ARCHITECTING AND IMPLEMENTING DYNAMIC, CROSS-PLATFORM USER INTERFACE LIBRARY

Juhani Alanko

Bachelor's Thesis
May 2013

Software Engineering
School of Technology

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

Author(s)
ALANKO, Juhani

Type of publication
Bachelor's Thesis

Date
7.5.2013

Pages
32

Language
English

Permission for web publication
(X)

Title
ARCHITECTING AND IMPLEMENTING DYNAMIC, CROSS-PLATFORM USER INTERFACE LIBRARY

Degree Programme
Software Engineering

Tutor(s)
PIETIKÄINEN, Kalevi

Assigned by
Star Arcade Oy

Abstract

The thesis focuses on designing and implementing a graphical user interface library. It was an assignment for Star Arcade Ltd., a Jyväskylä-based game company. The purpose of the library was to provide the tools for both game programmers and graphical designers for creating user interfaces in Star Arcade products.

The thesis elaborates on how a dynamic and cross-platform user interface library has been developed in terms of particular usability and technology requirements. The thesis also discusses the problems that arose in both design and implementation phases.

The library was developed to replace the old user interface library, which had become obsolete in terms of usability and meeting the needs of the other developers. Thus the requirements for a new library were moderately well defined, although they changed a few times during the development process.

The user interfaces were described using XML for the application code which was written in C++. The final library was generic and extensive, although defined within set constraints. The interfaces created according to the specification described in the thesis are cross-platform and automatically work in the way the designer wanted without having to consider multiple interface definitions for different target platforms.

The library was adopted in all Star Arcade in-development products and at the moment of writing the thesis it is a part of Star Arcade's third-party SDK.

Keywords
user interface, cross-platform, data-driven programming, game programming, XML, C++, MVC

Miscellaneous

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

OPINNÄYTETYÖN
KUVAILULEHTI

Tekijä(t)	Julkaisun laji	Päivämäärä
ALANKO, Juhani	Opinnäytetyö	7.5.2013
	Sivumäärä 32	Julkaisun kieli Englanti
		Verkkojulkaisulupa myönnetty (X)

Työn nimi
ARCHITECTING AND IMPLEMENTING DYNAMIC, CROSS-PLATFORM USER INTERFACE LIBRARY

Koulutusohjelma
Ohjelmistotekniikka

Työn ohjaaja(t)
PIETIKÄINEN, Kalevi

Toimeksiantaja(t)
Star Arcade Oy

Tiivistelmä

Opinnäytetyö käsittelee graafisen käyttöliittymäkirjaston suunnittelua ja toteutusta. Työ toteutettiin toimeksiantona Star Arcade Oy:lle, joka on jyväskyläläinen peliyhtiö. Kirjaston tarkoituksena oli tarjota työkalut käyttöliittymien toteuttamiseen Star Arcaden tuotteissa sekä pelikoodareille että graafisille suunnittelijoille.

Opinnäytetyön tarkoitus on kertoa, miten dynaaminen ja alustariippumaton käyttöliittymäkirjasto on tässä tapauksessa ja tiettyjen vaatimusten ohjaamana toteutettu. Sen tarkoituksena on myös perustella kehitystyön aikana tehtyjä valintoja sekä selventää, miksi ja miten lopputulokseen on päädytty. Työssä käsitellään lisäksi esiin nousseita ongelmakohtia ja niiden ratkaisuja.

Motivaationa käyttöliittymäkirjaston kehittämiselle oli korvata aiemmin käytössä ollut, tarpeiden muuttuessa ominaisuuksiltaan ja käytettävyydeltään vanhentunut UI-kirjasto. Näin ollen vaatimukset uudelle kirjastolle olivat kohtuullisen hyvin tiedossa, mutta kehitystyön edistyessä tilanteet muuttuivat usein, ja samalla vaatimuksia jouduttiin määrittelemään uudestaan.

Opinnäytetyö toteutettiin käyttäen XML-kuvauskieltä käyttöliittymien kuvaukseen sekä C++-ohjelmointikieltä käyttöliittymäkuvauksien integrointiin Star Arcaden tuotteisiin. Tuloksena oli geneerinen ja kattava, mutta tarpeisiin rajattu käyttöliittymäkirjasto, jolla voi toteuttaa dynaamisia, suorituskykyisiä ja visuaalisesti näyttäviä käyttöliittymiä. Käyttöliittymät ovat alustariippumattomia ja toimivat suunnittelijan haluamalla tavalla laitteesta ja näytön resoluutiosta riippumatta.

Käyttöliittymäkirjasto otettiin käyttöön kaikissa Star Arcaden kehitystyön alla olevissa tuotteissa, ja se on opinnäytetyön kirjoitushetkellä osa kolmansille osapuolille avointa Star SDK-kehitystyökalupakettia.

Avainsanat (asiasanat)
käyttöliittymä, alustariippumattomuus, tieto-ohjattu ohjelmointi, peliohjelmointi, XML, C++, MVC

Muut tiedot

Contents

List of Figures

Glossary

client layout, client UI

Any UI layout that is provided by the Star Arcade Client. These are provided by the system and the individual games do not need to describe them.

cross-platform

Developing software for, or running software on, more than one type of hardware platform (PCmag).

data-driven programming

A programming paradigm, in which the program statements describe the input data to match and process rather than a sequence of program steps (Stutz 2006).

Diesel Engine

The multiplatform game engine used internally in Star Arcade.

game client, Star Arcade Client, Star Lounge

A layer of code in Star Arcade technology that handles the network interaction and provides graphical UIs and functionality for common tasks in Star Arcade games.

game layout, game UI

Any UI layout that is provided by the game. The Star Arcade Client does not contain knowledge about these, but they are loaded into its UI manager.

layout

The graphical representation of user interface elements that are related to each other and displayed together, in this project's scope the elements of a page or a view.

user interface

A layer of interaction between human and computer. In the case of this thesis, it is assumed that the user interface is graphical.

widget

An object that represents a graphical element. May contain properties such as size, position, text and image.

Acronyms

API Application Programming Interface.

MVC Model-view-controller software pattern, in which the data (model) is separated from its representation (view) and the user interaction is handled via a third, separate module (controller).

UI User interface.

1 Background

1.1 Star Arcade as a project environment

Star Arcade is a small to medium-sized game company located in the centre of Jyväskylä. It employed around 30 people at the time of working on this thesis. The company had a few game projects going, each of which utilised an in-house game engine. The logo of the company is seen in Figure 1.

Figure 1: Star Arcade company logo

Structurally, Star Arcade's departments are divided roughly into programmers, graphic artists and marketing people. Some programmers work on game titles, while others are focused on the "core technology", the building blocks for the games. The core technology people also developed a third-party game SDK, which at the time of writing of this thesis was about to be released to other developers globally.

As Star Arcade focuses on mobile social games, the projects are on the smaller side compared to big game projects such as The Elder Scrolls or Call of Duty. Typically, a game project employes one programmer and artist, and all projects are steered by the company managers.

The thesis was an assignment in the core technology department, and its goal was to produce a user interface library to be integrated in the game client from which it could be utilised in game projects.

1.2 Out with the old library, in with the new

The main reason for the UI library was the old UI library which had served for a time, but lacked in usability and had partially become obsolete in terms of new features. The old library was developed by a person who was no longer working for the company, and most people found the library as too hard to use effectively. Thus the new UI project was started and the author of this thesis began designing and implementing a new, slightly less intimidating UI library.

