Bachelor Computer Science

University of Amsterdam

CLIsis: An Interface for Visually Impaired Users of Apache Isis Applications

Sander Ginn

June 2, 2016

Supervisor(s): Maarten Marx (UvA), Dan Haywood (Apache)

Abstract

This will be the abstract.

Contents

CHAPTER 1

Introduction

Every application that offers a method of interaction with a human being requires some form of user interface to enable this interaction. But what exactly characterises a user interface? Galitz[23] provides the following definition for 'user interface':

> *The user interface is the part of a computer and its software that people can see, hear, touch, talk to, or otherwise understand or direct.*

Thus, in the context of an application, a user interface is a means of interaction between the user and system based on mutually understood communication methods. These communication methods can range from visual methods such as computer displays to audible methods like speakers.

This is still a very abstract definition of what a user interface does. It does not specify how system information is presented, how the user provides input, what permissions and restrictions the interface must adhere to, et cetera. Nielsen[41] describes an overview of user interfaces throughout history which will provide some practical examples.

When the computer was in its infancy, several primitive user interfaces have been developed. As the use of computers became more widespread, however, the *line-oriented interface* was adopted as a method of interaction. The name is derived from the characteristic that the user interacted with a system on a single line; once submitted, the command could not be modified anymore. As long as the information is well structured this method of interaction still has valid use cases in present times, as it effectively delimits functionality, preventing novice users from running into issues. Assistive technology for visually impaired users excelled with line-oriented interfaces; as all content was character based, it could easily be made interpretable for someone who lacks (good) vision, for example through audio[44].

The successor of line-oriented interfaces is the *full-screen interface.* This interface aims to exploit a greater deal of the screen real estate by enabling the user to interact with more elements than just one input line. The full-screen interface also introduced the concept of menus, where functionality available to the user is listed in a hierarchical structure. Menu design has been researched extensively[42, 33, 21] to determine the optimal balance of depth and breadth. Depth decreases complexity of menus while increasing ease of navigation; the contrary applies to breadth. However, menu-based interfaces also marked the beginning of a decrease in accessibility for visually impaired users, as the information displayed gradually became more catered towards a visual mental model. Visually impaired users tend to mentally model information differently, and thus directly translating a visual interface to a non-visual representation can be extremely difficult to comprehend for the target audience[18].

Finally, the user interface type which is currently the most widespread, is the *graphical user interface* (GUI). The vast majority of modern applications offer a graphical user interface as their primary method of interaction. A key characteristic of GUIs is that interaction is offered through direct manipulation. Rather than issuing a command which exactly specifies those parameters of what has to happen, the user gets continuous feedback of their input commands. The obvious advantage of a GUI is that it can take advantage of the human sight to represent data in a more intuitive way. Research has shown that when the interface adheres to some fundamental principles, a user's cognitive performance increases with a GUI. An example of one of these principles is Miller's Law, proposed in his seminal research on information processing[37]. However, Nielsen also addresses the issue that motivates this research: a GUI sidelines the visually impaired from using an application.

Enabling the visually impaired to use applications despite a lack of or very bad vision has been an ongoing effort since GUIs were widely accepted as the de facto standard[14]. However, it has proven to be difficult to design a GUI that is also accessible without visual cues. A big issue is that standardisation has proven ineffective: even though the United States has passed laws that enforce accessibility of websites[8] and the World Wide Web Consortium has developed the Web Content Accessibility Guidelines[11], research has shown that these guidelines are very often ignored and thus websites are rendered inaccessible for accessibility tools such as screenreaders[36].

Visually impaired computer users often make use of a *screen reader* to translate visual objects into a representation that can be interpreted by assistive technology, such as braille terminals or text-to-speech. However, Lazar et al. found that 4 out of 5 top causes of frustration for blind users of screen readers were related to poor design choices of the software or website developer[34]. As a solution to these problems, we will implement a bespoke interface that does not require the use of a third party screen reader.

This research aims to provide an alternative interface for visually impaired users for applications developed with the Apache Isis[2] framework. This interface will not serve as a replacement for the existing interface nor blend in with it. The interface is derived from the metamodel of the framework and thus content unaware. This means that the interface is readily available for any Isis application and no alterations are necessary to add it to an existing application.

1.1 Research questions

The central research question in this thesis is:

Can a content unaware graphical user interface be adapted to a non-visual interface so that visually impaired users are able to interact with the application? (1)

For the non-visual interface to be useful it is imperative that it offers the same functionality as the graphical user interface. Therefore, we will also answer the following subquestion:

When implementing a non-visual interface, can the integrity of the domain model be maintained while providing an effective and simple method of interaction? (2)

Finally, it is desired that the non-visual interface does not severely impact user performance, as this would imply that it is not a legitimate alternative to the existing interface. Thus, a second subquestion will be answered:

Compared to the standard interface, how is performance affected when the user employs the new interface? (3)

1.2 Overview of thesis

A short overview of what this thesis will describe is given.

CHAPTER 2

Theoretical background

User interface design is a thoroughly studied discipline with strong roots in psychology. In the 1980s GUI development exploded due to better hardware[39]. This meant that traditional user interfaces had to be redesigned to accommodate to the graphical features of the modern computer. Chapter 1 described the issues visually impaired users experienced with this new method of interaction. Section 2.2 expands on some existing efforts on improving usability for the visually impaired. Furthermore, section 2.1 will describe what Apache Isis entails. Finally, section 2.3 briefly addresses some pitfalls when a user interface is adapted to a new form.

2.1 Apache Isis

As a software engineer, picking a framework for developing web applications can be a tedious process. There are dozens of frameworks for Java alone, with the oldest and most adopted one being Spring[9]. The vast majority of these frameworks are based on the *model-view-controller* (MVC) pattern, where the view displays information to the user, the controller processes interaction between the user and the view, and the model contains the information and logic that manipulates this information[35]. The relations between the components are depicted in figure 2.1.

