an event is the main notification time. A calendar application should give a notification with sound at this time. A mobile application will typically use the notification system of the device, such as the notification bar of the Android system, with the notification opening the application if clicked. The application itself will firstly show the title, image and description of the event. If there are instructions, it should go on to show these in sequence each time the user acknowledges an instruction. Finally it should ask to start navigation if the event has a route.

If the event has a reminder, a notification is also given at the reminder time. It can use the same notification mechanisms as the main notification, but the details need to show how much time is left before the event starts. Note that a reminder has no information of its own, so except for the minutes left it cannot give the user any information different from that of the event itself.

An event is a simple object, and there is no link between events in this calendar model. It is very much up to the secondary users how to use events to create a helpful plan for the primary user. The event and its “start” time is primarily a mechanism for notification. For instance, if the primary user is to be at school at 8:30, we might need an event at 7:50 to remind the user to get ready to go, with instructions on what to bring. Then an event at 8:00 for actually going, possibly with route for navigation. The time to actually be at school – 8:30 – is probably less interesting for this calendar use, but it may be good to enter it so that it is displayed in the calendar.

5.4 Context awareness middleware

As part of the framework, we have the POSEIDON Context Reasoner Middleware, which includes software for the acquisition of and reasoning over context. This has been developed for the Android platform which can be included by different POSEIDON compatible applications.

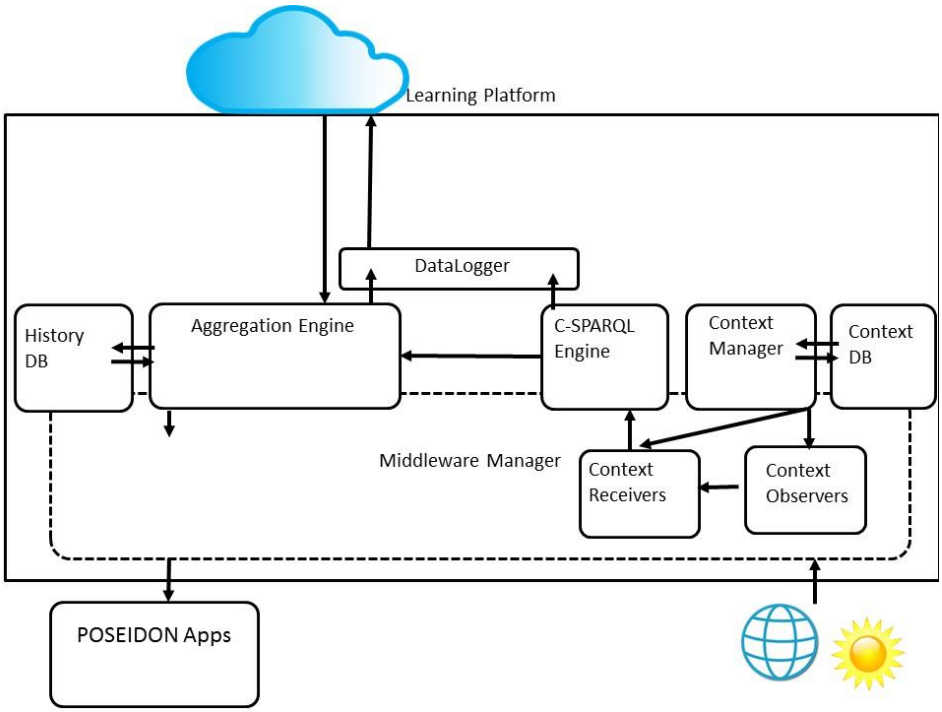

Figure 20: Context Awareness Middleware

In Figure 20, we present the architecture of the context awareness middleware. Regarding applications developers are expected to develop, they are represented here as POSEIDON Apps. The main components of the architecture include:

- **Context Observers:** These are the actual context acquisition components that collect the raw contextual data about the device, user, and environment. Developers can extend the middleware with new context observers at runtime.
- **Context Receivers:** These are the components that receive the raw context data, and send structured RDF streams to the C-SPARQL Engine. Developers can extend the middleware with new context receivers at runtime.
- **Context Manager:** This component handles the runtime management of context observers and receivers based on the context requirements of the POSEIDON compatible applications. At runtime it can import, dynamically load, and remove context observers and receivers.
- **C-SPARQL Engine:** This component handles the basic context reasoning by running the rules developers create for atomic contexts.
- **Aggregation Engine:** This component handles the higher level context reasoning by running the rules developers create for aggregation contexts.
- **DataLogger:** This component collects and logs different middleware events, including context state changes. This is then sends the data to the learning platform.
- **Learning Platform:** This component collects all events logged from the device, which can be used for learning some basic patterns e.g. The user makes the same number of deviations on a particular route.
- **Middleware Manager:** This component handles all communication between the POSEIDON Apps and the context awareness middleware. It is this component, you as a developer will be connecting to.

To include the use of the context-awareness middleware in your application, your application can use the middleware in one of the following scenarios:

- **Centralised Application:** In this scenario, the middleware is a completely separate application which can be installed and used independently. This app is distributed using the Google Play Store[17]. To use the middleware in this scenario, each application needs to use the Service declared as `org.poseidon_project.context.IContextReasoner`. To use this service, each application needs to bind to the service, before interface methods can be invoked.
- **Integrated into an Existing Application:** In this scenario, the middleware is integrated as a library into an existing application during compilation. In most circumstances, this usecase is not expected to be used. To use the middleware within this scenario, the containing application needs to use the core service class directly, which is named `org.poseidon_project.context.ContextReasonerCore`. Each request can be made directly, as a conventional java class.

5.4.1 Including the Middleware

In the scenario where the middleware is to be used as centralised application, the developer needs to include the `IContextReasoner.aidl`. To do this, the developer should firstly download the latest available version to ensure version compatibility from the opensource source code repository[18]. Once the developer has this file, they need to insert it in the "aidl" directory of their Android application under the following package name "`org.poseidon_project.context`", as should in below Figure.

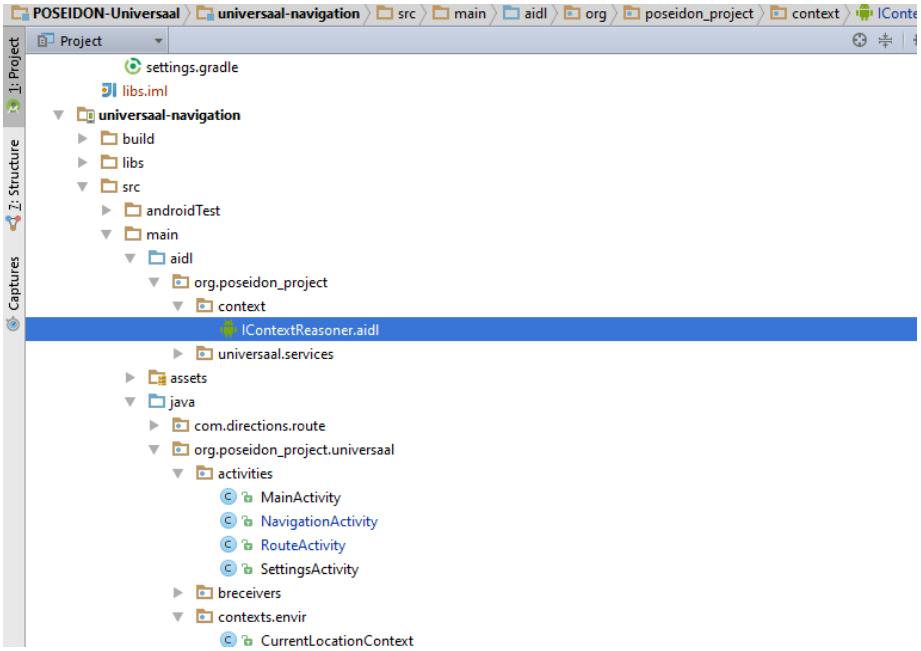

Figure 21: Context Middleware Interface

In the other scenario of incorporating the middleware in your application, then firstly the developer should create a new project folder for containing libraries if one does not exist e.g. libs. Next, the developer needs to get the source code for the middleware, and then **Check out** from the repository into that library folder as shown in the below figure.

[17] https://play.google.com/store/apps/details?id=org.poseidon_project.context
[18] https://github.com/deankramer/POSEIDON-Context/blob/master/reasoner/src/main/aidl/org/poseidon_project/context/IContextReasoner.aidl

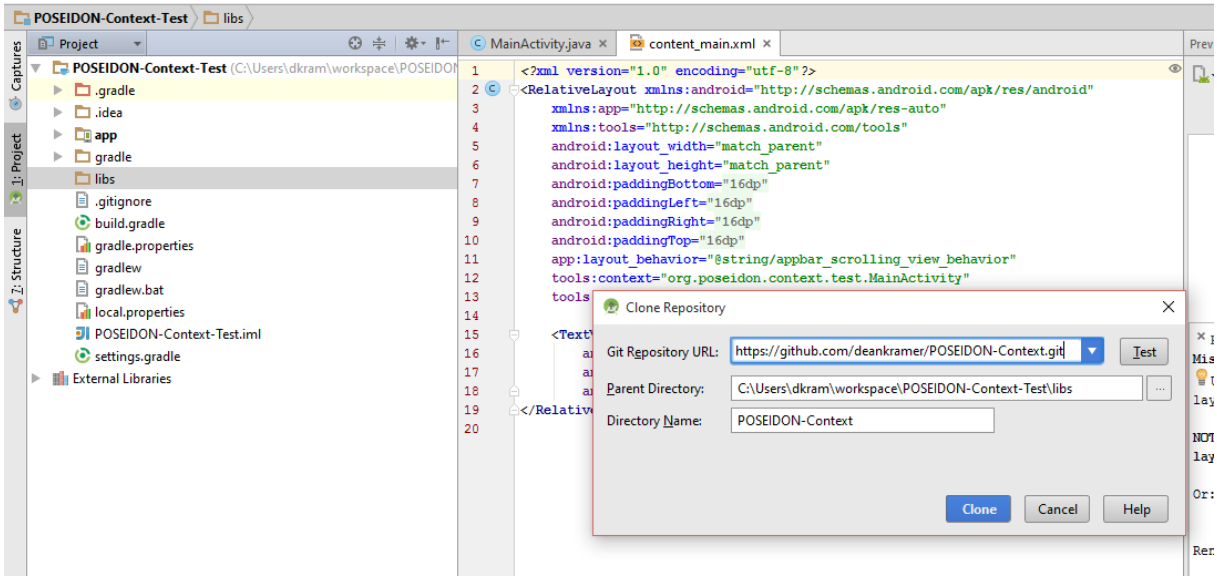

Figure 22: Checking out the middleware source code to a library

Once the source code is checked out into the desired directory, this **Module** needs to be added to the project for compilation and linking. This is carried out by right clicking on your **app Module** and clicking on **Module Settings.** Next, the developer needs to click on the Add symbol "+" in the top left corner of the window. The developer needs to select the source directory, which is the where we last checked out the reasoner source code to.

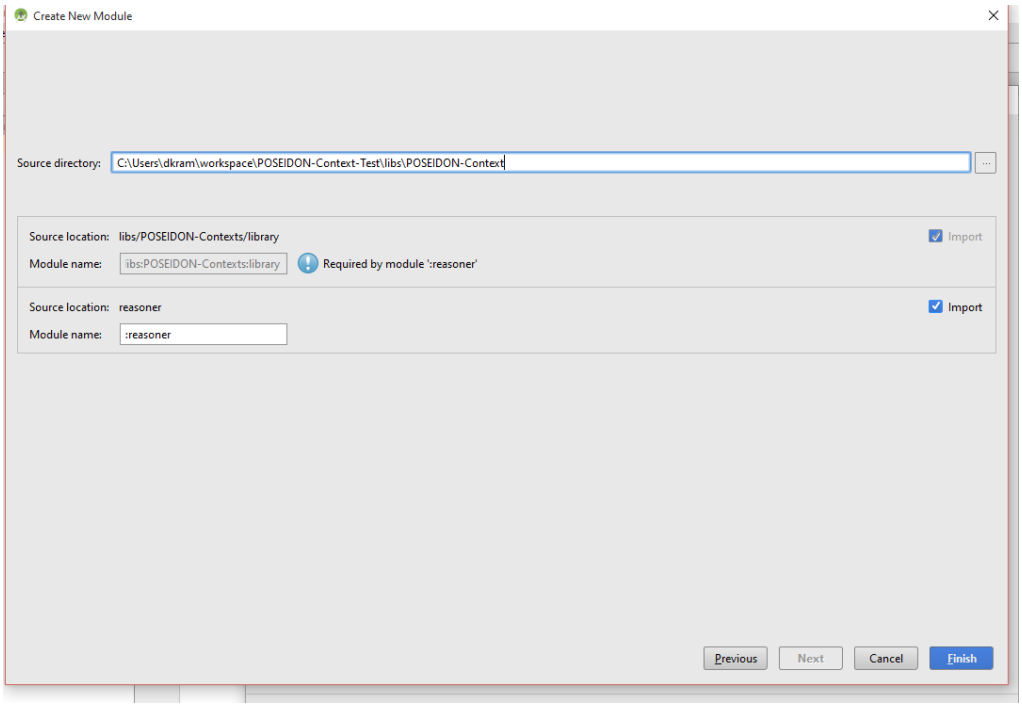

Figure 23: Adding Context Reasoner Module Dependency

5.4.2 Available Context Observers

Using creating new context rules, we have different Context Observers already available for use. These include:

- **BatteryContext:** Monitors the remaining amount of device battery in percent

- **WifiContext:** Monitors the status of the WiFi receiver on the device, whether it is not active, disconnected, connecting, or connected to an access point
- **TelephonyContext:** Monitors multiple parameters of the device's telephony receiver. This currently includes connection status, and whether the device is currently roaming
- **CompassContext:** Monitors the current orientation of the device in degrees.
- **LightContext:** Monitors the current amount of ambient light, measured in lumens.
- **ExternalStorageSpaceContext:** Monitors the amount of storage space remaining on the device shared storage area including the SD card, measured in megabytes (MB)
- **GPSIndoorOutdoorContext:** Monitors and deduces if the device is indoors or outdoors using the GPS receiver.
- **WeatherContext:** Monitors and sends weather data for defined locations.
- **StepCounter:** Monitors the current number of steps the user walks within a given interval.
- **DistanceTravelledContext:** Monitors the total distance the device moves within a given interval.