The scope and size of the project was known even in the beginning of the project, but as the project went on, it grew larger than anticipated and took considerably more time than it would have according to the initial plan. This was mainly because of rapidly changing requirements and also other reasons that are outside of the scope of this thesis.

Structurally, the UI library resides at the core of the Star Arcade technology. In Figure 2 the UI code is located in the Star Social Lounge layer (Star Arcade promotion material).

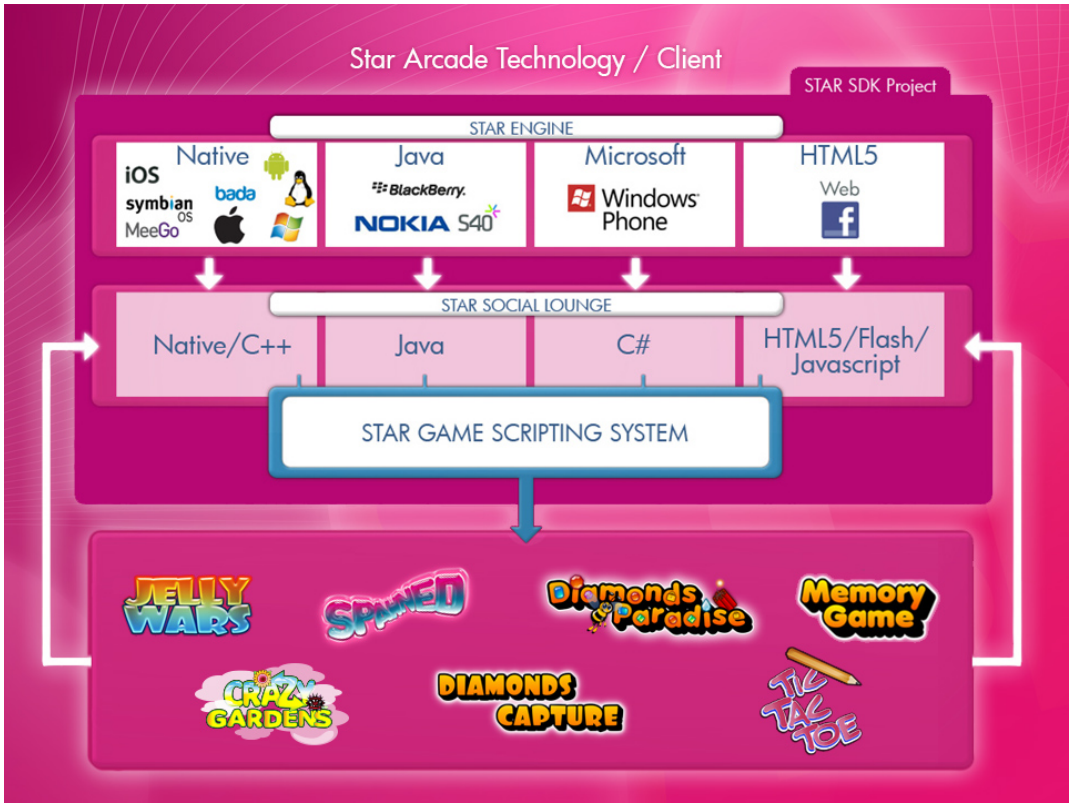

Figure 2: Star Arcade Technology

1.3 Time and human resources

In the beginning of the project, October 2012, it was scheduled to be complete in roughly a month or a bit more, and the main motive was to build the system without worrying too much about time. However, the project scope grew during the development and also due to the author's inexperience in architecting complex systems the project took more time than anticipated.

Nearing December, there started to be pressure to finalise the library before the end of the year. After January 2013, it was decided that since the UI library was bound to Star Arcade products as a whole, the upcoming games would only be released when all aspects of the game were complete. This also meant that the UI library would be worked on until everyone was satisfied with it.

As the project went on, multiple people took part in it. The main work was done by the author of this thesis, but some additional code (UI manager, for instance) was done by the Star Arcade lead coder, Arto Katajasalo. Many design-related problems were also tackled with Arto and Mika Kytöjoki, the author of Star Arcade's in-house design tool called Positioner. From the design side, the library was used first by Jari Saarinen, who designed UI layouts using Positioner that were exported to game assets. Jari supplied valuable feedback and bug reports from the user's point of view.

This thesis focuses on the author's part in the development process, and other people's work is clearly credited and the authors mentioned where needed. At the moment of writing this thesis, the UI system is in its final testing phase and it has been contributed by many people. This thesis focuses on the earlier development process and shaping of the library.

2 Designing a reusable UI library

2.1 What is user interface?

A user interface library is a system that is going to provide basic building blocks for almost everything that is interactable on the screen. Thus, it has to be as reusable and generic as possible, so that many different layouts can be produced with it.

That leads to a basic princible in designing a UI system: it has to rely on current specifications and requirements, but it cannot rely on current layout functionality. This means that it has to provide ways to function in any kind of layout, not just the current ones. For example, if the current layout design has a button that expands upon clicking it, and shrinks upon releasing the mouse button, the UI library has to take into consideration that in other layouts the animation might not be desired at all. So the library should provide different ways of animating the button, from which the library user can choose, instead of hard-coding any one idea into the implementation.

A well designed and planned out system is tremendously easier to implement as the big picture has already been formed and broken into manageable features. The different parts of a UI system are illustrated in Figure 3.

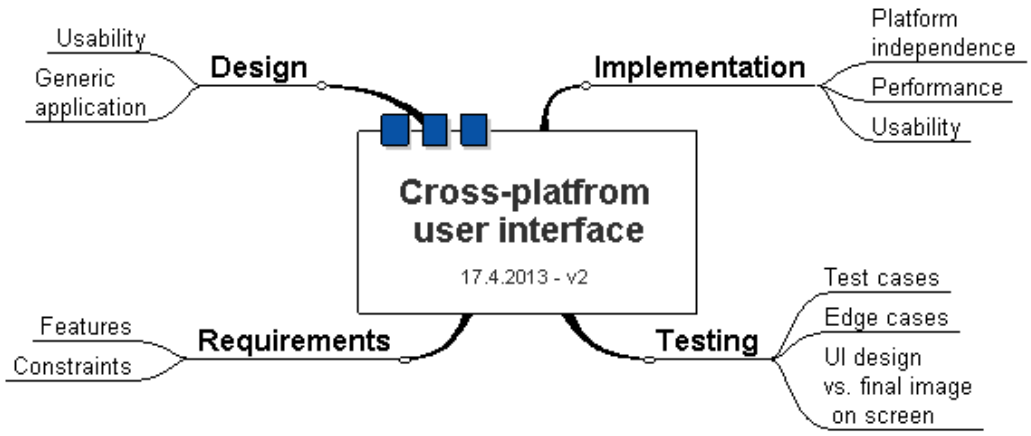

Figure 3: The design of a UI system

2.2 Who uses the system?

The UI system consists of the design part, which produces a UI description in XML format. This is done by an artist in Star Arcade's case, utilising an in-house design tool that exports the design into XML. However, more generally, this is the start of the UI pipeline, in which the graphical idea is formed. The designer is asking questions such as "What does the UI look like?".

The second part of the pipeline is importing the XML description into game code. This is where the UI library is actually used and the XML description is parsed into Widget nodes forming a tree. The Star Arcade client layouts are automatically loaded upon starting the game, and the game code handles loading of game layouts. Constructing the layouts into user interfaces is the game programmer's territory, and she or he decides how the UIs work. They ask questions such as "How does one interact with the UI?".