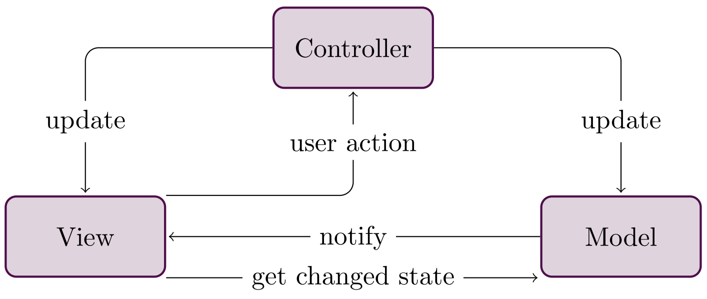


Figure 2.1: Interaction between the components of an MVC application

While the MVC pattern itself has a lot of advantages, it has received criticism in the context of web development. The concept of separating business logic from presentation logic is often not adhered to in web applications, resulting in controllers that are contaminated with logic that should live in the model[6]. Trygve Reenskaug, who introduced the MVC pattern while working at the Xerox Palo Alto Research Center in 1978, concluded that true MVC-based applications were hard to achieve[45]:

> *"The conventional wisdom in the group was that objects should be visible and tangible, thus bridging the gap between the human brain and the abstract data within the computer. This simple and powerful idea failed ... because users were used to seeing objects from different perspectives. The visible and tangible object would get very complex if it should be able to show itself and be manipulated in many different ways."*

2.1.1 Domain-driven development

Even though Reenskaug did not consider the MVC pattern to be successful, the philosophy of displaying information in a manner that feels natural to humans is still a very valid one. Many methodologies have been developed in an attempt to capture this philosophy, of which *domain-driven design* (DDD) is a well-known example. Conceived by Eric Evans in 2004, DDD strives to streamline and align the domain experts (the business experts) and the technical experts (the developers). By clearly defining what terms to use for what concepts and ensuring that both parties understand what they mean, a *ubiquitous language* is formed, and information asymmetry can be avoided[20]. The *domain model* is then built solely with terms in the ubiquitous language. The key to success is then to apply model-driven development; by developing the software while strictly adhering to the specified domain model, all stakeholders are able to comprehend the content of the software. When the domain model proves to be insufficient to implement a new feature, the stakeholders should work together to extend the ubiquitous language so that the domain model becomes complete again. This creates a bidirectional relationship between the domain model and the code: when the domain model changes, so does the code and vice versa.

2.1.2 Naked Objects pattern

One of the frustrations often expressed regarding DDD is that while the ubiquitous language combined with the derived domain model may very well tackle the problem of goal diversion, it also increases overhead greatly. Consider figure 2.2: the domain model is represented in the business logic layer. As the domain model will be refined over the course of time, and that any changes to the domain model must be reflected in the code. This means that any modifications to the domain model will have to be applied to the other layers, too.

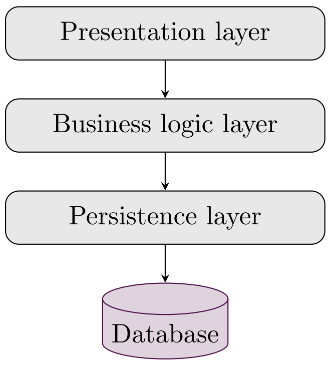


Figure 2.2: The main application layers in DDD

To address this issue, Richard Pawson designed the *Naked Objects* (NO) pattern, based on three core concepts[43]:

1. Business logic should only reside in domain objects[1]
2. The user interface is an exact representation of the domain objects, and thus is an object-oriented user interface
3. The user interface is derived completely and fully automatically from the domain objects through reflection[2]

These three concepts combined provide a means of developing software that robustly follows the philosophy of DDD. The first two concepts ensure that the software implements the domain model and prevents the use of technical language. The application, whether prototypal or deployed, will feel familiar to the stakeholders and developers alike. The third concept is not

[1]A domain object is an entity that holds relevant data, in order to transfer the data between the application layers.

[2]Reflection is a language's ability to inspect and dynamically call classes, methods, attributes, etc. at runtime.

```
@DomainObject
public class SimpleObject implements Comparable<SimpleObject> {
    public TranslatableString title() {
        return TranslatableString.tr("Object: {name}", "name", getName());
    }

    @javax.jdo.annotations.Column(allowsNull="false")
    @Property
    @Getter @Setter
    private String name;

    public TranslatableString validateName(final String name) {
        return name != null && name.contains("!") ?
            TranslatableString.tr("Exclamation mark is not allowed"): null;
    }
}
```

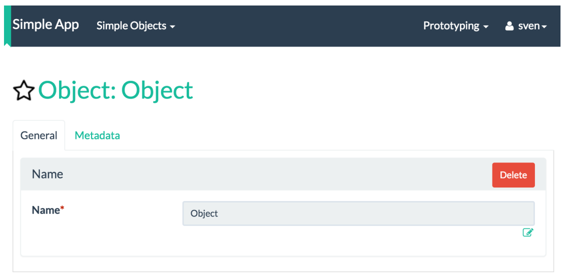


Figure 2.3: A very simple example of how the user interface is derived from code

necessarily related to DDD, but does take away the burden of developing the presentation layer and thus places the focus on the business logic: i.e., the domain model.

In the foreword of Pawson's thesis, Reenskaug states that the NO pattern could be seen as an important extension and implementation of the original MVC pattern.