5.4.3 Using the Middleware

Once the middleware is integrated into your application development, you can then begin to use the different functionality through the given APIs. Detailed information on these have been described in Deliverable "D5.6 Integrated Technology".

5.5 Learning Platform

The learning and reasoning module is separate, yet complimentary to the context-awareness middleware. It is designed to provide statistics of different desirable activities and events over data that is collected from other applications, and internal context changes by the context-awareness middleware.

5.5.1 Using the Learning Platform

The learning platform has been designed as a web service, as part of POSEIDON. It is not designed as user facing services, but by other services, which might be user facing.

The platform has been designed as an XML-RPC, which serves as an approach for remote procedure calls. The main URL where all procedure calls are currently made is:
https://surveyposeidon.mdx.ac.uk/rpc/rpc.php

To use the learning platform, each developer needs to first attain an API Key. This API key is how our service identifies which applications/services are making the procedure calls. These are private, and should be unique for each application/service. To call the learning service, the developer needs to call the function `poseidon.runQuery`. Using this function, the following queries can be used:

1. Number of Deviations and Standstills: During a navigation journey, both deviations and being standstill for a long amount of time can be seen as undesirable. This graph counts the number of incidences.
2. Journey Time: This graph shows how long the user takes each time they take a particular route.

Further details on the parameters expected for this function can be found in D3.2.

5.6 Interactive table

The interactive table is part of the stationary system. The hardware setup and the firmware development of the interactive table is described in D4.3 – Interactive table. The interactive table is a prototype of a new interactive device which combines the size of a multitouch table with 3D hand position recognition. It is intended to be unobtrusively built in tables.

For the POSEIDON prototype it is usable in combination with keyboard and mouse of the PC as interaction device. The Moneyhandling Training and the Navigation Training, two services of the stationary POSEIDON system, are developed in such a way, that they can be used with the interactive table.

For an Example Project connecting the Interactive Table to a Unity program, see Section 6.6 Interactive Table API.

6 Developer tools and components

This chapter presents software and code libraries provided to developers to enable them to develop POSEIDON applications. The project provides tools for the R4C-AS methodology and context middleware developed in the project. It specifies tools and framework for developing mobile and web applications. And it provides code and libraries to help developers connect their applications to the POSEIDON infrastructure.

6.1 Tool Support for R4C-AS

We have created some tools to provide support creating the diagrams explained in Section 3.4.2. We base our tool on Modelio[19], an open source modelling environment that enables its extension through Module Development in Java. It is based on Eclipse ERP[20], but it works as a standalone application. It is compatible the use of UML/SysML diagrams. We have created a module that extends the features of Modelio with:

- SysML Requirements Diagram
- Context Dependency Diagrams
- Automatic Document Generation
- OWL Ontologies
- Traceability Matrixes

6.1.1 Writing requirements in tables and diagrams

In the extension we have programmed, there are two ways of creating and documenting requirements. This can be done in form of a table or in form of a diagram. We have extended the features of Modelio to enable the writing of requirements in form both ways. The developers can describe the requirements and their descriptions by simply using a table which can be exported/imported to/from excel documents.

[19] https://www.modelio.org/

[20] Eclipse is an integrated development environment (IDE) based on Java. It contains a base workspace and an extensible plug-in system for customizing the environment.

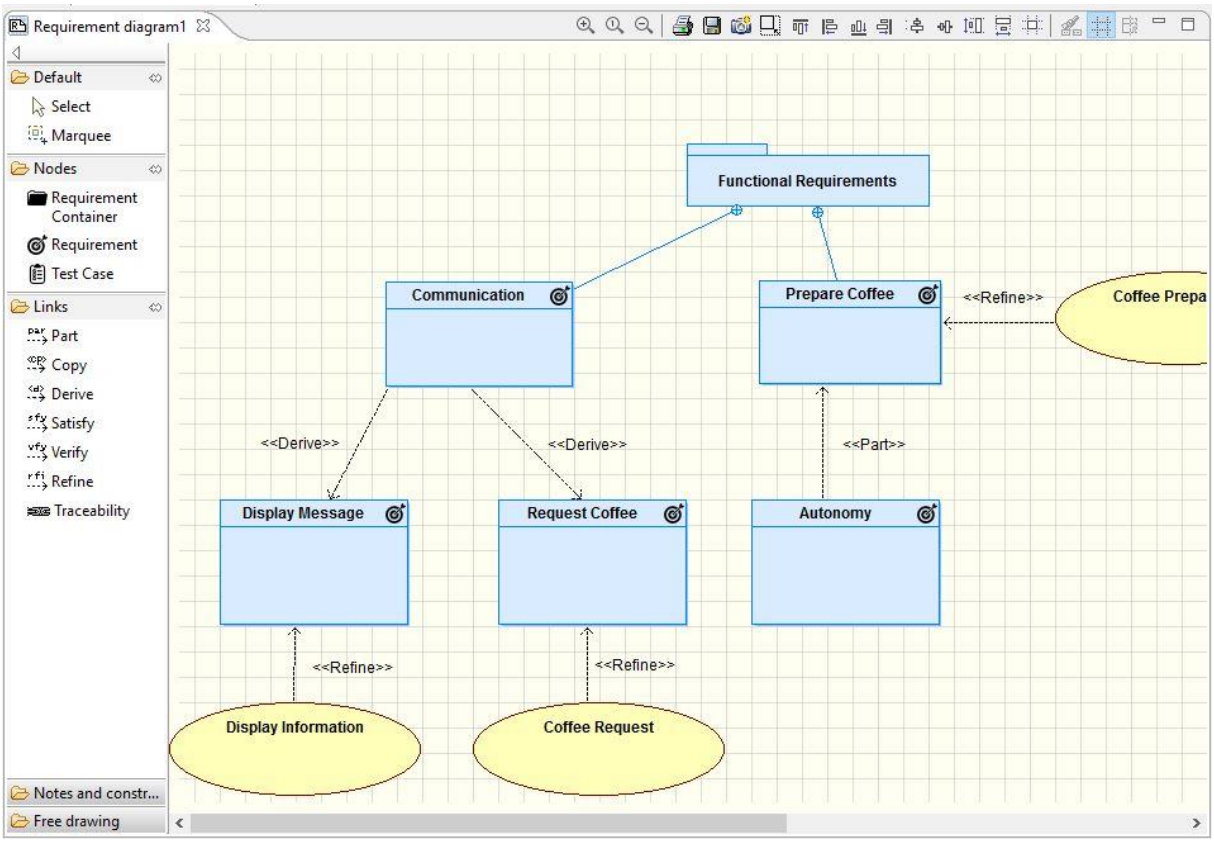

Figure 24: Visual representation of requirements and functionalities in the tool.

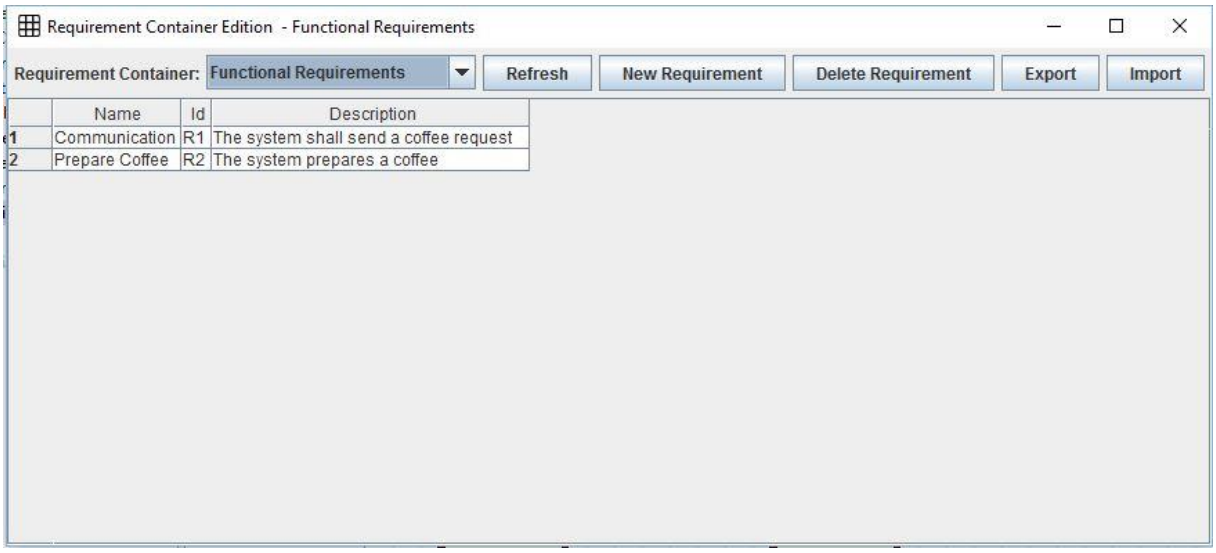

Figure 25: Tabular way of creating and documenting requirements with the tool.

6.1.2 Traceability Matrixes

The requirement diagram is the primary medium in SysML for conveying traceability among requirements as well as traceability from requirements to structures and behaviours in the system model. As you add new elements to the model, you will create relationships from those elements back to the requirements that drove the need for their creation. Establishing requirements traceability in this manner is an ongoing activity throughout design and development. We have also implemented a feature of requirements diagrams which is called traceability matrixes. Traceability matrixes can trace different requirements (Figure 26) and different contexts (Figure 27).

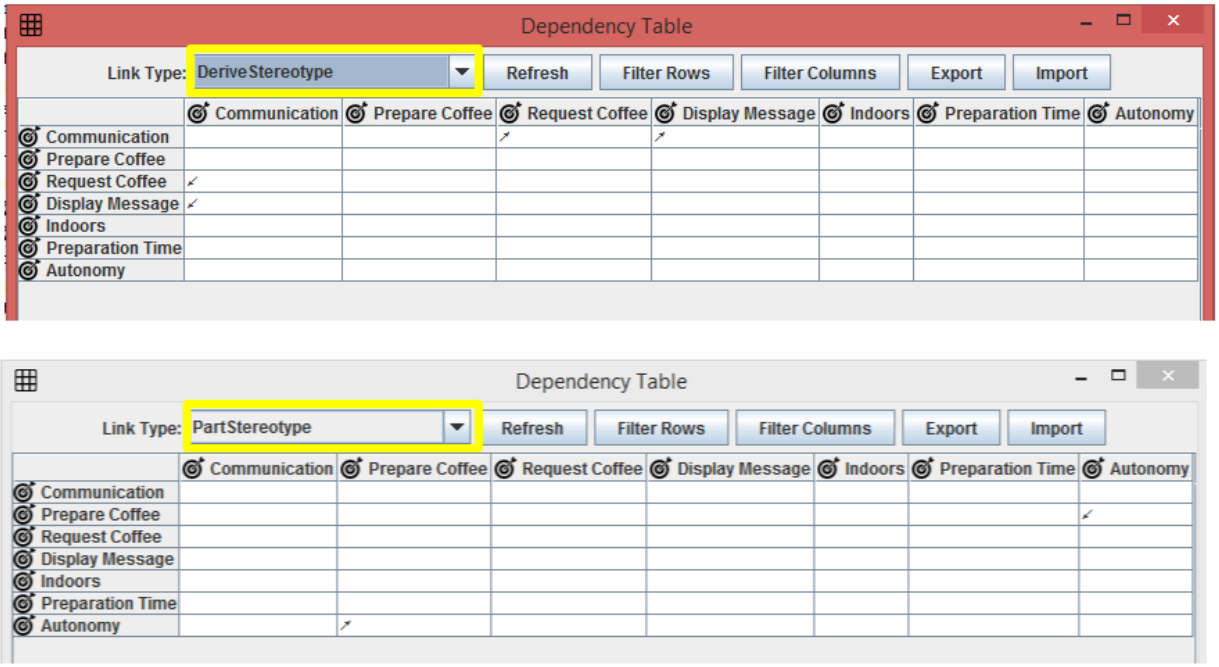

Figure 26: Traceability matrixes for SysML Requirements.