The third part of the pipeline is presenting the UI to the actual player who is enjoying the game. It is very important that the experience is as fluid and intuitive as possible. This means that even if the designer has succeeded in creating a visually impressive layout, and the game programmer has done a good job with animating and presenting it, the UI system itself still has to provide good performance to ensure that there are no delays, freezing or any weird functionality happening on screen. The interface user should probably not be asking questions such as "How do I use this UI?", as the UI is ideally so intuitive that the user will not notice it at all.

The relations of the users to the system is illustrated in Figure 4.

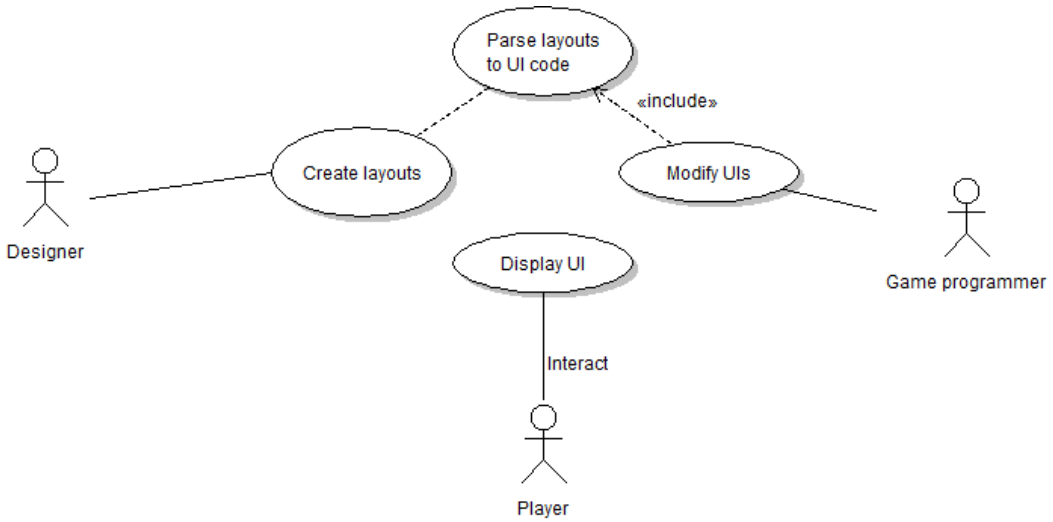

Figure 4: The users of the UI system

2.3 What is generic enough?

When designing a system that is generic, one inevitably faces a point where the system is in danger to become over-generic. This is mostly prominent when the project has tight time constraints. The problem is that a completely generic system is an idea, something that can not be constructed in the real world. The architect of the system has to decide when to stop making the system more generic to be able to construct a functional system at all.

During the development of the UI system this had to be kept in mind when deciding about different features. For instance, the Container widget (which handles list functionality) had to be simplified a couple of times to keep it functioning as a simple container and not an entire UI subsystem of its own.

2.4 Meeting the requirements

The UI system was built on requirements that defined what was expected from the system and how it had to function as a part of a bigger system. These requirements had partially formed during the reign of the last UI system, and some other requirements were invented when people were designing new Star client. As an opposing force to the generic behaviour and re-usability, the needs from the Star client often dictated how exactly the UI system had to behave and this resulted in

many situations where a decision had to be made between keeping a feature generic and making a "special case" of it.

Special cases are rules that are hard-written into the code and serve no logical purpose if the reader is not familiar with the requirement set for the system. Most often they are a sign of a system that is in need of a layer of abstraction or some new API functionality, but sometimes special cases have to be made to get things done without investing many hours to build new complex systems that may be difficult to use. In most cases the UI library was developed without special cases, and those that were implemented, were clearly marked and commented to fix them later.

2.5 Thinking from a third-person point of view

2.5.1 Star SDK

The UI library was developed to be a part of a software development kit (SDK) which was to be released to third parties later on. This caused a lot of additional planning and thinking, as you could not only restrict the use cases to those that were relevant in Star Arcade. This also proved to be beneficial, as every development decision had to be thought out from all points of view, you could not just rush the coding.

Preparing a system for all possible use cases is also something that is impossible to implement, so there have to be limits that are recognised during the development process. These constraints form the "sandbox" for any developer using the library, and when defined appropriately, they ensure that the library is used correctly and in the way the developer wanted it to be used.

When designing any system, the developer should make interfaces easy to use correctly and hard to use incorrectly (Meyers 2005, 78). Following this simple rule made it easier to shape the API of the UI library. The constraints should not feel like a limiting factor, and thus the API had to be as logical and easy to use as possible, so that the unknown developer would follow the same reasoning while using the system that the developer of the system itself when architecting it.

2.5.2 In-house usage

Using the system in Star Arcade provided valuable feedback and helped finding and correcting mistakes in the library, but the use cases and the purpose of the library was limited to Star Arcade's projects and as such it was difficult to predict how the system was going to be used outside Star Arcade. However the SDK release was a secondary purpose, and the primary raison d'être of the library was to be an in-house tool for Star Arcade projects. So the features of the library originated from the needs of the Star Arcade developers, and were implemented so that any third-party developer could use them.

Sometimes it is hard to think outside the box, and especially so when other people to get their work done as quickly as possible. There was no place for quick and dirty solutions, and sometimes it took some effort to convince other people that a certain feature could not work the way they wanted to, because it had to be more generic to be used outside the company. On the other hand, this also helped the developer of the UI library to identify unneeded features, or features that had to be re-designed from a completely different point of view.

Especially designing the Container widget proved to be a difficult task, because it had to be kept simple to use, but often it had to perform complicated arranging of child widgets, to be scrollable to any direction, dynamically add and remove child widgets and keep other widgets' relative positions in tact. A short look to the container implementation is illustrated in Figure 5. In short, it took a lot of effort to keep the Container from becoming a completely new subsystem within the UI system, and this required co-operation from the layout designers to convince them that sometimes their wishes could not be fulfilled if the system was to be kept sane enough for other people to use.