2.1.3 Java Framework

Apache Isis is a framework written in Java that implements the NO pattern. The core strength of the framework is that behaviour which would normally be defined in controllers in classical MVC applications is automatically generated by reflection. A major benefit of this feature is that prototyping is greatly simplified; the business logic is virtually all that needs to be programmed, and the framework does the rest. This means that project managers, product owners and other parties involved can be presented with working prototypes rather than charts and specifications, and it relieves developers from wasting precious time on coding the visible part of an application - code which often has limited reusability in a later stage of development.

This does not, however, imply that Apache Isis is merely useful for prototyping. The automatically generated interface is perfectly suitable for a lot of business oriented applications. Estatio[5], an estate management application built on Apache Isis commissioned by a listed real estate investment firm, makes use of the standard wicket viewer. In case a given application demands a bespoke user interface, it can be designed to use the REST API which ships with Apache Isis, which we will use to implement our interface.

Due to the underlying NO paradigm that the user interface is derived from the domain objects, Apache Isis is exceptionally suitable for this research project. Every bit of information that dictates what is visible to the user is found in the code that describes the functionality of the software, and thus we are able to develop a new user interface that retrieves everything it needs from what the developers have written regardless of the interface they intend to use.

2.2 Previous work

User interfaces make use of at least one of the traditional human senses to convey information. It is evident that GUIs are mainly based on exploiting human vision. As visually impaired people lack this sense, they have to interact with software through what remains: the auditory, somatic, gustatory and olfactory senses. The latter two senses are relatively unexplored in terms of human-computer interaction. While there have been developments such as 'smicons', representing icons through odour[28], and a display that represents rudimentary information through scents sprayed in the direction of the nose[48], it has proven difficult to employ these senses in a practical manner. The main reason is that smell and taste are chemical rather than physical sensations and thus a lot harder to synthesise, let alone accurately[32]. Therefore, effectively two senses remain available: the auditory and somatic senses.

Exploiting the hearing sense is a good way of making software usable for a large audience, as it only requires a form of audio output to function and virtually all modern devices have either built-in speakers or a headphone jack. Most auditory user interfaces synthesise text to speech and use auditory icons for basic operations, such as the sound of crumpling paper on a delete operation[40]. While this is regarded as one of the easiest ways of increasing accessibility of software, it still leaves a portion of the auditory sense unused, as it does not take advantage of spatial sound. Research has shown that basic elements of interfaces, such as menus, can be displayed very effectively in a spatial setting[22].

Software can also interact with the somatic sense through haptic feedback. Tactile feedback is a form of haptic feedback that relies on the sensation of touch, vibration and pressure. Visually impaired people who have learned braille can take advantage of output devices called braille terminals combined with screen reader software, which translates text on the screen to braille. This does not, however, offer practical support for graphical elements. With the advent of mobile technology, research in the field of tactile user interfaces has increased. The majority of these mobile devices have touch screens and vibrating motors, allowing gesture-based input and haptic feedback[27, 49].

2.3 Pitfalls in user interface adaptation

The concept of universally accessible user interfaces has gained a lot of traction since the widespread adoption of GUIs. It has proven difficult, however, to adapt an interface in such a way that it becomes both universally accessible and maintainable. The foremost shortcoming of adapting an existing user interface is that it is mainly an a posteriori change, and thus any future changes to the software will require modifications to all existing user interfaces. Another major issue is that it might be very difficult or even impossible to adapt certain functionality due to the constraints of the new interface[19]. Third, a lack of standardisation can increase the required amount of labour to such an extent that it is no longer feasible to perform the adaptation[38]. In the present time, it is generally accepted that the proactive approach of making design choices with universal accessibility in mind is more convenient and successful than the reactive approach of adapting an existing design[46]. As this is not within the scope of this research, we will not address this in detail.

CHAPTER 3

Methods

We have selected several methods for implementing the new interface. First, section 3.1 will describe the method applied to the existing interface in order to form a theoretical basis for the implementation. Section 3.2 then outlines what experiment methods are used to evaluate the interface.

3.1 User interface adaptation

Adapting an existing user interface to a new version has been described extensively[16, 17, 31], but virtually all research that concretely describes the legacy and novel user interface aims at implementing a 'next generation' user interface, whereas our goal could be interpreted as moving back one generation, as we will remove any visual aspects from the existing interface. Therefore, we have opted to apply a more abstract method of adapting a user interface.

In *Issues in User Interface Migration*, Moore describes migration as *"the activity of moving software away from its original environment, including hardware platform, operating environment or implementation language to a new environment"*[38]. Although not all of these characteristics apply to our situation, the article proposes a method for systematic user interface migration that fits our purpose. The migration process is partitioned in three stages which are each described in subsections 3.1.1, 3.1.2 and 3.1.3.

3.1.1 Detection

The first stage of the migration process is the detection stage. The goal of this stage is to identify user interface functionality in the existing code through analysis. Moore lists several techniques that can be applied to perform the analysis:

1. Pattern matching of an abstract syntax tree that is created by parsing source code
2. Syntactic/semantic analysis of the source code against predefined keywords that identify user interaction
3. Manual detection of the source code

The article rightfully stresses issues that might arise when utilising technique 3, such as the likelihood of introducing errors and an insurmountable amount of time necessary to perform the detection. This certainly holds true in situations where a content specific user interface is going to be migrated, and (partially) automated techniques such as 1 and 2 will prevent these issues to a great extent. Our research question, however, is focused on adapting a content unaware user interface and thus analysing the user interface of a specific application developed with Apache Isis is pointless, as there is no reassurance that any functionality found in this application is relevant to other applications. This invalidates techniques 1 and 2 in our research scope.

Fortunately, due to the property of Apache Isis that the user interface is automatically generated through reflection, functionality in the user interface can be described in a more abstract form. This reduces the size of the user interface ontology to such an extent that manual detection can be deemed a feasible technique for the detection stage. Furthermore, all relevant features are well-documented[3] and thus it can be verified that the detection results are complete and correct.