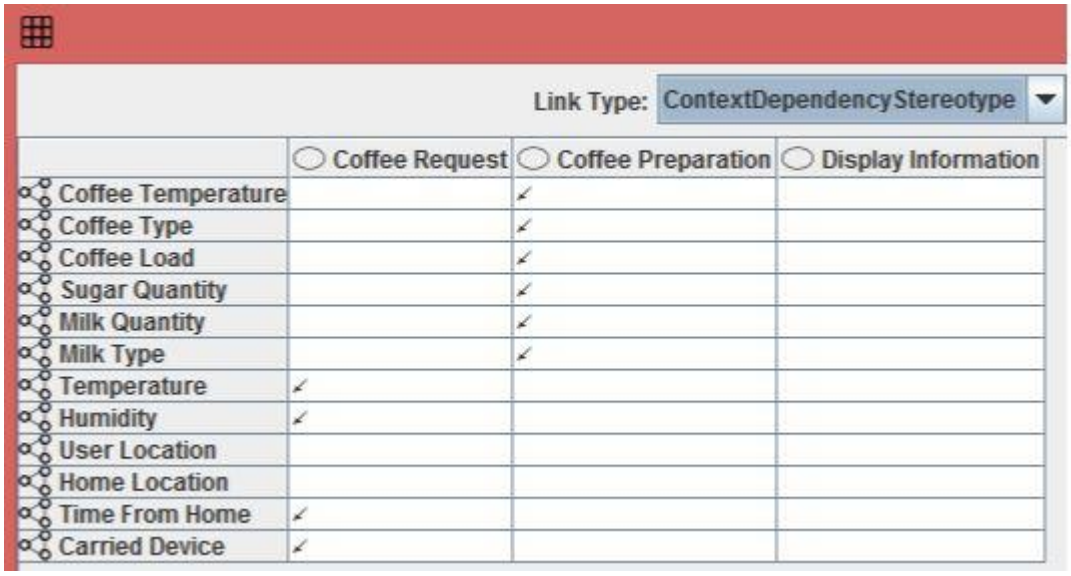

Figure 27: Traceability of context with functionality of the system.

6.1.3 Automatic Document Generation

The tool has the ability to generate some basic documentation based on the traceability matrices. This can be observed in Figure 28. Also requirements can be exported/imported in an excel document (Figure 29).

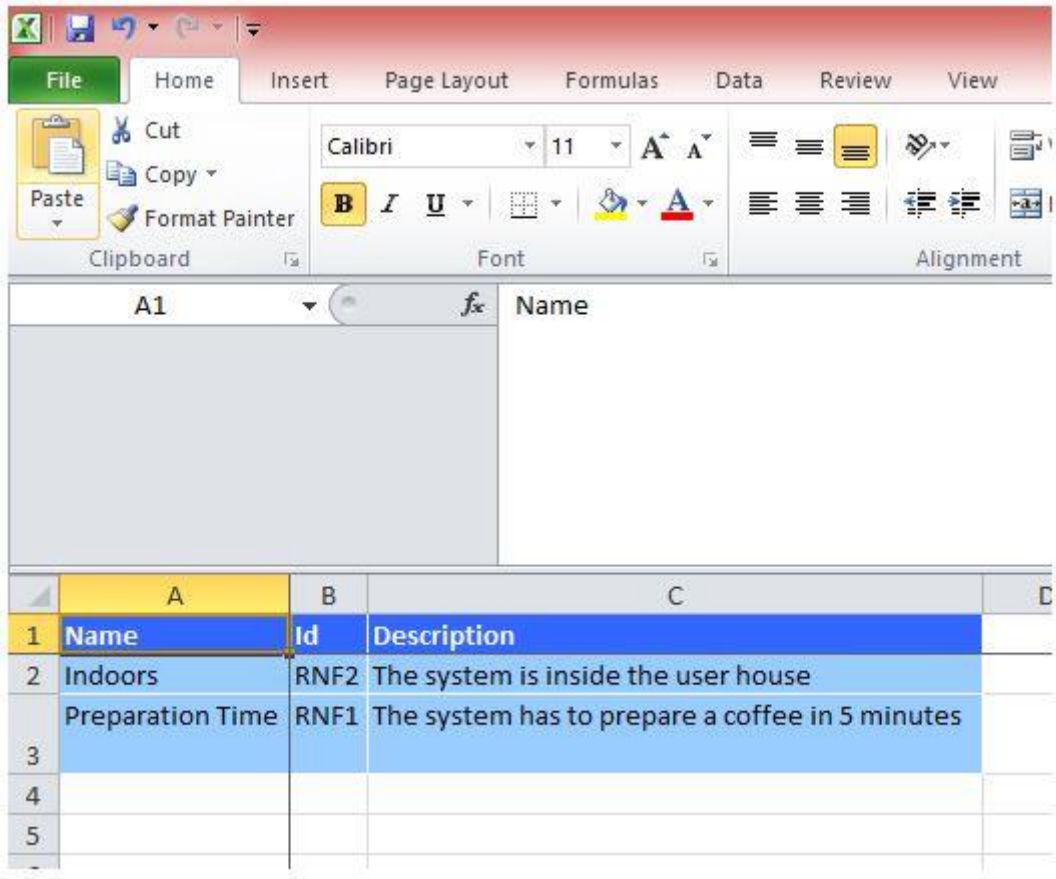

Figure 28: Example of a requirements table document automatically generated.

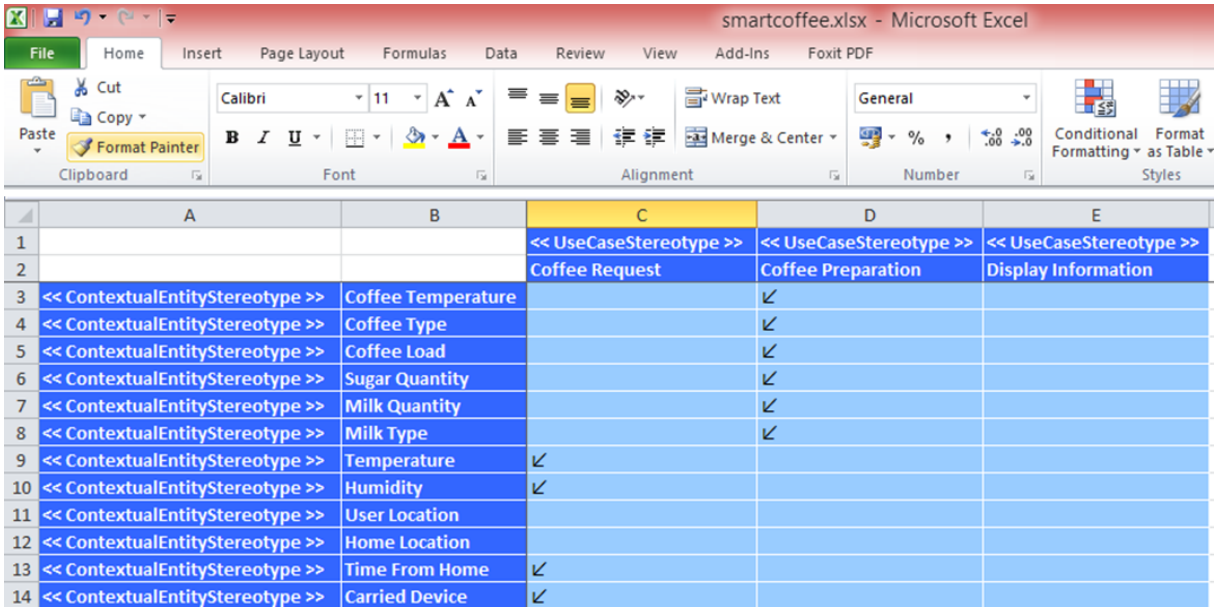

Figure 29: Example of a document automatically generated by the application.

6.1.4 OWL Ontologies

The tool enables the generation of an OWL document that can be used to enable the creation and manipulation of the context generated using our tool with other enhanced tools for ontology generation such as Protégé.

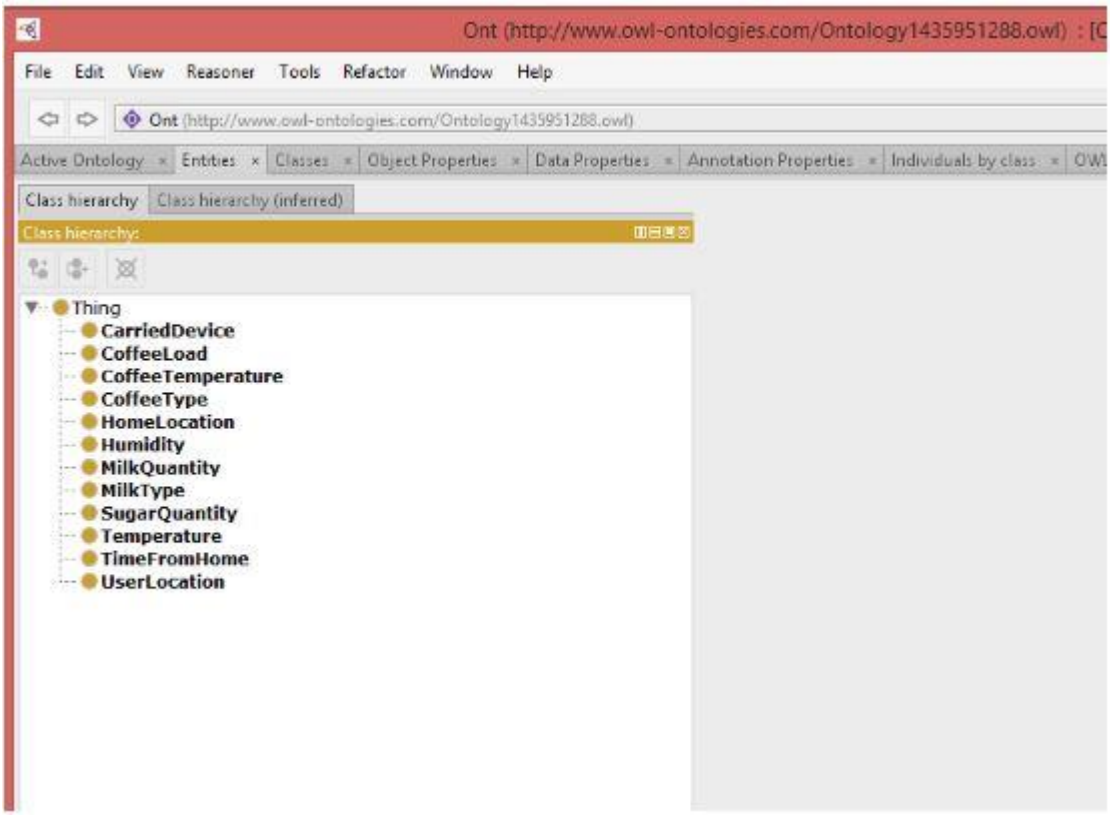

Figure 30: Example of an ontology automatically generated by our extension, modelled in protégé.

6.2 Context modelling

To facilitate the production of context rules for the Android Context Reasoner, we have created a tool to model each of the contexts. Using this tool, the developer can be better see all contexts in totality, add new contexts, test for in the model checking tool UPPAAL (Behrmann *et al.*, 2006), and export to C-SPARQL (Barbieri *et al.*, 2010).

This context modelling tool has been developed as a plugin to the modelling program *Modelio*[21]. Modelio is an opensource modelling tool for different notions including Unified Modelling Languge, Business Process Modelling Notation.

Context Models created in our tool are made up of three different model construct types:

1. **Context Source:** Used to specify the different raw context data, which encoded in RDF. These can be linked with many context rules.
2. **Context Rule:** Includes different logical expressions, and SPARQL methods, which are used with the Context Source to get a Context State.
3. **Context State:** States a particular context situation e.g. Low Battery.