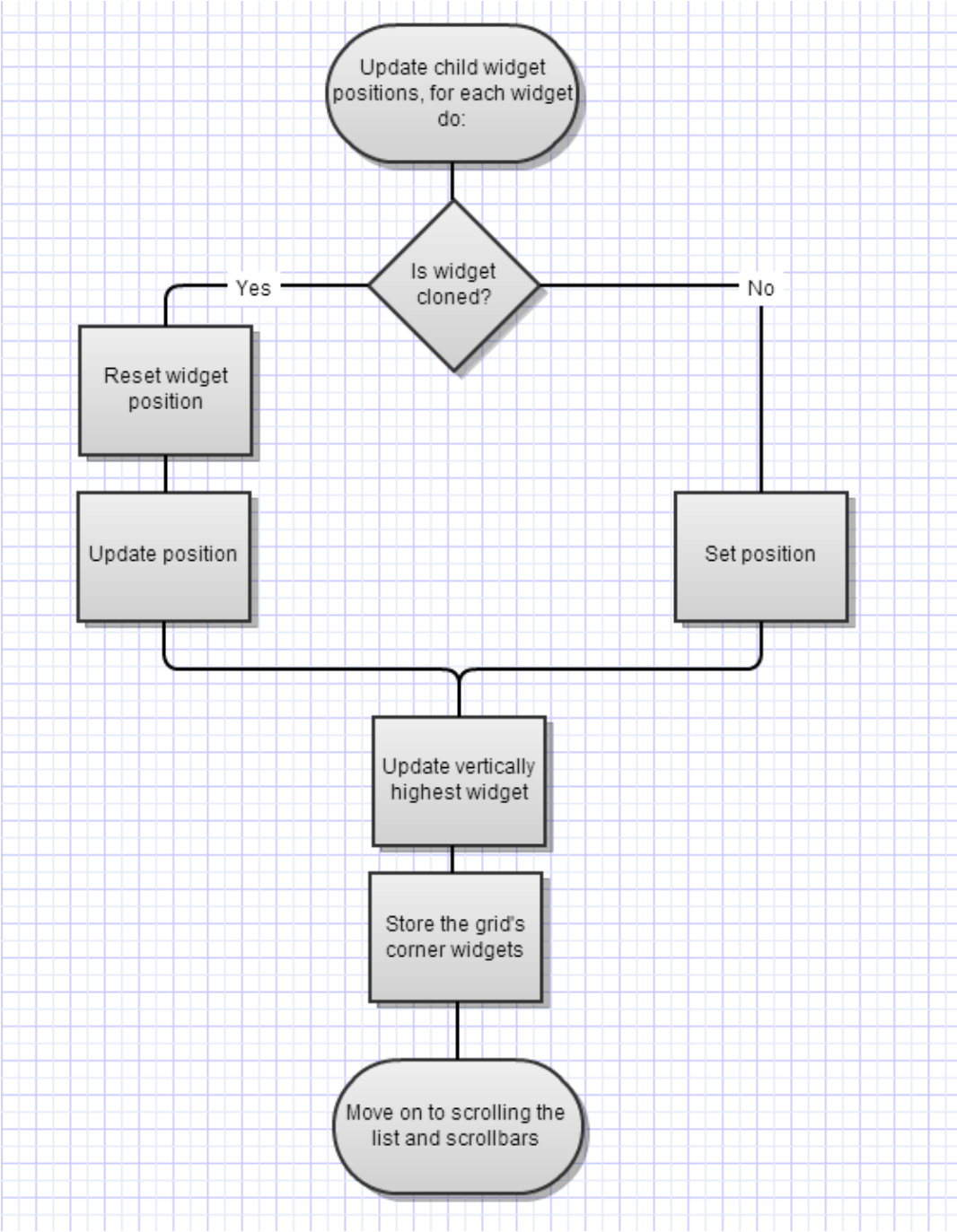

Figure 5: Widget updating inside Container

3 Implementing a dynamic UI library

3.1 System features

The implementation process took place in parallel with the design process in a way that the features were planned and prototyped one by one, and already implemented features were refactored or fixed when the requirements changed or bugs were found. Initially the API was not yet in use in Star Arcade, so breaking changes were allowed. This enabled quick prototyping without the need to keep the header files unchanged in the future.

The library had to work on multiple platforms without taxing the system memory or CPU too much, as the UI had to give room for the actual game to run smoothly. This had some implications in layout construction and drawing that had to be taken into consideration, such as culling the drawing of inactive views and adding some additional data properties to widgets to remove the need to re-parse XML files after the initial loading.

Finally, the system had to be re-usable and modular in a way so that parts of it could be extended or changed without the rest of the system needing any changes. This is achieved by using the MVC pattern in the code to separate the data from the controls and representation.

Main features of the system are illustrated in Figure 6.

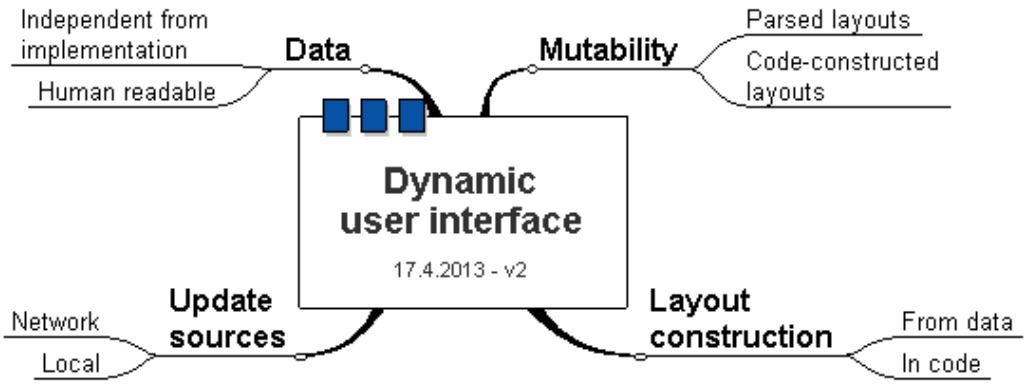

Figure 6: Features of the library

3.2 Describing a UI...

3.2.1 ...in the pipeline

The layouts are represented differently in various parts of the UI pipeline. As multiple people work on the layouts with different tools, the data has to be transferred from one tool to another and each tool handles the data in an optimal way for the tool, containing the relevant information about the layout.

At first, the layout designer has an idea in her head, which she transforms into an UI description by using the in-house Positioner tool. This process creates the rules that the game code uses to display the elements with correct graphics and texts in correct places. It can be seen as drawing the image of the layout, as it contains no functional data. The Positioner tool is presented in Figure 7.

Figure 7: The in-house Positioner tool

From the design tool, the layout data is exported to XML. This representation is called the UI description. It is a tree of elements that contain properties, of which some are common to all elements and others unique to the element type in question.

The XML description can be changed with a text editor, but as a rule it is not tampered with in Star Arcade UI pipeline to prevent arduous and error-prone manual labour and to keep the Positioner descriptions synchronised with the XML data.

The XML file is then given to the program code, which parses it during the start-up initialisation of the game to create a tree of widget objects into the device memory. Those widgets represent the elements described in the XML, and contain similar properties that are initialised according to XML data.

The data flow in the UI pipeline is illustrated in Figure 8.