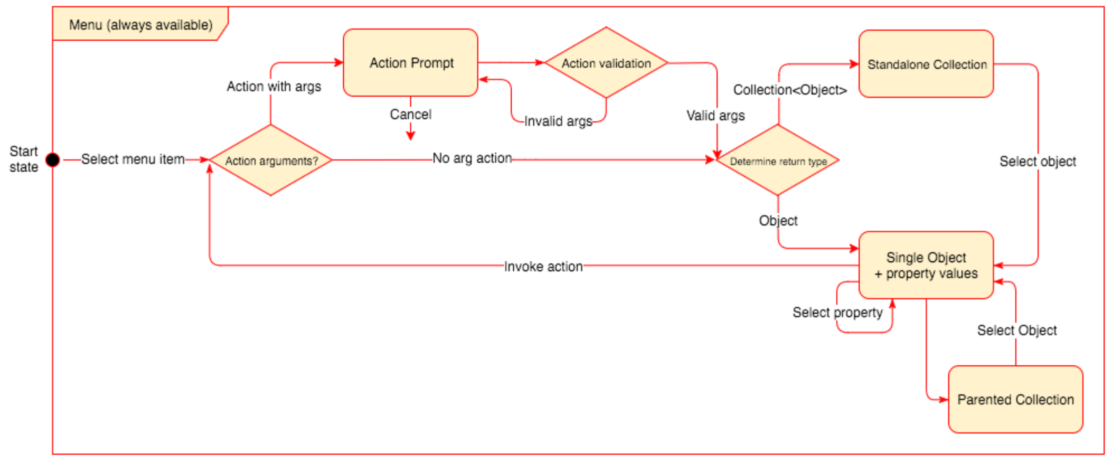


Figure 3.1: UML representation of the abstract user interface ontology

We performed manual detection on the framework which resulted in the following functionality to be exposed:

- **Menus**
  A user can access menus in the interface. Menus are available at any point in the interface.
- **Actions**
  A user can invoke menu and object actions.
- **Action prompts**
  If an action requires parameters, a user can enter the parameters in an action prompt.
- **Objects**
  A user can access objects and their properties.
- **Parented collections**
  A user can access collections that display objects related to a single object.
- **Standalone collections**
  A user can access collections that display objects, e.g. when an invoked action returns multiple objects.

These six elements together comprise the entire abstract user interface ontology. Figure 3.1 illustrates the relationships between the individual elements.

With the detection stage complete, we move on to the second stage.

3.1.2 Representation

The second stage of the migration process is the representation stage. The objective of this stage is to describe and document the functionality that we uncovered in the detection stage. It must be described in such a way that it is not dependent on the target platform or technology while still adequately representing all requirements of the user interface.

We have drafted a set of specifications which represent the functionality which our new interface must implement. This stage is particularly relevant to answering research question 2, as it allows us to judge whether or not the domain model is still intact; successfully implementing all specifications implies that the integrity of the domain model is maintained.

Specification S1	
Description	Menus can be accessed at all times
Rationale	Classes annotated with `@DomainService` are menus in the GUI. In the new interface menu selection will be the first user interaction and menus are available at any point in the interface.

Specification S2	
Description	Objects can be accessed
Rationale	Classes annotated with `@DomainObject` are objects in the GUI. Any object that is available in the GUI must be accessible in the new interface.

Specification S2-A	
Description	Object properties can be accessed
Rationale	Object variables annotated with `@Property` are properties of this object, such as names or dates. In the new interface, primitive properties must be visible and object properties must be accessible.

Specification S3	
Description	Collections can be accessed
Rationale	Variables annotated with `@Collection` are collections in the GUI. Collections should be displayed correctly depending on if they are parented or standalone (see figure 3.1) and all objects must be accessible.

Specification S4	
Description	Actions can be executed
Rationale	Functions annotated with `@Action` are operations on a certain entity in the application, such as executing a search query. Any action that is available in the GUI must be available in the new interface.

Specification S4-A	
Description	Action parameters can be accessed
Rationale	Actions may have parameters necessary to execute them. These parameters must be accessible.

Specification S5	
Description	A help menu can be accessed at all times
Rationale	Users must be able to get context-specific help at any point in the interface.

Specification S6	
Description	The application can be terminated
Rationale	The user must be able to log out and exit the application.

Specification S7	
Description	The application provides error handling
Rationale	The framework offers a lot of error feedback, such as invalidation messages after incorrect parameters. The new interface must provide a method of handling errors.

3.1.3 Transformation

The third and final step in the migration process is the transformation step. We found ourselves in a similar situation as in the representation step; Moore describes a number of (partially) automated techniques to perform the transformation, often relying on generating code based on mappings between a specific type of representation such as XML and the detection stage results. Again, this is tailored towards content-aware user interfaces, and thus we will simply use a manual transformation as our user ontology is concise enough to do so.

The actual transformation stage is described in detail in chapter 4.

3.2 Experiments

To answer research question 3 and subsequently research question 1, we will conduct a series of experiments to evaluate how user performance differs from the standard user interface when the new user interface is used. First, a theoretical approach is taken by applying the GOMS method, which we will describe in section 3.2.1. Some limitations apply to this method, however. To compensate for these limitations we will run time trials with a set of test subjects to obtain empirical results, as explained in section 3.2.2. The results of these experiments are covered in chapter 5.