[21] Available at https://www.modelio.org/

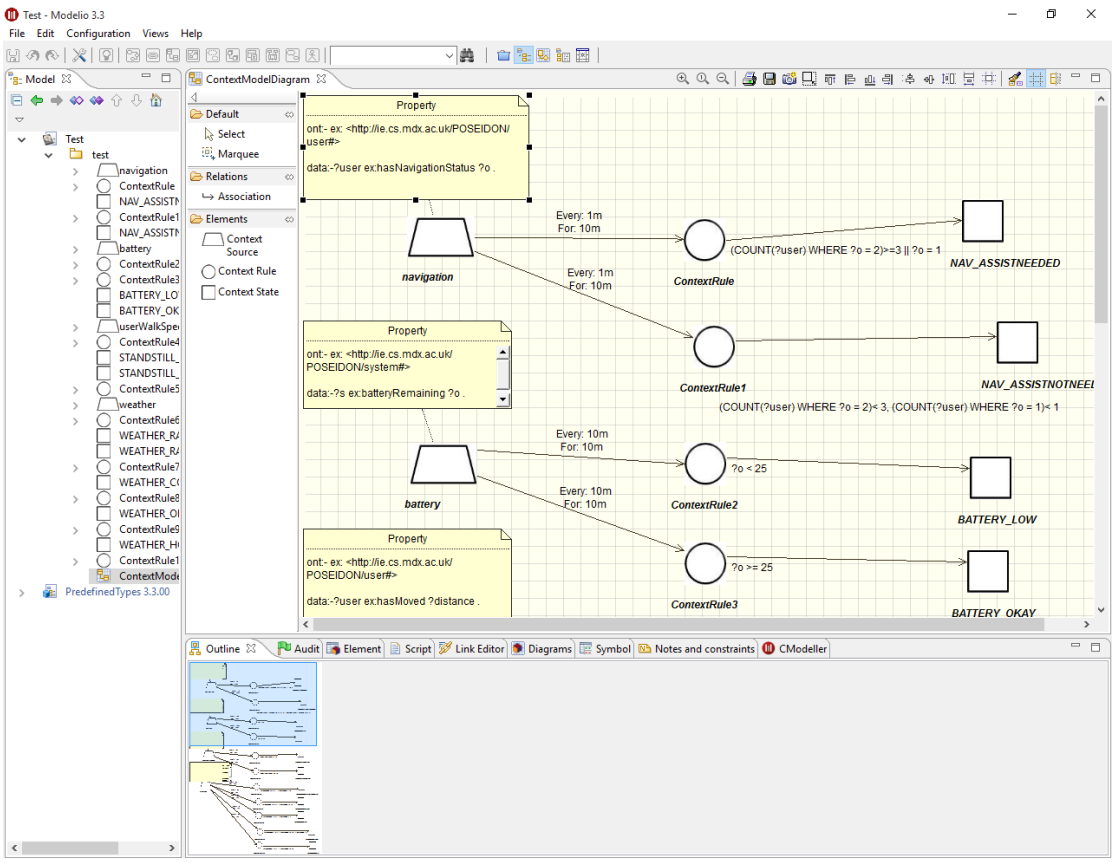

Figure 31: ContextModeller

6.2.1 Installing the Plugin

To use the context modelling tool, it firstly needs to be installed into your Modelio environment. To do this, firstly the developer needs to add the plugin to the **Modules Catalog.** This is accomplished by firstly clicking on **Configuration** in the menu bar, and **Modules Catalog.**

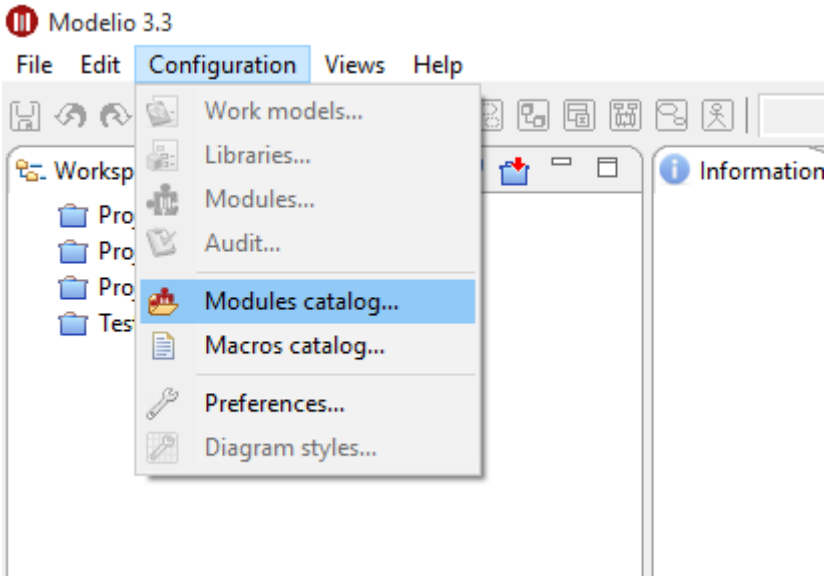

Figure 32: Editing Module Catalog

Then, the developer must click on the button entitled "**Add a module to the catalog…**", which opens an open file dialog. You must then select the ContextModeller plugin (with the file extension **.jmdac**). If the module is then added correctly, you should see the module in the Modules list, also being shown as *Compatible*.

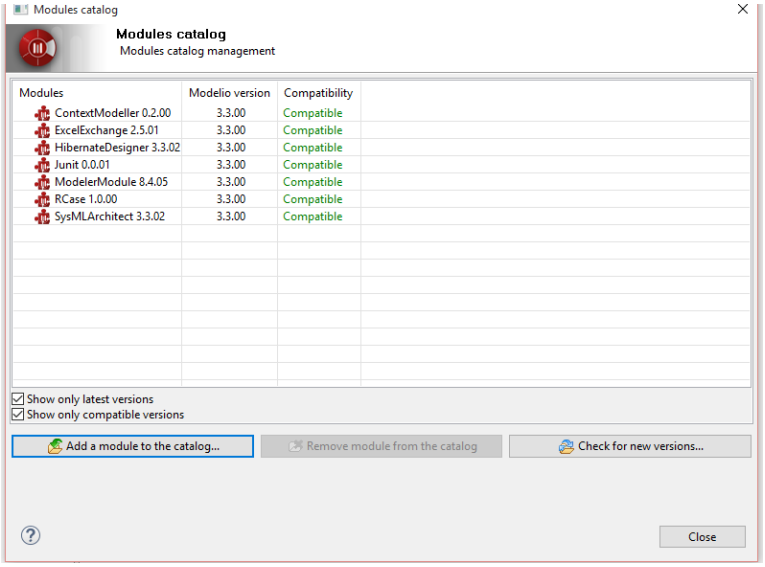

Figure 33: Modelio Module Catalog

6.2.2 Starting a new project

To start a new project, the developer should go to **File** and **Create a project**. The project needs a name, but description can be left empty.

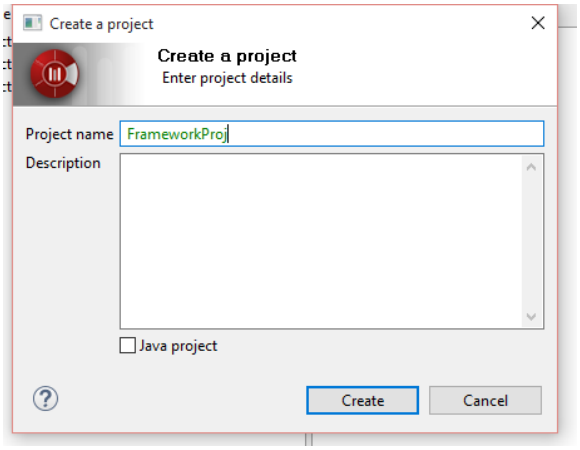

Figure 34: Create a new Project

Now that a new project is created, the developer needs to link the ContextModeller plugin with it to use it. To do this, the developer must click on **Configuration** and **Modules** in the menubar. The user will then be presented with a "Project Configuration" window showing the modules currently used in the project. The developer then needs to click on the "Open modules Catalog" button on the right side of the window, shown in the figure below.

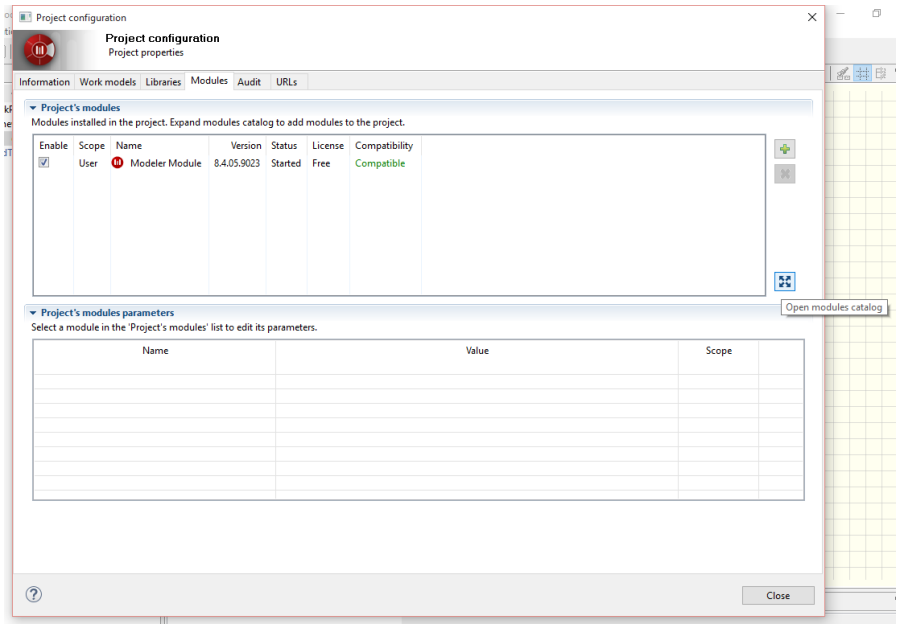

Figure 35: Open Modules Catalog

Once the Module Catalog list opens in the right side of the window, the developer needs to click on the ContextModeller, and click on the "+" button to add it to the project. Once the module has been added, as shown in the diagram below, the developer can close the window.

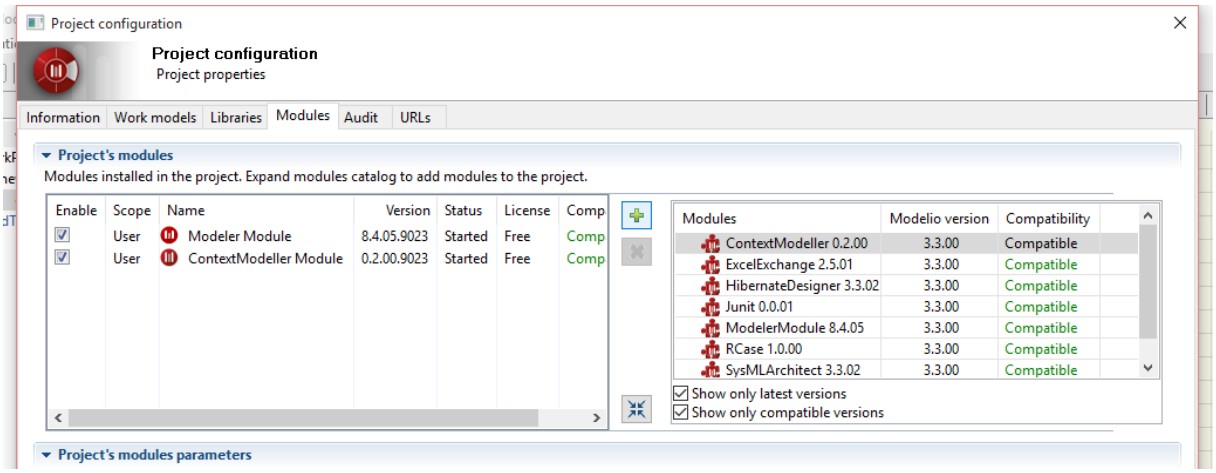

Figure 36: Add ContextModeller module to project

Each context model is created in a ContextModelDiagram. The developer therefore needs to create a new diagram to begin modelling. To add a new diagram, the developer needs to right click on the folder in their project and move the mouse to **ContextModeller Module,** and then to **Diagrams,** finally clicking on **Context Model Diagram.** The developer then has their environment ready to model contexts.

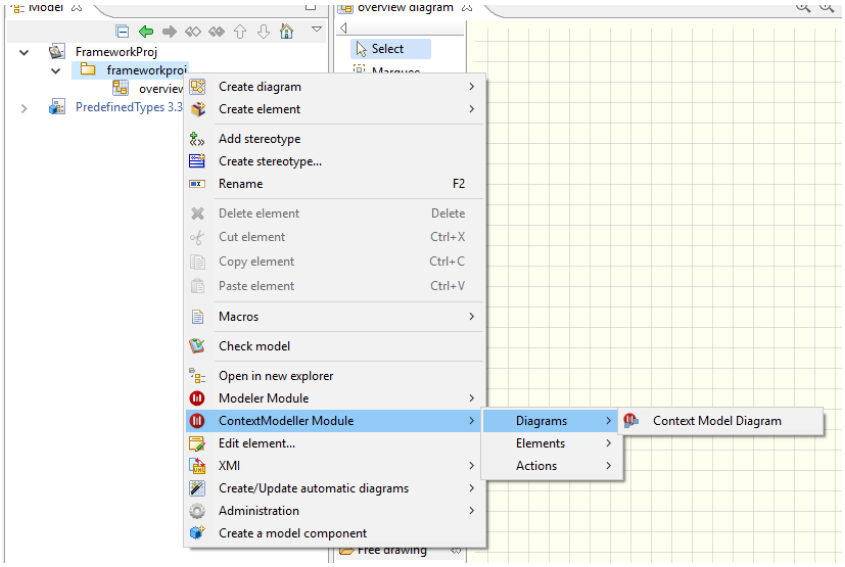

Figure 37: Create new Context Module Diagram

6.2.3 Adding new Context Sources

To add a context source to the context model, the developer can click on the workspace palette, and drag it to a specific location on the workspace. Once it is in the diagram, the developer needs to fill in the different ontological properties. To do this, the developer should click on the context source icon, and then click on the CModeller tab below the main workspace. The developer can enter any RDF prefixes in the “Ont” property, separately multiple prefixes with a comma. Next, RDF data can be included in the “data” property. RDF variables needed in the Context rules can be declared in the this property as regular RDF by the use of a question mark “?” and a name.