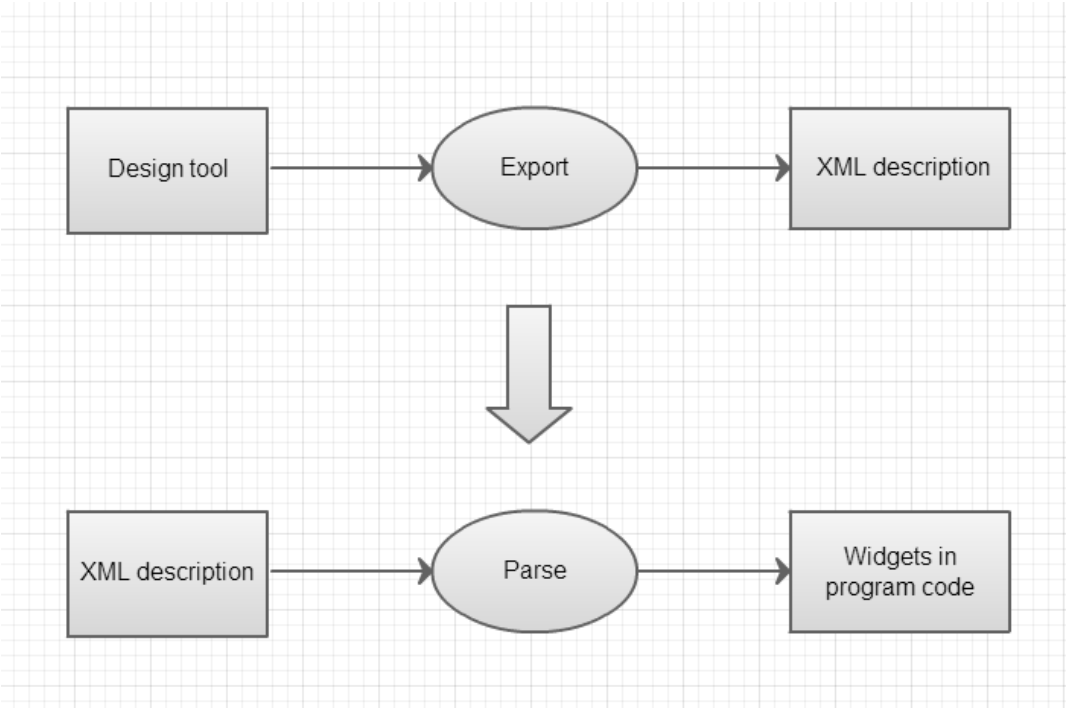

Figure 8: Data flow in the UI pipeline

3.2.2 ...for machines

Although XML is considered human-readable (discussion about the validity of that sentiment is outside the scope of this thesis), the data described in it is eventually consumed by program code to construct objects represented by the XML data. Thus the data has to be read and understood by the code at some point before the construction of the UI code.

Especially in a CPU- and memory-constrained environment such as a mobile phone, the UI parsing can take up time and memory and has to be optimised to keep the

loading times short and the flow of the program smooth. A lot of this optimisation can be done in designing the XML layout syntax, so that there is as little additional logic in reading the description as possible. Unfortunately, this can result in repetitive XML that consists of long lists and is fairly unreadable. In this thesis's case, there is one tag in the XML that is somewhat monstrous to read (the area tag list for different device aspect ratios) but it makes the parsing a bit faster and easier to do in the code.

An example of the area tag:

```
<element name="example_element">
  <areas>
    <area target_aspect_ratio="1.33333"
        x="0.1" y="0.5" width="0.8" height="0.3" />
    <area target_aspect_ratio="0.75"
        x="0.2" y="0.25" width="0.7" height="0.5" />
    [... more area tags for each aspect ratio]
  </areas>
  ...
</element>
```

The real description contains more attributes and properties but they are not shown to keep the example simple.

3.2.3 ...for humans

The UI description in XML is not intended to be edited by humans during the UI description process, but nothing prevents from doing so. This means that somewhere, someone is going to edit it fully by hand. And for this reason, the description needs to be human readable. This was achieved by asking questions such as "What does this tag really describe?" and "What does this tag not describe?" when coming up with tags. Naming and terminology plays a big part in this and good XML tag names can really help the designer to understand how the elements work.

It was also helpful to defend the developer's point of view when challenged about different UI tags, as you could justify your decisions by rationalising the ideas behind

names. For instance, a Container element was named so because it contains other elements. Thus it should not affect any peer elements on its level, it should care only about its child elements.

Below is an example of a fully described element in XML:

```
<element name="example_element" type="label"
        visible="1" touchable="1" touch_propagation="0">
  <areas>
    <area target_aspect_ratio="1.33333"
        x="0.1" y="0.5" width="0.8" height="0.3" />
    <area target_aspect_ratio="0.75"
        x="0.2" y="0.25" width="0.7" height="0.5" />
  </areas>
  <color argb="n/a" />
  <gradient argb1="n/a" argb2="n/a" direction="n/a" />
  <image src="n/a" frame="n/a" frame_x="n/a" frame_y="n/a"
        stretch_mode="n/a" />
  <text_field>
    <text_values local_id="42" string="Some example text" />
    <color argb="ffffff" />
    <font name="arial" size="0.3" />
    <alignment horizontal="center" vertical="bottom" />
    <multiline value="1" />
  </text_field>
</element>
```

The text field element was broken into multiple child elements for the purpose of keeping it readable. It was initially a list of attributes in a single XML tag, but it grew too long. The same treatment was done to a few other elements, as well.

The features of the XML description are illustrated in Figure 9.

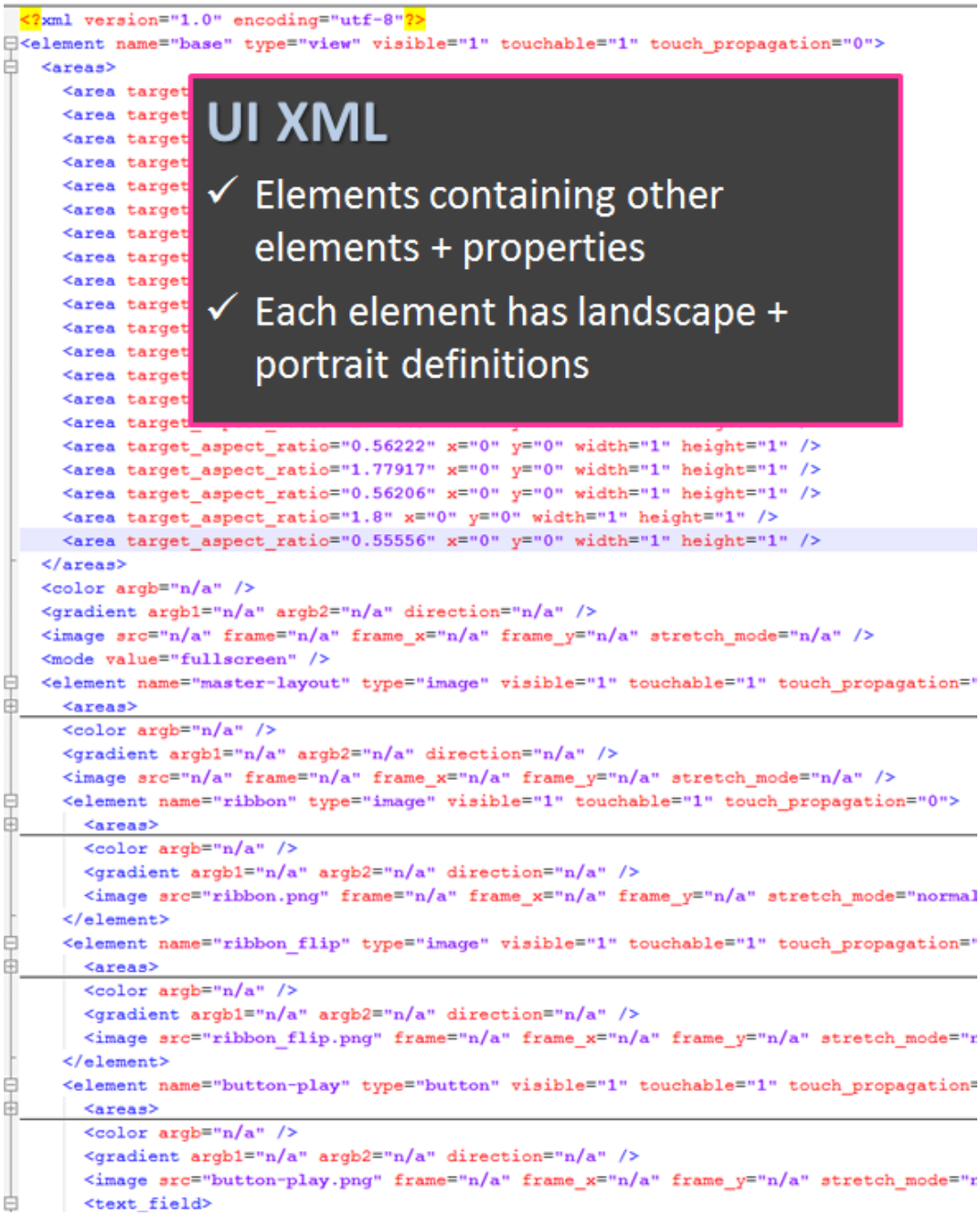

Figure 9: The XML description

3.3 Replicating the description in code

3.3.1 Widget hierarchy

The XML description resides as a file in the game assets, from where it is read into the device's memory as a hierarchy of widget objects. The widget hierarchy is found in many other UI libraries as well, such as Qt, SFML and wxWidgets. The widget hierarchy is based on inheritance, which was suggested by the technical lead in Star Arcade. The original plan was to base the library on composition instead of