3.2.1 GOMS

The *Goals, Operators, Methods and Selection rules* (GOMS) method is a well-established method to model efficiency-related design issues and is often applied in early stages of user interface design evaluation[47, 26, 29]. It is a qualitative method that aims to predict user execution time of a goal-oriented task. There are four individual components:

- The **goal** is what is expected of the user to accomplish in the task
- The **operators** are physical and cognitive processes that are required to complete the task
- The **methods** are series of operators that the user can execute to reach the goal
- When there are multiple methods to achieve the same goal, the **selection rules** will decide which method is picked

Physical operators are processes such as moving the mouse to a target destination or pressing a key, whereas processes like deciding between two options and remembering previous information are examples of cognitive operators. For more accurate results, it is recommended to determine the operator time coefficients empirically in a controlled environment which resembles the context of the GOMS analysis[24]. We are unable to do so within the time frame of this research and thus will apply the coefficients posed by the inventors of the method, which have been adopted more universally[30]:

- **K - keystroke**: .28 seconds for an average nonsecretarial typists
- **T_n - sequence of n characters**: n × K seconds
- **P - point with mouse to target**: 1.1 seconds
- **B - press mouse button**: .1 seconds
- **H - home hands to keyboard or mouse**: .4 seconds
- **M - mental act of routine thinking or perception**: 1.2 seconds

Research has shown that typing accuracy of visually impaired users is not significantly different from sighted users[25], and thus we will use the average speed for keystrokes.

The advantages of GOMS analysis are that it is a fast and cheap way of obtaining results. It does, however, have several limitations that must be kept in mind[47]. First, the model applies to expert users and execution times are thus based on users who are familiar with the system; novice users will usually perform worse than the projected time. Second, it does not account for errors which in reality will occur. Third, it can only apply to serial execution of tasks, excluding parallel tasks from the analysis. Finally, it does not take user fatigue in account which will increase during extended usage.

3.2.2 Time trial

To attain empirical results regarding user performance in the new interface, we will compose 5 test scenarios that will be executed by a small number of test subjects. The scenarios are small tasks related to Incode's Contact app[4]. All scenarios will be executed in both user interfaces, where we will avoid bias by providing them out of order and alternating between the user interfaces[15]. All user interaction will be captured to enable analysation of every step that is required to fulfil the tasks.

To log the user interaction in the standard interface, we will use an adapted form of Apache Isis' `PublisherServiceLogging` class. This is a built-in module to publish events such as action invocations or property changes to a logfile. A small adaptation was made to include logging of events such as the loading of objects or typing in a parameter field.

Since the new user interface will only have one input field that is permanently in focus, all user interaction will be confirmed with pressing the enter key. This allows us to print the input text with a timestamp, logging all user interaction.

For each step, we can then take the mean of time it took for each participant to get to the next step, up until completion of the task. By plotting a timeline for both user interfaces and their respective means, we will be able to visualise the performance difference between the user interfaces in a clear manner.

CHAPTER 4

Implementation

This chapter will provide insight on how the new user interface is implemented. First, a description of the frameworks and tools used is given in section 4.1. Then, section 4.2 will give a brief description of the core components that drive the new user interface.

4.1 Frameworks and tools

The new user interface will be browser-based and is built with several modern frameworks and tools. Due to the highly adoptable characteristic of speech feedback, we have opted to use the Web Speech API, a W3C specification that is currently in development. At the time of writing, Google's Chrome browser offers the best support for this feature, and thus our new user interface requires the use of Chrome. More details on the Web Speech API are provided in section 4.1.3.

4.1.1 REST API

One of Apache Isis' core features is the availability of a *representational state transfer* (REST) API. A REST API provides a simple way of opening up an application to third party software or implementations. Unlike SOAP which uses remote objects, action invocations and encapsulated functionality, REST only exposes data structures and their current state[13]. It utilises HTTP to transfer its data and represents its data in JSON format, making it highly compatible with other technologies.

Richard Pawson, who conceived the NO pattern, and Dan Haywood, lead committer of Apache Isis, have drafted the Restful Objects specification. This is an extension of the REST specification with a set of rules to describe a domain model implementing the NO pattern[7]. Apache Isis implements the Restful Objects specification.

The REST API allows us to easily connect our user interface to the existing back-end of any application. Once the user is authorised, only those services and objects that would be visible to the user in the standard user interface are exposed in the REST API. Furthermore, the standardised technologies used in the API enables the use of a wide range of web frameworks, and thus we can pick one that caters best towards achieving the effective and simple interaction part of research question 2.

4.1.2 AngularJS

One of the main areas of concern in our new user interface is that the target user group should not have to worry about anything that can impede the use of the application. Therefore, we have opted for AngularJS[1] as the framework to develop the new user interface in. AngularJS greatly simplifies the development of single-page applications, which is a valuable asset for our user interface. All user input is provided through one input field, and it is desirable that the user is unable to accidentally bring the input field out of focus. With a single-page application, the browser will never refresh, providing an uninterrupted user experience.

Furthermore, AngularJS has built-in support for consuming a REST API through its `$resource` factory, reducing the complexity of the code needed to interact with the back-end of the application.

AngularJS distinguishes three main components:

- The **scope** acts as the model in the framework, containing the data that is relevant to a certain state of the application.
- The **controller** processes user input and updates the scope with new values.
- The **view** is a dynamic representation of the scope data. If the controller updates the scope, the view will automatically represent the changes.

4.1.3 Web Speech API

The Web Speech API is an interorganisational effort to make the internet more accessible through recognition and synthesis of speech[12]. While it is currently still in development, Google Chrome offers full support of the specified functionality. After some initial experiments, we have concluded that the API's speech synthesis works remarkably well and is easy to use. Moreover, it works with JavaScript out-of-the-box and thus is easily integrable with the AngularJS application.

Aside from the speech synthesis functionality, the Web Speech API also offers speech recognition. Brief testing found that while the speech recognition functionality performed far better than expected, there were some issues to overcome when synthesis and recognition were combined which would be too time consuming to solve within the time frame of this research. If development continues after completion of this research it will definitely be implemented at some point, but for now we will adhere to keyboard input.

4.2 Functionality

This section will describe the AngularJS components that have been implemented. We will not cover the views, as they are simple HTML files that do not contain any significant functionality.