Following the addition of context rules, and adding an association between a context source and rule, the developer will need to set the frequency and the data time window of the source. The frequency relates specifically to how often the context rule will use the source data. The data time window on the other hand corresponds to the maximum age of the source data. Each of these properties are set by clicking on the association line between the context source and rule, and going to the **CModeller** tab, just the same as setting the context source properties.

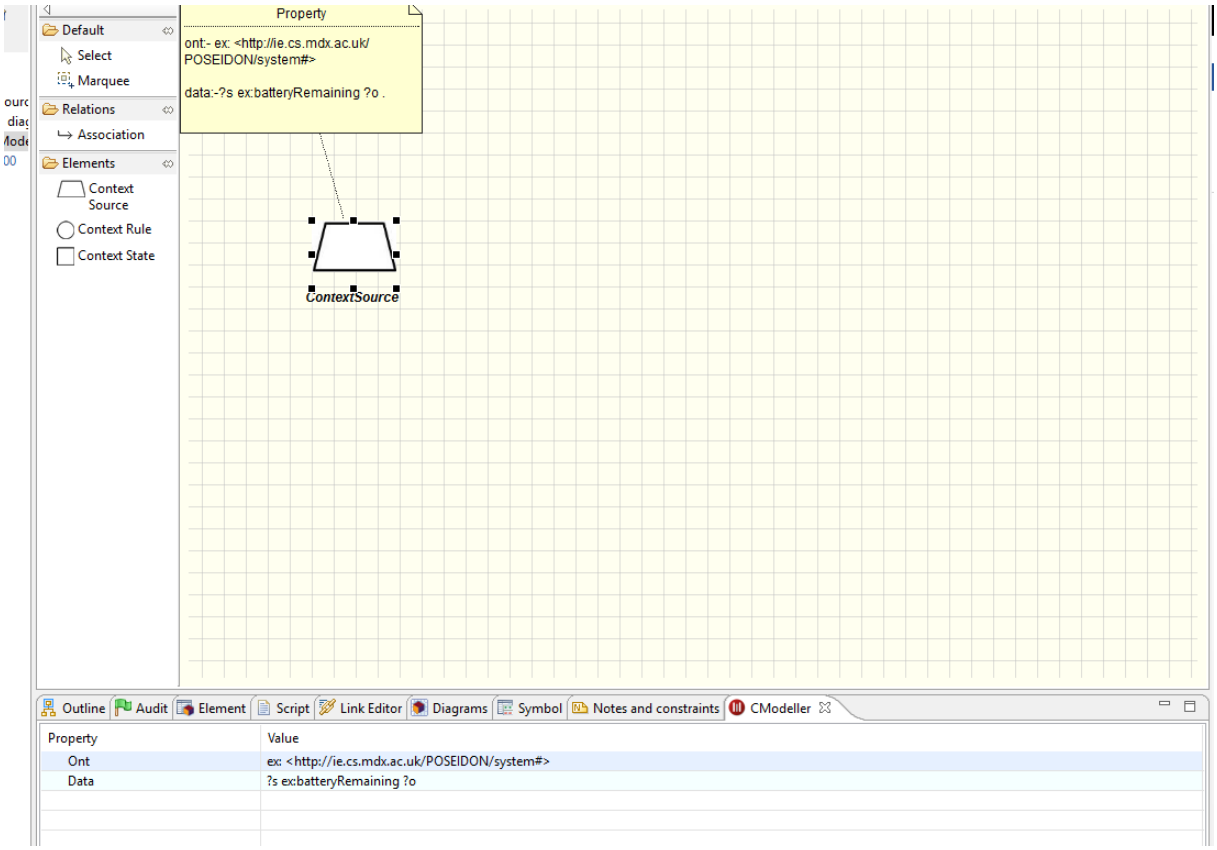

Figure 38: Altering Context Source Properties

6.2.4 Adding new Context Rule

To add a context rule, like the context source, the user needs to drag a context rule from the palette to the workspace. Once the rule has been dragged to the workspace, the developer needs to set the specific properties of the context rule, but clicking on it, and going to the **CModeller** tab. In this tab, the developer can use either, or in combination:

- **Logical Evaluations:** These allow the rule to evaluation specific RDF data variables specified in the context source. As an example for a battery context, we will need to see if the battery level is less than 15 for a Battery_Low state. We have set the batteryRemaining object variable to ?o. We can therefore write the logical evaluation "`?o < 15`".
- **Method:** In SPARQL and SPARQL related languages, often there are different defined methods and functions that can be used within queries. We support up 3 of these methods in any given rule. In the CModeller tab at first, you will only see properties for one method. As you add properties to these, extra property rows will appear for extra methods. For a given method, the developer needs to set the **Method, Method Triple Var,** and **Method Result Expression** fields. In the **Method** field, the developer types the method name, and the method parameters. Method parameters should be RDF variables as set in the context source. As an example, the developer could want to count all instances of a particular value with "`COUNT(?o)`". Next, in the **Method Triple Var** field, the developer needs to place any context source related RDF. This can be helpful if there is a lot of RDF in the context source, as it means the rule will not have to evaluate over all of it to run the Method. If the developer does not enter any RDF in this field, the tool will simply copy all RDF from the context source. Finally, the **Method Result Expression** field needs completing. This is where the developer can evaluate against the result value returned by the stated method. Using the running example

of `COUNT(?o)`, this method will return a single number as a result of counting all instances of the value stored in the variable ?o. The developer can then express a logical evaluation over this value, much the same as a Logical evaluation over values in the context source, described above. As an example, the developer may wish to have a rule that causes a context state if the returned value of `COUNT(?o)` is greater than 2. The developer could then just write "`> 2`" in this field.

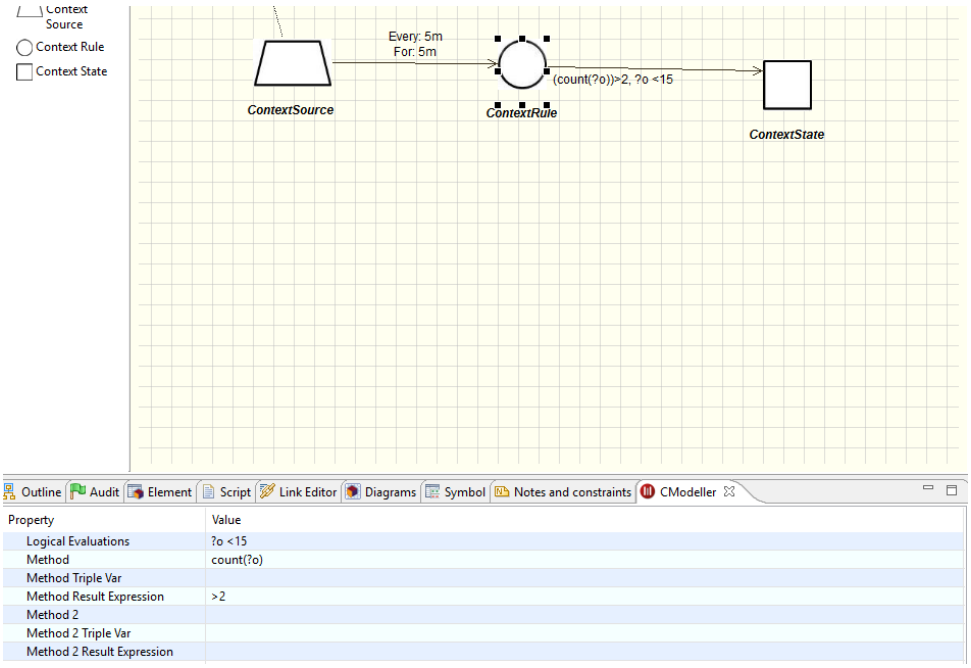

Figure 39: Adding Context Rule Properties

6.2.5 Adding new Context State

A context state is a particular value caused when an associated rule is evaluated to be true. A context state can only be associated with a single context rule. To add a context state, the developer needs to drag the Context state from the palette on the left, to the workspace. Once the context state has been added, the developer needs to set the name of the state by double clicking on the name. This will open a properties window, where the developer can set the state to e.g. Battery_low. This can then be associated with a given context rule.

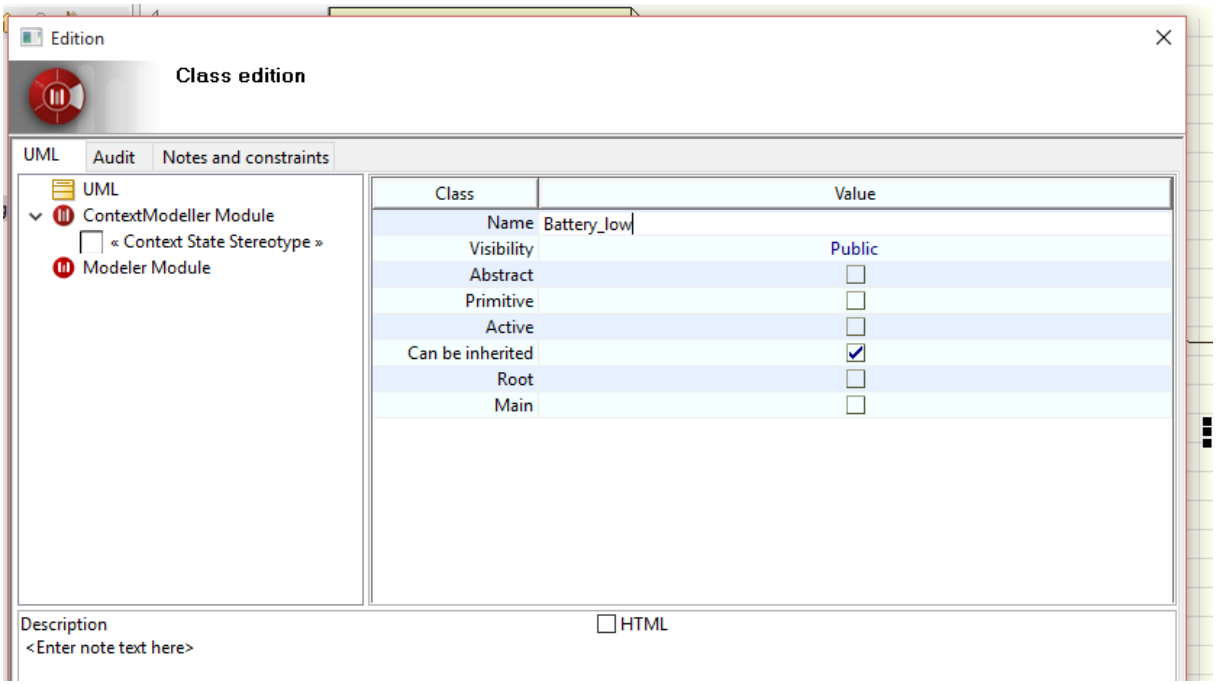

Figure 40: Altering Context State name

6.2.6 Exporting Models for Checking in UPPAAL

Once the developer is satisfied with their model, they can generate a UPPAAL model to test for model correctness. To do this, the developer must right click on their model diagram in the project view in the left of the window. The developer should go to **ContextModeller** and then **Actions,** lastly clicking on **Export to UPPAAL Rules**. This will open a save file dialog, where the developer can save the UPPAAL model with the filename of their choosing in a directory.

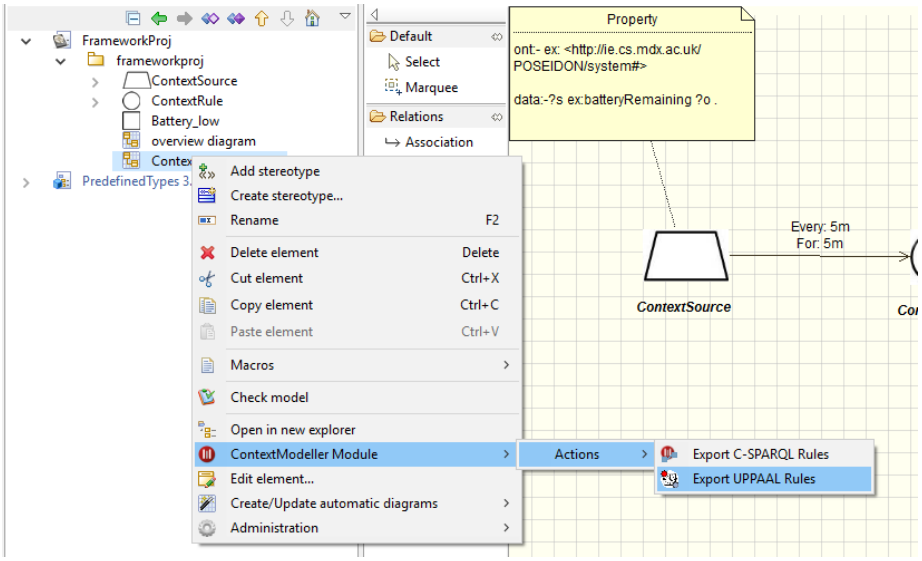

Figure 41: Exporting to UPPAAL Model

6.2.7 Exporting Models to C-SPARQL Rules for deployment

Once the developer has tested their model in the UPPAAL tool, and is satisfied the model is ready for deployment, the developer can export the model to the specific C-SPARQL rules and aggregation rules. To do this, the developer must right click on their model diagram in the project view in the left of the window. The developer should go to **ContextModeller** and then **Actions,** lastly clicking on

Export to C- SPARQL Rules. This will open a save file dialog, where the developer can save the deployment rules with the filename of their choosing in a directory.