inheritance, which would have been a more modular and modern way of programming the library.

Every widget is based on the Widget class, which contains default functionality for drawing and positioning. Specialised widgets, such as Button and Label, are derived from Widget and they add special functionality relevant to their purposes. For example, Button contains functionality to display a click animation when it is touched, although every Widget handles the basic touch event. This kind of extending upon existing functionality is present commonly in all specialised widget types.

As seen in Figure 12, the widget hierarchy is small and each widget has a logical real-world role. Introducing intermediary widgets (such as different layout configuration widgets) to the widget family would have made the system significantly more difficult to comprehend and it would have caused implications on the UI description itself, which has to be comprehensible to non-programmers.

The class names conform to the Star Arcade naming conventions, but are here simplified for the thesis.

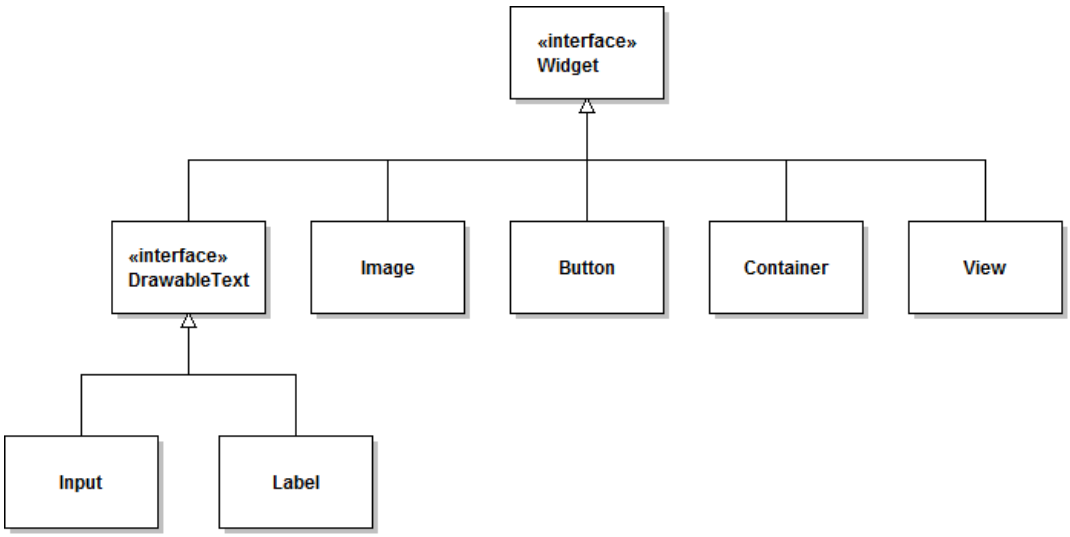

Figure 10: The widget class hierarchy

The View class is a little bit different to other derived widgets, as it is the container for a single layout or UI page. The View handles all the drawing, input and other common tasks of its child widgets, and the views in turn are managed by a UI manager class, which was created by the Star Arcade lead coder Arto Katajasalo.

The UI manager takes care of activating and deactivating views, and it also handles their input.

The XML parsing is done in a loader class that constructs widgets according to the XML data. It reads the widget's type first, and constructs an object of that type. Then it proceeds to parse and fill name, size, position and other data.

Calling the UI loader is simple in game code:

```
// start by initialising UI loader
UILoader pLoader;
pLoader.Startup();

// test view
TestView* pTestView = TestView();
pTestView->Startup();

retcode = pLoader.LoadFromFile( L"testview.xml", pTestView );
if( OK != retcode )
{
        pTestView->Shutdown();
        delete pTestView;
        return retcode;
}
// add view to UI manager hierarchy
m_pUiManager->AddView(pTestView);
// activate the view, displaying
// it on screen with activation animation
m_pUiManager->ActivateView(pTestView->GetId());

// shutdown UI loader
pLoader.Shutdown();
```

Another way to construct UI objects is to create them by hand in program code. This is often needed when completely new objects are created to reflect some data, such as player lists. One of the main features of the UI library was to be able to

construct widgets and widged hierarchies without any external XML description.

Here is an example of constructing a view by hand and adding it to the manager:

1. First create a test view:

```
// create test view
CDieselFloatRect viewArea(0.0f, 0.0f, 1.0f, 1.0f);
TestView* pTestView = new TestView();
pTestView->Startup();

// set view properties
pTestView->SetArea(viewArea, Widget::Orientation_Portrait);
pTestView->SetArea(viewArea, Widget::Orientation_Landscape);
pTestView->SetName(L"test_view");
pTestView->SetMode(Mode_Fullscreen);
pTestView->SetFillColor(0xffff0000);
```

2. Create a test widget to display inside the view:

```
// create test button
Button* pTestButton = new Button();
pTestButton->Startup();

// set button properties
pTestButton->SetImage(L"button_image.png",
                      Widget::StretchMode_Normal);
pTestButton->SetOrientation(Widget::eOrientation_Portrait);
CDieselFloatRect buttonRect(0.0f, 0.0f, 0.5f, 0.5f);
pTestButton->SetArea(buttonRect,
                     Widget::eOrientation_Portrait);
pTestButton->SetArea(buttonRect,
                     Widget::eOrientation_Landscape);
pTestView->AddChild(pTestButton);
```

3. Add the view to UI manager:

```
// add to UI manager
m_pUiManager->AddView(pTestView);
```

A real-world example is seen in Figure 11:

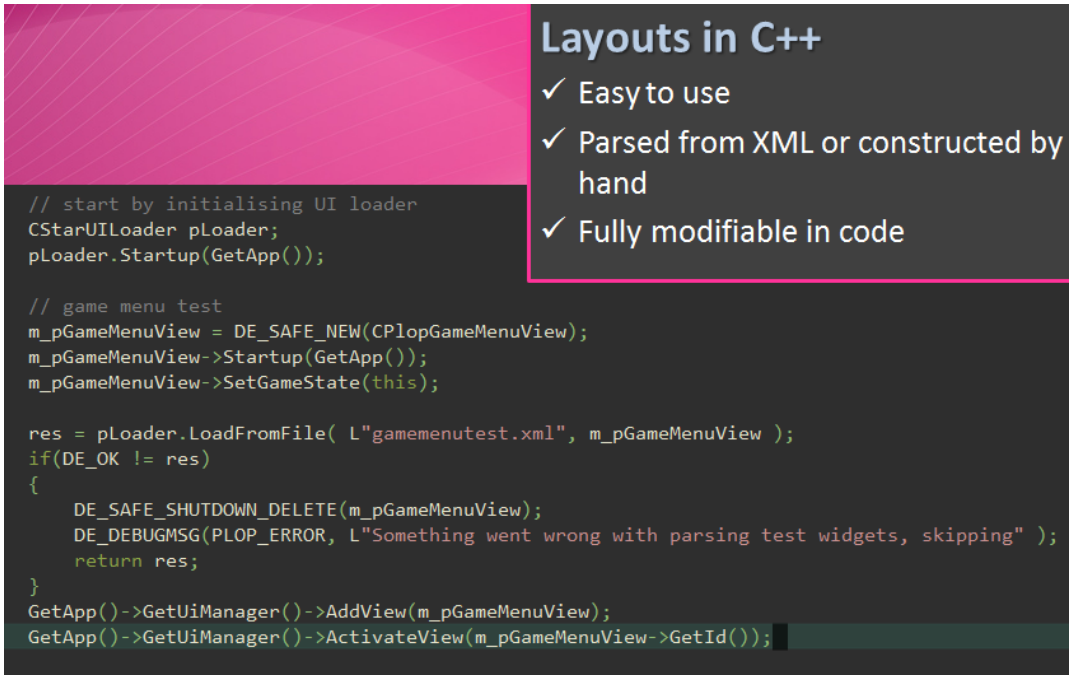

Figure 11: C++ code utilising the UI library

As soon as views are added to UI manager, they are updated and drawn automatically. The widgets use an Observer pattern to inform observers about changes in their state, such as activation, deactivation or touch.

3.3.2 Dynamic approach

The UI design produces a description that is a snapshot of the layout in its default state. As the program runs, the layout needs to display the data according to the state of the program and the positioning and visibility of UI elements will change. As a data-driven system, it has no need to know what kind of information it is displaying, as long as it is given instructions which objects to display and what data to replicate. Sometimes the user has to be able to re-arrange items on screen and, for example, scroll lists.

All this mutability creates a need for the system to be fully modifiable during runtime. It also requires the layout designers to acknowledge that their UI designs will be only the initial snapshot of the final layout, and that means that extra work has to be done to for instance restore designer positions of items that are destroyed

and re-built or moved. This kind of co-operation was needed throughout the process of layout design, and the resulting combination of opinions offered valuable insight from other people's points of view.

The widgets store a lot of data of their position and size, which results in having no need to read them from XML more than once, at the cost of increased memory footprint. Below is a snippet of the base class Widget's header to illustrate the data properties that every widget contains.

Data members:

```
View*           m_pEventHandler;
Surface*        m_pImage;
int             m_iFrameIndex;
uint            m_iFillColor;
FloatRect       m_rcImageSubRect;
Widget*         m_pParent;
bool            m_bVisible;
bool            m_bTouchable;
bool            m_bMouseButtonPressed;
WidgetType      m_eType;
StretchMode     m_eCurrentStretchMode;
uint            m_uTouchIndex;
Orientation     m_eCurrentOrientation;
WidgetArray     m_arrChildren;
PtrArray        m_arrObservers;
bool            m_bIsCloned;
void*           m_pUserData;
```

Area definitions:

```
// relative areas
FloatRect       m_rcPortraitArea;
FloatRect       m_rcLandscapeArea;
// draw areas
FloatRect       m_rcPortraitDrawArea;
FloatRect       m_rcLandscapeDrawArea;
```

Definitions specific to displaying stretched "corner images":

```
// corner image
FloatRect       m_rcCornerSubRect1; // top left
FloatRect       m_rcCornerSubRect2; // top right
FloatRect       m_rcCornerSubRect3; // bottom left
FloatRect       m_rcCornerSubRect4; // bottom right
FloatRect       m_rcCornerPos1; // top left
FloatRect       m_rcCornerPos2; // top right
FloatRect       m_rcCornerPos3; // bottom left
FloatRect       m_rcCornerPos4; // bottom right
FloatRect       m_rcGapSubRect1; // top
FloatRect       m_rcGapSubRect2; // left
FloatRect       m_rcGapSubRect3; // right
FloatRect       m_rcGapSubRect4; // bottom
FloatRect       m_rcGapPos1; // top
FloatRect       m_rcGapPos2; // left
FloatRect       m_rcGapPos3; // right
FloatRect       m_rcGapPos4; // bottom
FloatRect       m_rcCenterPos;
FloatRect       m_rcCenterSubRect;
float32         m_fCornerImageWidth;
float32         m_fCornerImageHeight;
```

3.3.3 Generic special cases

As seen in the class hierarchy, the View class is a derived Widget. The same applies to Container, yet they both differ from Button and the other derived widgets in their logic. They are not special cases per se, but the fact that they do not represent similar functionality as their peers is a sign that there is room for improvement in this regard. Many times the requirements handed to the engineer are prone to constant changing, and they almost always mirror the features of a system that fits the needs of the company at current time. However the current situation is a specialisation of a more generic system that represents the all possible use cases and features needed for different projects altogether. And since the plans

are prone to change, the system should take into account that the requirements define a special case which needs to be translated into more general terms.

The Container was especially hard to implement in a way that would cater for all possible situations, when it needed to arrange its child widgets very specifically in different views. It has two ways of obtaining child widgets: as templates and as normal widgets. The template widgets are not drawn with the Container, but they act as if they were on a shelf from where the user can quickly access them and clone new widgets from the template widgets. The cloned widgets are in turn added to the Container hierarchy to represent the data.

For example, when a player list is displayed, multiple clones of a single player template are instantiated and added as normal child widgets to the Container. But what happens when there has to be a set of items already defined, which do not belong to the player list data set, but act as a part of the list? In the example, the player list contains a search box which is scrolled with the player items when the user flicks a finger on the list. The search box is not deleted with the data-spawned child widgets, but it is moved with them. These situations proved so troublesome to handle that eventually the pre-defined widgets were removed from the specification altogether and only templates and added clones were allowed inside the Container.

3.4 View-based environment

When a widget is constructed, it is handed to a View that is handed to UI manager. They form a chain of handlers that take care of a lot of boilerplate iterating, adding, removing and observer informing. The result is a neat system that performs complex and laboursome tasks to ensure it is easy and logical to use. The Star UI consists of a collection of View classes that each implement their own touch handling, usability interface and business logic.

A fabricated example view's header file could look like this:

```
class TestView : public View
{
public: // constructor & destructor
```

```
        TestView();
        virtual ~TestView();

public: // this particular view's interface
        void SetState(State* pState);

public: // virtuals for upper classes
        virtual void HandleWidgetEvent(Widget *pSender,
                                       EventType eventType);

        virtual void OnViewStateChange(ViewState eState);

private: // data
        State*          m_pState;
};
```

The events propagate through the UI manager to the views, which handle the events and return the handled event so that it won't propagate further. The sequence can be seen in Figure ??.