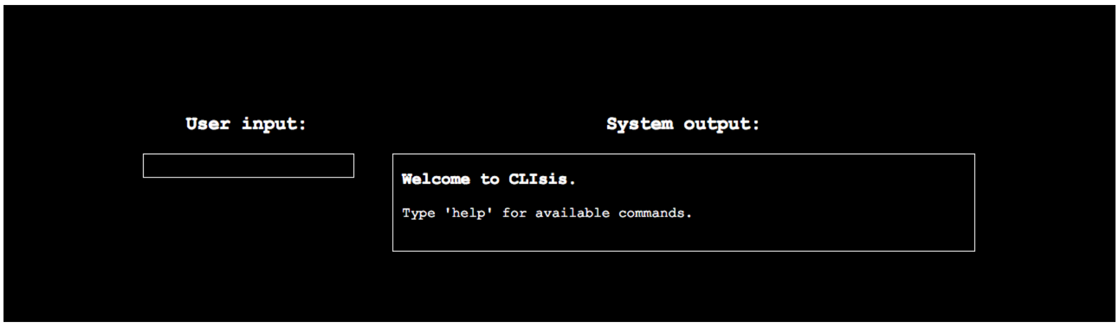
4.2.1 Main module

The main module **app.js** is the core of the application. Its main task is to implement the routing between the different states in the application. We have used the UI-router addon[10], as it provides a lot of additional functionality over the default routing provided by AngularJS's `$routeProvider`.

At any time, the application has two active views, as shown in figure 4.1: the user input view and system output view. This is achieved by adding an abstract state `base`. While an abstract state can not be active itself, child states inherit all features that are assigned to the abstract state. In our case, `base` only defines the parent view, which divides the screen in two separate views. The user input view never changes, while the system output view is updated conforming to user input. The following states are defined:

- `base.noOutput` displays a welcome screen.
- `base.home` displays the available menus.
- `base.services` displays the actions of a menu.
- `base.serviceAction` processes a menu action invocation. As this is an internal process, it has no view attached to it.
- `base.serviceActionParams` dispays parameters if a menu action requires them.
- `base.object` displays the properties, collections and actions of an object.
- `base.objectAction` processes an object action invocation. As this is an internal process, it has no view attached to it.
- `base.objectActionParams` displays parameters if an object action requires them.
- `base.collection` displays a collection.

- `base.login` displays the login screen.
- `base.error` displays the error message when an error is thrown.
- `base.help` displays the current context and available commands in that context.



```
Welcome to CLIsis.

Type 'help' for available commands.
```

Figure 4.1: The start screen of CLIsis

Furthermore, AngularJS supports filtering variables in views. There are three filters defined in the main module:

- `substringAfterChar` is used to capture the last part of a string after a certain character.
- `splitToLowerCase` takes a string, splits it on capitalised characters and then joins it with spaces in lower case. This is used so that the speech synthesiser correctly pronounces object keys, which are represented in camel case.
- `startFrom` calculates an index for pagination in displaying a collection.

4.2.2 Services

Services are processes that are not visible to the user, but serve as a method of sharing code across the application. In most cases their task is related to communication between the front- and back-end of the application.

- **authentication.js** is used to control the authentication process when logging in. The majority of the code was taken from Incode's Contact app[4], with slight alterations where necessary.
- **services.js** offers a function `getServices()` to retrieve the available menus from the REST API.
- **actions.js** contains several functions: `getActions()` retrieves all actions for an object or menu, `invokeAction()` invokes a specific menu action, `invokeObjectAction()` invokes a specific object action and `getActionParams()` is a helper function to determine whether the state should be routed to the parameter state or the invocation state.
- **objects.js** has a function `getObject()` to retrieve a specific object, `getCollection()` to get a collection of objects, and two helper functions `getObjectType()` and `getObjectId()`, which are used to parse an URL retrieved from the REST API to determine the object type and id. The function `buildObjectHref` is used to build the URL that is used in the AngularJS application.
- **speechService.js** is the service that reads out the user input and system output. It has a method `speak()` and `cancelSpeech()`, which are self-explanatory.
- Then there are two convenience services: **errorService.js** takes care of throwing errors, and **rootScopeSanitiser.js** makes sure that the `$rootScope`[1] is not polluted with stale variables.

[1]The `$rootScope` is a global scope accessible by all active controllers. We use it to transfer certain data between controllers.

4.2.3 Controllers

Each state of the application has its own controller. The controller ensures that the corresponding view is updated with new information. Aside from their individual behaviour described below, they all make use of AngularJS's `$scope.$watch` functionality, which 'watches' the data in the `$scope` for changes. When a change occurs, it triggers the `speak()` function so the user gets spoken feedback.