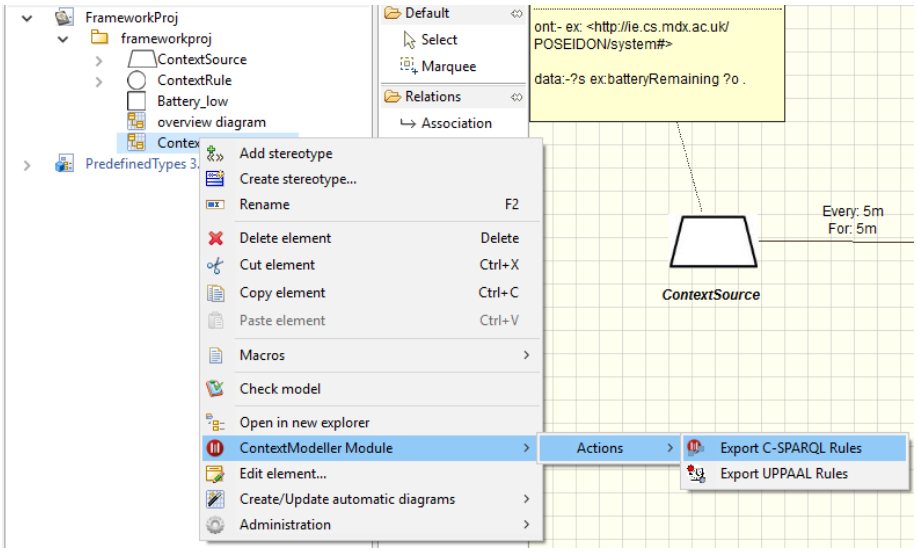

Figure 42: Exporting to C-SPARQL Rules

6.3 Android SDK

Android is the primary mobile platform in the POSEIDON framework architecture, and the mobile platform for which we explicitly support native app development. The Android platform has a solid application framework and development environment, used by a very large community of developers. It is well documented on the official developer website:

http://developer.android.com/

6.3.1 Development environment

For the development work, the Android SDK[22] is the base, containing the code libraries, build tools and much more. Eclipse[23] was the development environment of choice at the start of the POSEIDON project, with plugins for the Android SDK, and was used for initial POSEIDON Android development. However, a new development environment, called Android Studio[24], has taken over as the officially supported environment.

6.3.2 Application framework

Android provides a rich application framework for building applications in the Java programming language. Here we will just briefly mention a few aspects significant to POSEIDON, while the developer should use the official developer website for more.

The Android system includes a data provider pattern. Data provider APIs for calendar and contact data are included in the Android APIs. The data providers act as central repositories for such data on the device, and any application can retrieve and update data through the standardised APIs. Various data synchronisation services exist for the data providers, taking care of synchronising local data with cloud services. Using this pattern, the application developer does not need to know which cloud service is used. The POSEIDON infrastructure makes use of the calendar data provider for access to

[22] http://developer.android.com/sdk/index.html
[23] http://www.eclipse.org/
[24] http://developer.android.com/tools/studio/index.html

calendar data on the device. Although we have so far used Google Calendar for cloud storage of the data, we can later switch to a different calendar service without any change being needed in applications interacting with calendar data through the data provider API. All that is needed is that we provide a synchronisation service which plugs into the Android system.

Another important aspect of the Android application framework is the user interface development. We have written a guide on Android user interface design and implementation, specifically targeting applications for cognitive disabilities. It is currently found in chapter 4 of deliverable D4.5 version 2.

As a POSEIDON-specific extension of the Android application framework we provide libraries for connecting the app to the SmartPlatform, described below in section 6.5.

6.4 Web framework

Web applications are based on the web technologies and standards of HTML[25], CSS[26] and JavaScript[27]. These are the framework basis for such applications, but to facilitate the development of good applications in an efficient way we have looked for existing web development frameworks to use in the POSEIDON web application and include in the POSEIDON development framework. A web application should be able to run in the browsers of all modern platforms. It should be responsive, which in this context means that it must be able to adapt to all screen sizes. This is important for the POSEIDON framework, as we want the same applications to be available on both mobile and stationary systems.

6.4.1 AngularJS

AngularJS is a flexible JavaScript framework for extending the HTML vocabulary. We have used it to develop the POSEIDON web application, to easily separate view, data model and flow. The framework and its documentation is found here:

https://angularjs.org/

6.4.2 Bootstrap

Bootstrap is a free and open-source collection of tools for creating websites and web applications, originally developed at Twitter. It has good support for responsive design, to make sure applications work on all screen sizes. It has style sheets which are compiled into the CSS which controls the look of the web site, greatly simplifying the design work and making it easy to change the look and feel at any time. The Bootstrap CSS Library adds standard interface components to create consistent UI. Bootstrap and its documentation is found here:

http://getbootstrap.com/

6.4.3 POSEIDON web application code

The POSEIDON web application for carers is made with AngularJS and Bootstrap. The application code is modular, with the application functionality separated into reusable components and modules in a structured way. The application is defined by an index file which includes the desired components, settings and branding customization. All this is done to make it easier to update and reuse the code. The plan is to make the application code available as open-source as a basis for continued development.

[25] HyperText Markup Language: https://en.wikipedia.org/wiki/HTML
[26] Cascading Style Sheets: https://en.wikipedia.org/wiki/Cascading_Style_Sheets
[27] JavaScript: https://en.wikipedia.org/wiki/JavaScript

6.5 SmartPlatform client libraries

Client code components for connecting to Tellu SmartPlatform have been developed as part of the project framework. These handle connecting to the REST API and the edge for submitting raw data. These components are made freely available in the developer section of the project website, and can be included in client applications, middleware and server applications, enabling these to connect to the SmartPlatform service without using the server-side API directly.

Here is a list of the artifacts:

- **findit-rest-lib:** A Java library (jar file) for the SmartPlatform REST API, to facilitate data access.
- **findit-rest-android:** An Android-specific extension of the rest-lib, adding asynchronous transaction handling.
- **findit-edge-lib:** A Java library (jar file) for the SmartPlatform edge, to facilitate posting events to SmartPlatform instances.
- **findit-edge-android:** An Android-specific extension of the edge-lib, adding asynchronous transaction handling.
- **findit-lib-demo:** Source code for an Android application which demonstrates the use of both the rest and edge libraries.

Documentation is also provided for the libraries.

There is an ongoing effort in the last year of the project to refactor more of the POSEIDON prototype Android application into general open-source libraries for Java and Android, to provide more functionality to Android developers. Results of this effort will be published on the project website when ready.

6.6 Interactive Table API

For the POSEIDON prototype the Interactive Table is usable in combination with keyboard and mouse of the PC as interaction device. The Moneyhandling Training and the Navigation Training, two services of the stationary POSEIDON system, are developed in such a way, that they can be used with the Interactive Table.

The connection between the POSEIDON application and the Interactive Table can be done using the Interactive Table API. For the POSEIDON application implementation we have used Unity as developing tool. Here the Interactive Table API is used. Additionally to the API description, we provide a very simple example project which demonstrates the connection of Unity to the Interactive Table. This example project, as well as the Interactive Table API documentation description are available on the POSEIDON project webpage.

In the following section, we describe the example project which integrates the POSEIDON stationary system with the Interactive Table.

6.6.1 Connect Interactive Table API - Unity Example Project

The example project is a very simple demonstrative Unity application, in which a cube is rotate to the right or to the left, controlled by the interactive table. When one starts the application one faces one side of the cube. By holding the hand in the air above the interactive table, the cube turns and one can see the differently colored sides of the cube.

This code sample is available on the POSEIDON website.

In the code the interactive table is referred to by its developer name CapTap, derived from capacitive table which is based on capacitive sensing.

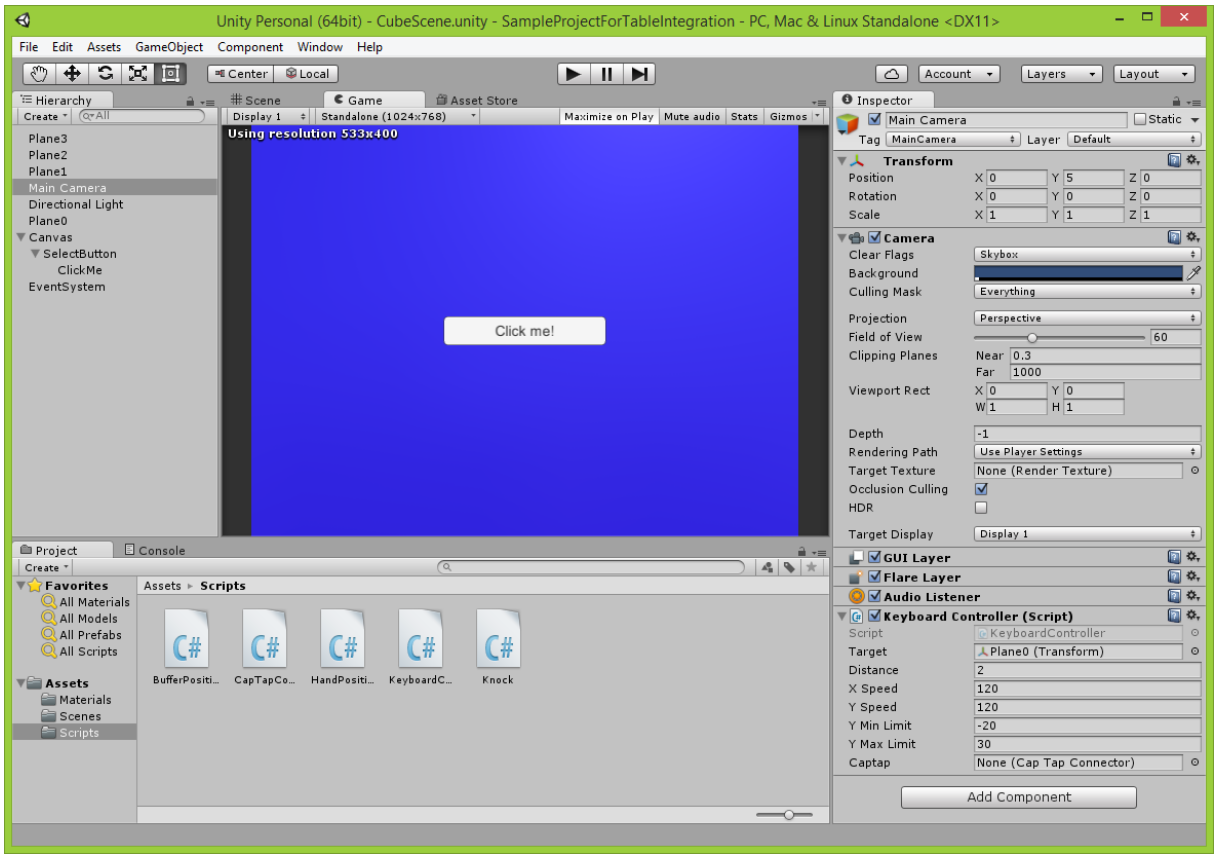

Figure 43: View of Unity development tool with opened sample project.

Browsing through the project, one finds the “Main camera” object. In the inspector view, the “Keyboard Controller” script is attached to this game object, like shown in Figure 43.

```
captap = this.gameObject.AddComponent<CapTapConnector>();
//connect to captap
if (captap == null) Debug.Log("captap war null");
Thread worker = new Thread(captap.startClient);
worker.Start();
```

Figure 44: Code sample from KeyboardController script

This script attaches a captap connection to the current game object. The captap is represented by the type “CapTapConnector”. In the code extract presented in Figure 44 the “startClient” process of the CapTapConnector class is started asynchronously in a separate thread.

When initiating the CapTapConnector, a hand position is initialized in the constructor. In this case it is the recentLeftHand, which holds the last known position of the left hand. See Figure 45.

```
public CapTapConnector()
{
    recentLeftHand = new HandPosition (-1, -1, -1);
    //Logging path
    fileName = "/logging-" + DateTime.Now.Day + "-" +
}
```

Figure 45: Code sample from CapTapConnector script

The CapTapConnector is a class which establishes the TCP/IP connection with the CapTap. This is done in the "startClient" process. First, the hostname of the PC inside the CapTap and the communication port is initialized, see Figure 46.

```
public void startClient() {
    string hostName = "pccaptap";
    int port = 1234;
```

Figure 46: Code sample from CapTapConnector script initializing the hostname and communication port of the TCP/IP connection to the CapTap

Then, by using the interactive table API, different parameters of the table have to be set. For example using the SETPORTS command to set specific USB ports which should be used to connect to the table. Subsequently some important values are set. These commands are explained in the API of the CapTap and in D4.3 Interactive Table.