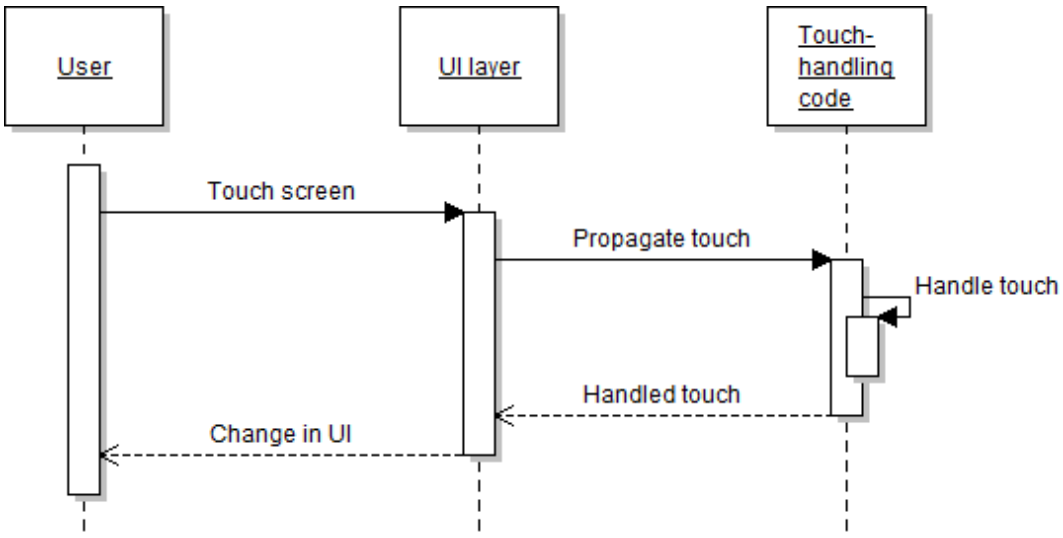

Figure 12: The sequence of UI event handling

The HandleWidgetEvent function is a pure virtual function that needs to be implemented by every widget class separately. It receives the information about

touches, which are in turn reacted to in view's class code. OnViewStateChange is a virtual function that is called when the view is activated or deactivated. It can be used, for instance, to initialise data before displaying it when the user activates the view.

A screenshot from a game utilising the UI library is seen in Figure 13. King of Words (c) Star Arcade 2013.

Figure 13: The UI library in use

4 Conclusions and implications

4.1 The project's outcome

The project took some six to nine months to get to finished status, and it was mainly a success at least at the moment of writing this thesis. The UI system was taken into use at Star Arcade quite early during the development process. As of spring 2013, it has been included in one published game and another game utilising it is waiting to be released. The library is also a part of the Star SDK, which is to be released during summer 2013.

According to the feedback received from the project, the library was better than the old UI library in terms of ease of use and functionality. On the other hand, its functionality was different to the views of the technical lead, which was largely a result of the constant change in requirements, the inexperience of the author of this thesis in designing large and complex systems, and corner cutting due to tight time frames. The help and support of the lead coder and the design tool coder was much needed.

4.2 Cycles of development

The constantly changing requirements during implementation caused the need to refactor the code multiple times, which took a large amount of time and was frustrating from the developer's point of view. This resulted in some big changes in the code that left the untouched code look like legacy code. This is almost never a good thing in programming, and yet there was no time for a full re-write of the system. With this in mind, the system could have been a lot more finished and robust at the end of the development process.

As seen in Figure 14, the sprints were interrupted by changing ideas that had to be implemented immediately. This happened because the whole development platform in Star Arcade was under re-construction and various ideas were tested out.

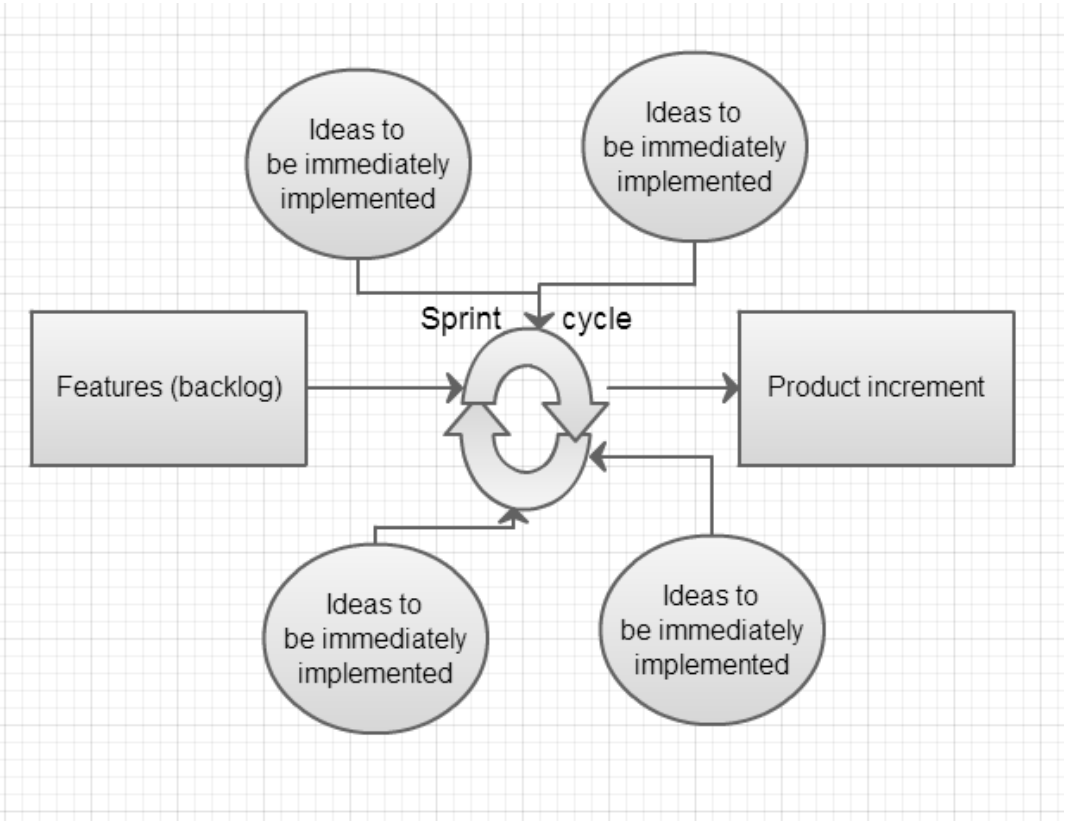

Figure 14: A development sprint

4.3 The future of the library

During the development process, many changes to existing ideas made it appealing to refactor large portions of the code at once. However, the time constraints tightened around the middle point of the timeline, and rendered larger refactoring simply impossible to do until the launch of the new system. This meant that every new feature was likely to not change too much even if it was proven to be less than ideal.

In the future, the library is likely to be either replaced by a new one or refactored heavily. The change backlog at the end of the project was already quite long, and contained enough features for a completely new system. However, the core of the library is quite solid and likely to remain in use in the future, as it enables the actual implementation to vary and still be a powerful tool that is also easy to use.

References

Meyers, S. 2005. Effective C++, Third Edition: 55 Specific Ways to Improve Your Programs and Designs. New Jersey: Addison-Wesley Educational Publishers Inc.

Company presentation. 2012. Brochure. Star Arcade promotion material.

Definition of: cross platform. N. d. Article on PCmag website. Retrieved 6.5.2013. http://www.pcmag.com/encyclopedia/term/40495/cross-platform#fbid=aHfb3ldkqPq.

Stutz, Michael. 19.9.2006. Get started with GAWK: AWK language fundamentals. developerWorks. IBM. Retrieved 6.5.2013. http://www.ibm.com/developerworks/aix/tutorials/au-gawk/section2.html