- **InputController.js** is the core controller of our user interface. It is always active and processes the user input that is provided through the input field. Aside from some helper functions that ensure that the focus of the browser is always on the input field and the user can not accidentally tab to the URL bar, the majority of its functionality resides in the function `evaluateInput()`. This function retrieves the input from the input field, splits it on spaces and then feeds the first element to a switch. The switch recognises 13 different commands. On commands that require one or more parameters, the switch also validates whether they are present and correct.
  - `menus` directs the application to the `base.home` state.
  - `menu` can either take the name of a menu or the index it is displayed with as the parameter. It then directs to the `base.services` state.
  - `actions` broadcasts a `$showActions` event. If a menu or object is currently in scope, it shows its actions and they are spoken out.
  - `action` can take either the name of an action or the index it is displayed with as the parameter. It then determines whether the action to be invoked has parameters or not; if it does, it directs to the `base.objectActionParams` or `base.serviceActionParams` state. If it does not have parameters, it directs to the `base.objectAction` or `base.serviceAction` state.
  - `field` takes an integer as a parameter to denote which parameter field is to be filled out with the content of the second parameter.
  - `submit` confirms that the user has filled out all parameter fields and invokes the action.
  - `get` can be used to get an object's property or collection, or an object from a collection. If the parameter is an integer, it directs the application to the desired state. If `get` is used on a collection, the parameter can also be a string. The controller then filters the contents of the collection with the input, and directs to the `base.object` state if there is only one result, or to a new `base.collection` if there are multiple results that fit the parameter.
  - `show` without a parameter displays the first five results of a collection. The parameters `next` and `previous` turn the pages, with wraparound on the first and last pages.
  - `properties` broadcasts a `$showProperties` event. If an object is currently in scope, it shows its actions and they are spoken out.
  - `collections` broadcasts a `$showCollections` event. If an object is currently in scope, it shows its collections and their sizes, and they are spoken out.
  - `back` goes back to the previous state. The controller keeps track of a `previousStates` stack, popping the top state when it goes back.
  - `help` lists the help menu.
  - `quit` logs the user out of the application and redirects to the login screen.
- **HomeController.js** gets the menus from the services service and then filters the results based on if there are actions available on each menu. If there are no actions available it is a system menu and bears no functionality for the user.
- **ServiceController.js** gets the menu actions from the actions service.

- **ObjectController.js** gets the object details from the objects service and then parses the results. For all collections that belong to the object, it gets the contents of the collection to display its size in the object view. It also provides a helper method `typeOf()` that is used in the view.

- **ActionParamController.js** processes the parameters that are required to invoke actions. Parameters can either be simple fields that require a form of text input, or they can be options which are predefined by the system. If the latter is the case, the REST API exposes them in an extra object key `choices`. The controller builds the HTML text with these choices. It also listens to a `$fieldInputEvent`, which contains the user input to update a parameter field.

- **ActionController.js** calls either `invokeAction()` or `invokeObjectAction()`, depending on if the action belongs to a menu or an object. If the action returns one object, the controller routes the application to the object state; if it returns multiple objects, it routes the application to the collection state.

- **CollectionsController.js** gets the collection of objects and manages the pagination in the view. Pagination is essential, as collections can grow large which will cause user frustration if the speech synthesiser speaks the content of a large collection in one go.

- **HelpController.js** sets boolean values based on the state in which the help command was issued, which the view uses to only those commands that are relevant to that state.

- **ErrorController.js** takes care of displaying the correct error message and **LoginController.js** handles the login process.

CHAPTER 5

Evaluation

This chapter describes the experiment results and answers the research questions.

5.1 Specification requirements

In this section, I will motivate which specifications have been met in the implementation. With this information I will be able to answer the first subquestion.

5.2 Experiments

5.2.1 GOMS

Here, the results from applying GOMS will be evaluated.

5.2.2 Time trial

Here, the results from the time trial experiment will be evaluated.

5.2.3 Performance difference

This subsection will evaluate the experiment results and answer the second subquestion.

CHAPTER 6

Conclusion

Here, I will answer the research question and make any other conclusions.

6.1 Future work

Any remaining future work can be discussed here.

6.2 Acknowledgements

Bibliography

[1] Angularjs. https://angularjs.org/. (Accessed on 06/02/2016).

[2] Apache isis. http://isis.apache.org/. (Accessed on 05/16/2016).

[3] Apache isis documentation. http://isis.apache.org/documentation.html. (Accessed on 05/31/2016).

[4] Contact app. https://github.com/incodehq/contactapp. (Accessed on 06/02/2016).

[5] Estatio. http://www.estatio.org/. (Accessed on 05/18/2016).

[6] Fulfilling the promise of mvc. https://www.infoq.com/articles/Nacked-MVC. (Accessed on 05/17/2016).

[7] Restful objects. http://www.restfulobjects.org/. (Accessed on 06/02/2016).

[8] Section 508 of the rehabilitation act. http://www.section508.gov/section-508-of-the-rehabilitation-act. (Accessed on 05/16/2016).

[9] Spring. https://spring.io/. (Accessed on 05/16/2016).

[10] Ui-router: The de-facto solution to flexible routing with nested views in angularjs. https://github.com/angular-ui/ui-router. (Accessed on 06/02/2016).

[11] Web content accessibility guidelines (wcag) 2.0. https://www.w3.org/TR/WCAG20/. (Accessed on 05/16/2016).

[12] Web speech api specification. https://dvcs.w3.org/hg/speech-api/raw-file/tip/speechapi.html. (Accessed on 06/02/2016).

[13] Battle, R. and Benson, E. (2008). Bridging the semantic web and web 2.0 with representational state transfer (rest). *Web Semantics: Science, Services and Agents on the World Wide Web*, 6(1):61–69.

[14] Boyd, L. H. et al. (1990). The graphical user interface crisis: Danger and opportunity.

[15] Chen, J.-W. and Zhang, J. (2007). Comparing text-based and graphic user interfaces for novice and expert users. In *AMIA Annual Symposium Proceedings*, volume 2007, page 125. American Medical Informatics Association.

[16] Classen, I., Hennig, K., Mohr, I., and Schulz, M. (1997). Cui to gui migration: Static analysis of character-based panels. In *Software Maintenance and Reengineering, 1997. EUROMICRO 97., First Euromicro Conference on*, pages 144–149. IEEE.

[17] Csaba, L. (1997). Experience with user interface reengineering: Transferring dos panels to windows. In *Software Maintenance and Reengineering, 1997. EUROMICRO 97., First Euromicro Conference on*, pages 150–155. IEEE.

[18] Edwards, W. K., Mynatt, E. D., and Stockton, K. (1994). Providing access to graphical user interfacesnot graphical screens. In *Proceedings of the first annual ACM conference on Assistive technologies*, pages 47–54. ACM.