- The threshold for the capacitive sensing using the command SET_THRESHOLD.
- The erode value using the command SET_ERODE.
- The sound threshold using the command SET_AUDIOTHRESH.
- The audio timeout using the command SET_AUDIOTIMEOUT.
- Turning off the function of sending the images which the table computes to the pc setting the parameter EN_SEND_IMG to false.
- The command to start the table using [START].

After starting the table software the table needs to be initialized by moving the hand above the whole surface of the table. This process is called calibration and is used to detect the maximum possible sensor value which can be reached by touching the table. These values are retrieved by calling the GET_MAX command. This initialization has to be done, because the values which are measured change on air humidity, environment and other factors, which can't be excluded otherwise.

The following endless while loop checks the incoming messages from the table by parsing the buffer. Once a valid command is identified by searching the content marked by parenthesis [ ], the message parameters are parsed and valid commands executed.

Especially the commands POS and KNOCK are handled, because they are needed for the application. The KNOCK is used to click the button on the side of the cube.

For POS the parameters are used to set the parameters of the current HandPosition, see Figure 47. For this application it is enough to use the left hand, the POS message includes both hands if it detects two, if not, any hand is the left-most hand and is therefore assumed as the left hand.

```
//Parse the POS-Message
else if (parts.Length == 7 && parts[0].Equals("POS"))
{
    //Debug.Log(msg);
    recentLeftHand = new HandPosition(Convert.ToInt32(parts[1]),
        Convert.ToInt32(parts[2]), Convert.ToInt32(parts[3]));
}
```

Figure 47: Code sample from CapTapConnector script updating the HandPosition by the parameters of the received POS command.

Going back to the KeyBoardController the "LateUpdate()" the Handposition parameters are checked. For this case, the table area is divided in 3 areas, two of them active. In the code the x component is checked. If it is smaller than 100, then the left area is active. If it is higher than 270 then the right area is active. The corresponding code is shown in Figure 48.

```
//detect if left or right active area. 100 and 270 as thresholds for x coordinate
int leftright = 0;
if (captap.recentLeftHand.x < 100)
    leftright = -1;
else if (captap.recentLeftHand.x >= 270)
    leftright = 1;
```

Figure 48: Code sample from KeyboardControllers script deciding the active area, controlling the view turning direction to the right or inverse to the left.

Running this sample code, will turn the cube while holding the hand above the surface of the interactive table.

The used API of the interactive table can be found in the attachment of D4.5 and is also published on the POSEIDON webpage.

6.7 Unity library for connecting to POSEIDON infrastructure

A small unity library provides client code components that connect a Unity application to Tellu SmartPlatform and file server using REST APIs. This handles submitting and enquiring for resources (JSON files or regular files) on the file server, and the necessary authentication. It is being made available through the developer section of the project website.

Here is a list of important functions:

- `public IEnumerator Login (string username, string password)`

Parameters
username – the username used for login
password – the password used for login
Description
If authentication succeeds writes data about authentication in smart-tracker-token.txt; if authentication fails, returns null

- `private IEnumarator GetAsset ()`

Parameters
none
Description

If it succeeds writes smart-tracker-id.txt, smart-traker-asset.txt; if it fails returns null

- `public IEnumerator Upload (string path, string name, string type)`

Parameters:
path – path of the file to upload
name – name of the file to upload
type – type of the file to upload
Description
If it succeeds, returns www; if it fails returns null

- `public IEnumerator ListResources (string type)`

Parameters:
Type – type of resource to fetch
Description:
If it succeeds, returns resource list; if it fails returns null

- `public IEnumerator ChangeResource (string path, string resourceID)`

Parameters:
Path – path of new resource
resourceID – resource ID of the resource that needs to be changed
Description:
If it succeeds, returns www; if it fails returns null

7 Conclusions

With this third version of the development framework deliverable, we have concluded the framework process. We have thoroughly analysed and discussed the framework requirements first listed in deliverable D2.1. We have developed and documented system development methodologies. Based on the framework requirements and the functional and non-functional requirements for the prototype system, we have developed three iterations of prototypes. The architecture and infrastructure of the framework comes in part from the available technologies and in part from the prototype developments. We defined a framework for the first prototype iteration, based on the SmartTracker service as the central hub. For the second prototype the framework was extended with middleware for the stationary and mobile systems, primarily to support context awareness directly in these systems. For the final version of the framework, the architecture has been extended with services formalised from the prototypes, to create an infrastructure that supports all the functional aspects of the POSEIDON solution and to which other applications can connect. Tools and components are being provided to developers.

Although it has been a much more challenging process than envisioned, and not all ambitions have been fully realised, we have succeeded in producing a development framework which supports the POSEIDON prototype system, allows this system to be maintained and extended in the future, and allows third-party developers to develop new applications which integrates with the system. We hope that the framework can be of use to other developers in the assisted living field and that it will contribute to the development of more services for people with Down syndrome.

Bibliography

Augusto, J. C., Callaghan, V., Cook, D., Kameas, A. and Satoh, I. (2013). Intelligent Environments: a manifesto. *Human-centric Computing and Information Sciences*, 3, p.12. [Online]. Available at: doi:10.1186/2192-1962-3-12.

Barbieri, D. F., Braga, D., Ceri, S., Valle, E. Della and Grossniklaus, M. (2010). C-SPARQL: A Continuous Query Language for RDF Data Streams. *International Journal of Semantic Computing*, 04 (01), World Scientific Publishing Company, p.3–25. [Online]. Available at: doi:10.1142/S1793351X10000936 [Accessed: 20 March 2015].

Behrmann, G., David, A., Larsen, K. G., H??kansson, J., Petterson, P., Wang, Y. and Hendriks, M. (2006). Uppaal 4.0. In: *Third International Conference on the Quantitative Evaluation of Systems, QEST 2006*, 2006, p.125–126. [Online]. Available at: doi:10.1109/QEST.2006.59.

Jones, S., Hara, S. and Augusto, J. (2015). eFRIEND: an ethical framework for intelligent environments development. *Ethics and Information Technology*, 17 (1), p.11–25. [Online]. Available at: doi:10.1007/s10676-014-9358-1.

Appendix: Architecture evolution

The POSEIDON framework architecture has been through major changes in the project, with two major iterations before the final one. This appendix has descriptions of these two iterations, reworked from the first two versions of deliverable D5.1. It presents the evolving picture of the architecture of the POSEIDON solution and framework.

First iteration

This was the initial stage of the project, where an initial architecture was established based on the technology we had and the need to start developing a prototype.

Design process

After the requirements gathering and analysis, the framework process leading to the initial version of this deliverable consisted of providing the necessary development framework for a first prototype, as well as planning ahead for the richer framework needed for the final POSEIDON solution. The Milestone *First Integrated Prototype* (MS8) was scheduled for month 9 of the project – the first of a total of four prototype iterations. This first tentative prototype came very early in the project, and was an important starting point for further development. The idea was to get both a tablet application with some functionalities as well as a first version of a home system implemented so that we could gain experience and have something to show the user groups and can start getting feedback. These two parts also needed to be connected through a server side.

It is important to keep in mind that the experimental nature as well as the very short time frame for implementation meant that the first prototype didn't try to put into place a definitive POSEIDON development framework. The focus in this stage was to provide what was necessary to support the implementation of this first prototype, using the technology with which the implementing partners had prior experience. Then the framework process would continue after developing the first prototype, given our experience from this implementation.

Prototype 1 architecture

Prototype 1 consists of a mobile system and a stationary system for the primary end users to interact with. As these are demonstration systems and not for real use, they are not connected together. However, an architecture connecting these systems to Tellu's SmartPlatform was designed. This architecture is shown in Figure 49.

Tellu's SmartPlatform (SmarTracker service in the figure) is the main framework component in this architecture. It is the central hub in the solution, with mobile and client applications being able to connect to it to share data. It is a place to implement server-side storage and logic. Applications can post data to the sensor edge. This data is stored, and it is processed by the rule engine. Applications can access the aggregated data in the SmartPlatform data model through the REST API. So the SmartTracker service was provided for server-side data storage, for server-side logic and context awareness and as the hub in the system, connecting the mobile and stationary clients. The web interface of the service is used for configuration and administration.

A first prototype mobile application was developed for the Android platform. The primary features are tracking and navigation, in addition to being a test-bed for context awareness. While navigating, the user's position, as well as the battery/power status of the device, is sent to SmartTracker for processing. The application is also connected to the SmartTracker REST API. For additional services in the application, such as routes for navigation and calendar for day planner, it can be connected to external services such as those offered by Google.

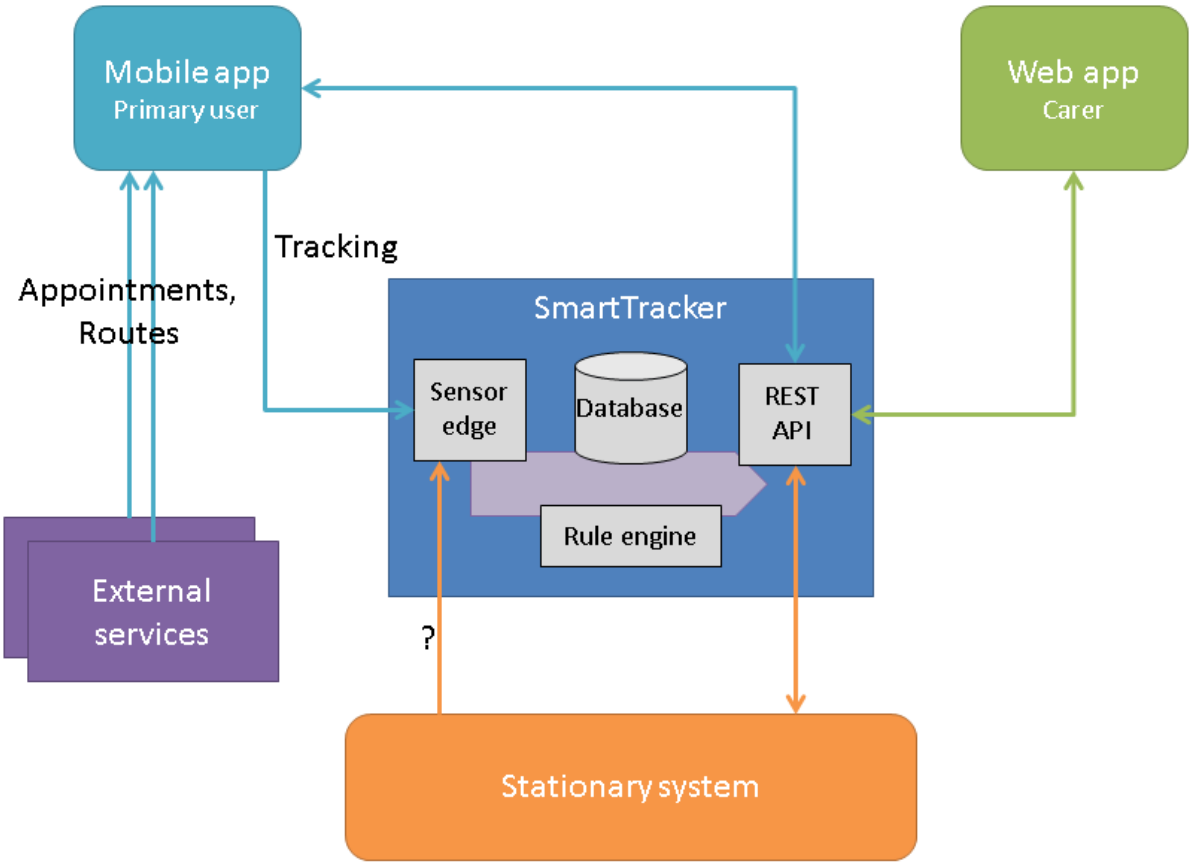

Figure 49: Prototype 1 conceptual architecture

The conceptual architecture also shows how a web application can be added to the system. Based on SmartTracker's API, such a web application can offer monitoring of the primary user by presenting the tracking and context data, and administration of the SmartTracker account, for instance editing preferences.

The stationary system is what is running on the home computer. Here prototype applications were developed for virtual reality navigation training and augmented reality guidance. The idea in the conceptual architecture is that this part can also connect to SmartTracker, to access preferences and context data, and possibly also to post context observations.

Second iteration

The second iteration of the architecture was developed for the second prototype. An updated D5.1 was delivered 15.05.2015.

universAAL

universAAL[28] is an open platform for Ambient Assisted Living technologies and services that was created based on the input gathered in a number of previous projects and aims to consolidate aspects into a sustainable, semantic, interoperable platform that is open and developed by a growing community. It was a natural candidate for the POSEIDON framework, as it is an extensive framework

[28] http://universaal.org/

for AAL solutions which consolidates numerous previous platform projects into a shared, generalized architecture and middleware.

universAAL allows for multiple communication methods allowing for interoperability between different instances of universAAL in different locations, and external clients. These include the following:

- **Web services**: universAAL enables the ability of external clients to invoke services within AAL spaces SOAP web services. Service requests sent by the client are formatted as Turtle Strings in RDF. Each request is accepted and translated into universAAL service requests by the server. These requests are then sent and handled over the Service Bus before service responses are returned to clients using SOAP responses.
- **AAL Space Gateway Communication**: This type of communication allows different geographically positioned AAL spaces to communicate. Using this communication, services and contexts can easily be shared among different AAL spaces. Each AAL space is connected through an AAL space gateway.
- **Node-to-Node Peering:** Different universAAL nodes can connect to each other in the same network/AAL space. This is handled by service announce and discovery using the SLP protocol. This protocol is handled using jGroups and jSLP, and allows for AAL spaces to be created and announced, discovered, and connected to.

universAAL was intended to be part of the prototype 2 framework. The process of learning about universAAL was started after the first prototype iteration. The project signed an agreement to be associated with the ReAAL project, which is working on commercialisation of the universAAL framework. This agreement meant we committed to trying to use universAAL, and that the ReAAL project would provide us support in doing so.

Design process

An early roadmap for the architecture and framework process was made in the first iteration, mainly stating the intention to add universAAL to the framework. Figure 50 shows a rough sketch of the prototype 1 and 2 systems. While the prototype 1 part roughly corresponds to the final prototype 1 conceptual architecture, the idea was that universAAL would take over the role as communication hub from SmartTracker. As SmartTracker is a service in its own right, not just middleware, it is not ideal as *the* communication framework in POSEIDON. It is better if it can be a pluggable component in the system, which could also be replaced or augmented by other services.

The idea for prototype 2 and beyond, based on our requirements discussion, was to use some form of message broker so that new components could be connected without relying on SmartTracker's APIs and without going through SmartTracker when this is not necessary. So rather than being directly connected to SmartTracker, we saw the components all connected to some form of bus. It was thought that universAAL would play this role.

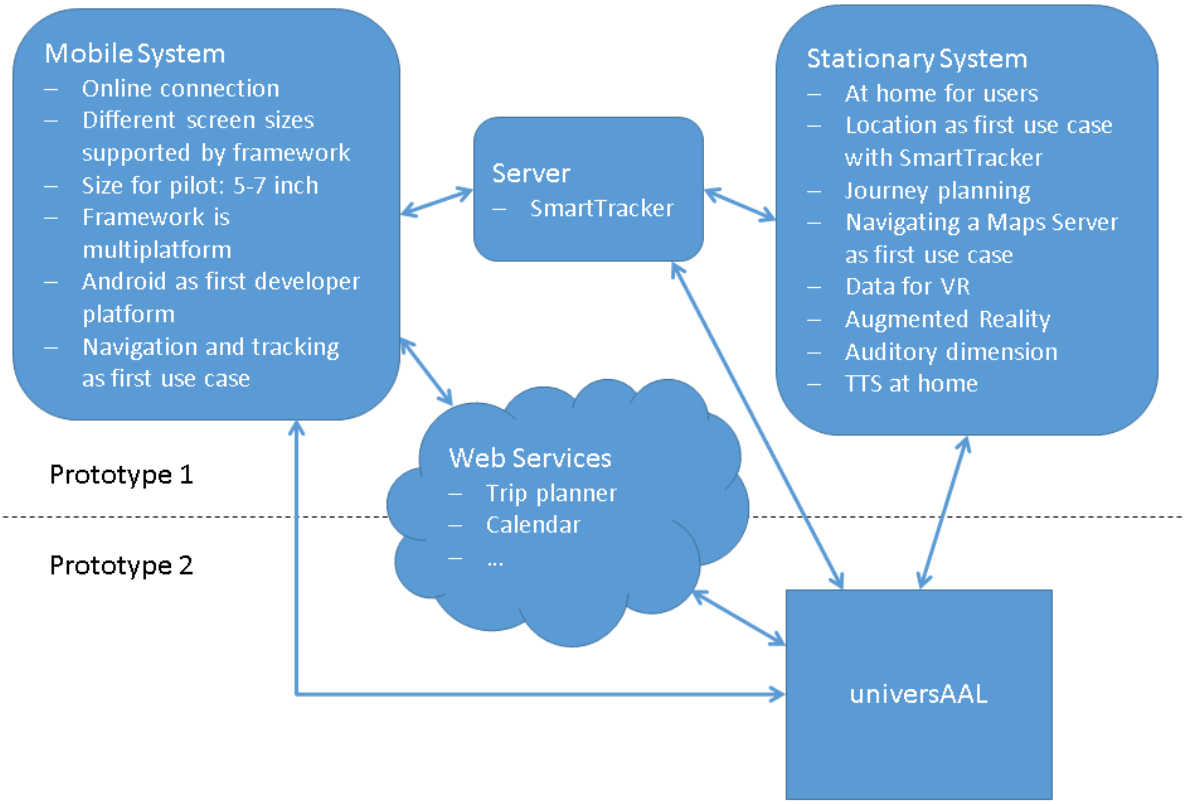

Figure 50: Early system architecture roadmap

Work was done to learn universAAL and test its usage in the POSEIDON context. We were in dialogue with the universAAL developers, asking for support and reporting problems. However, as we got to know unversAAL and its components in the second iteration, it became clear that it was not suited to be the main infrastructure for connecting all the POSEIDON components together. The primary purpose of universAAL is smart-home middleware. universAAL instances are deployed on home computers to act as gateways, for the interconnection of different sensors and applications. As such, it is mainly a framework for a stationary system. While it has mechanisms for connecting specific universAAL instances together, it is not suited to provide server-type services such as central storage of data.

The result was to divide the framework for prototype 2 into two main components. Tellu's SmartPlatform continued to be the cloud service part of the framework, being available online for all POSEIDON applications and users to connect to. The other main part was the middleware running locally, on the stationary and mobile systems, and this could possibly be universAAL.

As universAAL is primarily a smart-home middleware, and it has an Android implementation, we tested using it in the communication between the home computer and the mobile device in prototype 2. An open-source POSEIDON Android application was developed as part of the prototype, as a testing platform and to provide a code base and example for third-party developers. Using universAAL middleware on the device and communicating with the home computer for the transfer of route data from the home navigation system was first implemented in the open-source application, and later in the pilot application which is based on a more mature but proprietary code base. This was the planned way to transfer route data in the pilot. However, our pre-testing showed that this approach did not work well enough to be used in the pilot. It was not always possible to achieve a connection between PC and mobile device, and it was not possible to diagnose the

problems, making further pre-testing unfeasible. The installation and configuration was also cumbersome and not user friendly, and we found general stability and maturity issues with the universAAL components. We therefore ended up using an online file server to transfer route data in the pilot, and universAAL has not been used in the system from pilot 1 and onwards.

Prototype 2 architecture

Figure 51 shows the resulting second iteration conceptual architecture. The box labeled *Context middleware* is the middleware running on the stationary and mobile systems, mainly for context awareness. This framework component was still in heavy development at that time. universAAL was still being considered, and the plan was to deploy it on both the stationary system and Android device for the second prototype. However, the ontology and rules were running independently of universAAL.

Note that SmartTracker's rule engine is no longer indicated in the conceptual architecture, as the context awareness logic was now placed in the locally deployed context middleware. Moving this from server to client side was wanted to avoid the lag and possible fallout of connectivity associated with an internet connection. However, SmartTracker's rule engine may still be used at some point, if there is a need for logic which aggregates data from multiple users. The locally deployed middleware instances could be connected to SmartTracker. They could send context observations there so that they are stored in the cloud, independently of the device. This allows sharing this context data between instances belonging to the same user, retrieving the data with the REST API, authenticated by a SmartTracker account set up for the user.

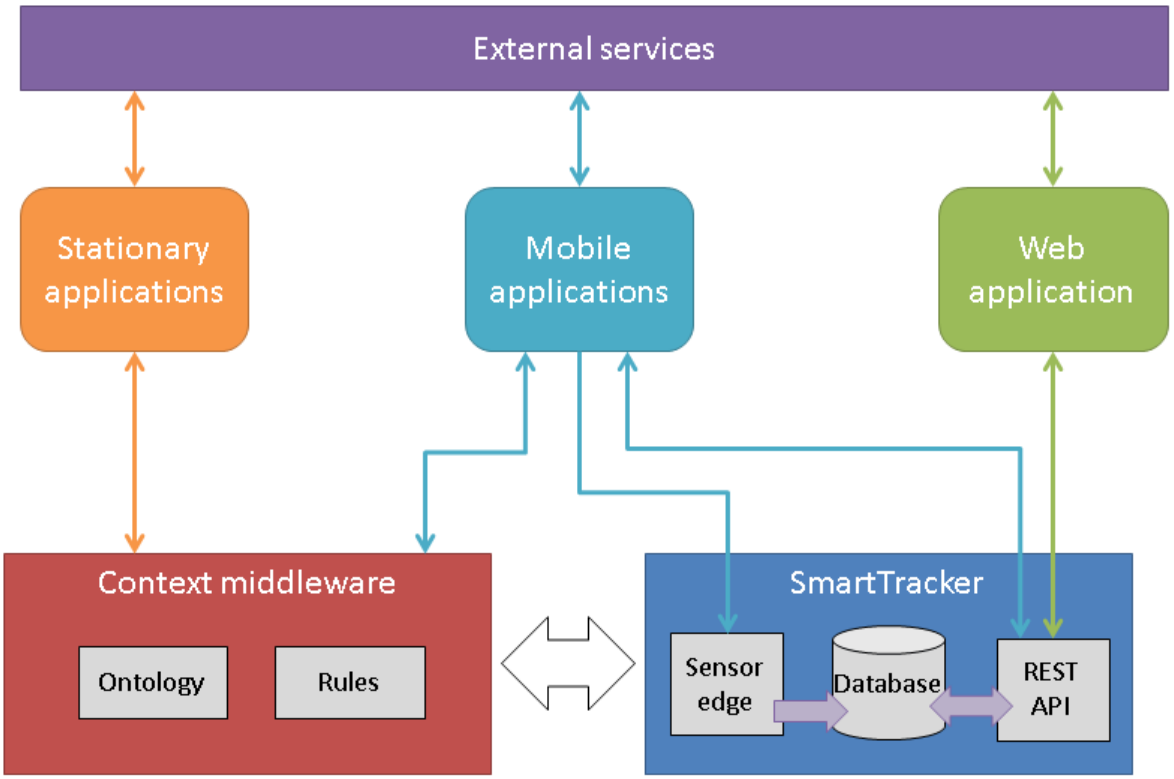

Figure 51: Prototype 2 conceptual architecture

Three types of applications can be connected to the framework in this model. Stationary applications are those running on a PC or laptop, at home or at another location the primary user regularly stays at. Such applications may be connected to the context middleware deployed on the same machine.

As in the first prototype, virtual reality training and augmented reality guidance were the focus of development here, and the interactive table was used as a control device.

Mobile applications are those running on a phone or tablet, and meant to be used anywhere, especially when not at a location with a stationary system. Context middleware is deployed on the same device. A mobile application can subscribe to context changes, and the middleware will monitor context on the device and deliver updates to the application. The mobile application can also provide input to the context reasoner of the middleware, posting data regarding its own state and context. Mobile applications may also connect to SmartTracker over the internet. They can send observations directly to SmartTracker, for the online monitoring. And they can access the SmartTracker data through the REST API, mainly for the user preferences stored here.

This conceptual architecture also states that there needs to be a web application, primarily for secondary users to be able to do monitoring and personalisation from any device and location. This is an end user interface for SmartTracker functionality, presenting the tracking data and letting the user interact with the preferences stored here. It was not yet clear if parts of this web application should be considered as a part of the framework or not. Tellu has made available our web application framework as open-source. This builds on some state-of-the-art open-source web application libraries and frameworks, adding an overall structure and modules for SmartTracker interaction. Using this, other developers will be able to create their own SmartTracker-enabled applications.

The applications needed more services than we were supplying with the POSEIDON framework components at this stage. The following services were seen as especially important:

- **Calendar data storage:** Secondary users are to be able to plan activities for the primary user with the POSEIDON system, and the planned events need to be available throughout the system. This means we needed cloud storage of the events. For the second prototype we were using Google Calendar to provide this, and it would later be defined as part of the framework.
- **Route planner:** The navigation aspect is another of the primary functionality set selected for POSEIDON. It requires both automated route planning and the ability to augment plans with personalised instructions. Google Directions was used as the automated route planner for prototype 2. Personalisation was done in the stationary system. An automated route planner is outside the scope of POSEIDON, and was also found to be unwanted in the first pilot. A route format and place to store routes were defined as part of the final framework.
- **Media storage:** Media such as photos and video is primarily needed for instructions and guidance to the primary user, configured and personalised by the secondary user. Here we were exploring options. It was not yet clear whether a specific service should be specified as part of the framework, but its importance meant it ended up as part of the final infrastructure.

All types of applications could connect to external services, and use them to share data between applications. For the second prototype, calendar functionality was added to the web application, so that it could be used for planning. The mobile application prototype also got calendar functionality, connecting to the same service. The route planner was used by the stationary system, where routes were personalised. Personalised routes were transferred to the mobile application for navigation. The mobile application could also request new routes directly from the route planner service, for navigation from any starting point, but without the personalisation.