[19] Emiliani, P. L. and Stephanidis, C. (2000). From adaptations to user interfaces for all. In *6th ERCIM workshop CNR-IROE, Florence, Italy.*

[20] Evans, E. (2004). *Domain-driven design: tackling complexity in the heart of software.* Addison-Wesley Professional.

[21] Fisher, D. L., Yungkurth, E. J., and Moss, S. M. (1990). Optimal menu hierarchy design: syntax and semantics. *Human Factors: The Journal of the Human Factors and Ergonomics Society*, 32(6):665–683.

[22] Frauenberger, C., Putz, V., and Holdrich, R. (2004). Spatial auditory displays-a study on the use of virtual audio environments as interfaces for users with visual disabilities. *DAFx04 Proceedings*, pages 5–8.

[23] Galitz, W. O. (2007). *The essential guide to user interface design: an introduction to GUI design principles and techniques.* John Wiley & Sons.

[24] Gong, R. and Kieras, D. (1994). A validation of the goms model methodology in the development of a specialized, commercial software application. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 351–357. ACM.

[25] Ishida, H. (1993). Accuracy and error patterns in typing of the visually handicapped. *Human Engineering*, 29(5):321–327.

[26] John, B. E. and Kieras, D. E. (1996). The goms family of user interface analysis techniques: Comparison and contrast. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 3(4):320–351.

[27] Kane, S. K., Bigham, J. P., and Wobbrock, J. O. (2008). Slide rule: making mobile touch screens accessible to blind people using multi-touch interaction techniques. In *Proceedings of the 10th international ACM SIGACCESS conference on Computers and accessibility*, pages 73–80. ACM.

[28] Kaye, J. N. (2001). *Symbolic olfactory display.* PhD thesis, Citeseer.

[29] Kieras, D. (1994). Goms modeling of user interfaces using ngomsl. In *Conference companion on Human factors in computing systems*, pages 371–372. ACM.

[30] Kieras, D. (2001). Using the keystroke-level model to estimate execution times. *University of Michigan.*

[31] Kong, L. (2000). *Legacy interface migration: From generic ascii UIs to task-centered GUIs.* PhD thesis, University of Alberta.

[32] Kortum, P. (2008). *HCI beyond the GUI: Design for haptic, speech, olfactory, and other nontraditional interfaces.* Morgan Kaufmann.

[33] Landauer, T. K. and Nachbar, D. (1985). Selection from alphabetic and numeric menu trees using a touch screen: breadth, depth, and width. *ACM SIGCHI Bulletin*, 16(4):73–78.

[34] Lazar, J., Allen, A., Kleinman, J., and Malarkey, C. (2007). What frustrates screen reader users on the web: A study of 100 blind users. *International Journal of human-computer interaction*, 22(3):247–269.

[35] Leff, A. and Rayfield, J. T. (2001). Web-application development using the model/view/controller design pattern. In *Enterprise Distributed Object Computing Conference, 2001. EDOC'01. Proceedings. Fifth IEEE International*, pages 118–127. IEEE.

[36] Leuthold, S., Bargas-Avila, J. A., and Opwis, K. (2008). Beyond web content accessibility guidelines: Design of enhanced text user interfaces for blind internet users. *International Journal of Human-Computer Studies*, 66(4):257–270.

[37] Miller, G. A. (1956). The magical number seven, plus or minus two: some limits on our capacity for processing information. *Psychological review*, 63(2):81.

[38] Moore, M. M. and Rugaber, S. (1993). Issues in user interface migration. In *Proceedings of the Third Software Engineering Research Forum*.

[39] Myers, B. A. (1998). A brief history of human-computer interaction technology. *interactions*, 5(2):44–54.

[40] Mynatt, E. D. (1995). Transforming graphical interfaces into auditory interfaces. In *Conference companion on Human factors in computing systems*, pages 67–68. ACM.

[41] Nielsen, J. (1994). *Usability engineering*. Elsevier.

[42] Paap, K. R. and Roske-Hofstrand, R. J. (1986). The optimal number of menu options per panel. *Human Factors: The Journal of the Human Factors and Ergonomics Society*, 28(4):377–385.

[43] Pawson, R. and Matthews, R. (2002). Naked objects. In *Companion of the 17th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 36–37. ACM.

[44] Poll, L. H. D. (1996). *Visualising Graphical User Interfaces for Blindusers*. Eindhoven University.

[45] Reenskaug, T. (2007). Programming with roles and classes: The babyuml approach. *A chapter in Computer Software Engineering Research*.

[46] Savidis, A. and Stephanidis, C. (2004). Unified user interface design: designing universally accessible interactions. *Interacting with computers*, 16(2):243–270.

[47] Schrepp, M. (1990). Goms analysis as a tool to investigate the usability of web units for disabled users. *Universal Access in the Information Society*, 9(1):77–86.

[48] Yanagida, Y., Kawato, S., Noma, H., Tomono, A., and Tesutani, N. (2004). Projection based olfactory display with nose tracking. In *Virtual Reality, 2004. Proceedings. IEEE*, pages 43–50. IEEE.

[49] Yatani, K. and Truong, K. N. (2009). Semfeel: a user interface with semantic tactile feedback for mobile touch-screen devices. In *Proceedings of the 22nd annual ACM symposium on User interface software and technology*, pages 111–120. ACM.

Acronyms

DDD domain-driven design. 10, 11

GOMS Goals, Operators, Methods and Selection rules. 18

GUI graphical user interface. 5, 6, 9, 11, 12, 17

MVC model-view-controller. 9–11

NO Naked Objects. 10, 11, 21

REST representational state transfer. 11, 21, 23

APPENDIX A

Application manual

A.1 Installation

This appendix provides instructions on running the demo application.

A.2 User manual

A concise manual to use the interface.