Helsinki University of Technology

Department of Electrical and Communications Engineering

Sami Rantala

Usability of user interface markup language

Thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Engineering

Espoo, April 12, 2001

Supervisor:	Marko Nieminen, Professor of User interfaces and usability
Instructor:	Gerry Callaghan, B. Sc.

Helsinki University of Technology **Abstract of the Master's Thesis**

Author:	Sami Rantala		
Name of the Thesis:	Usability of user interface markup language		
Date:	April 12, 2001	Number of Pages:	88

Department of Electrical and Communications Engineering
Professorship: Tik-86 User interfaces and usability

Supervisor: Marko Nieminen, Professor of User interfaces and usability
Instructor: Gerry Callaghan, B.Sc.

User Interface Markup Language (UIML) is an appliance-independent language for creating user interfaces. This study concentrates on how well user interfaces can be created with UIML, both from developers' and end-users' perspective. However the emphasis is on the developers' perspective.

First the existing knowledge is reviewed both on UIML and research methods. After this two sets of user interfaces are built for testing. The first set is built with platform-dependent languages, e.g. Java and Wireless Markup Language (WML), and the second with UIML. A set consists of a PC application and a WML application. The evaluation criteria and methods are defined before testing.

The user interfaces created during this thesis are identical within the limits given by the pre-release version of the renderer, which is the program that renders the UIML user interfaces. These limits bring out the biggest problem with UIML. It is totally dependent on the implementation of the renderer.

Depending on the target platform, the amount of UIML code needed to achieve the same outcome as with the platform-dependent languages ranges from only slightly more to double of the comparable platform dependent language. Mapping one UIML document to multiple platforms proved to be a difficult task as the platforms' user interface toolkits, e.g. Java and WML, were structurally very different.

Despite the limitations of UIML, it is a moderately good language for user interface creation. If good tools are made available and the renderers are up to the job, then UIML may be a good solution for multiple-platform user interfaces.

Keywords: User Interface Markup Language (UIML), user interface (UI), XML, usability, appliance-independent.

Teknillinen Korkeakoulu **Diplomityön Tiivistelmä**

Tekijä:	Sami Rantala		
Työn Nimi:	User interface markup language:n käytettävyys		
Päivämäärä:	huhtikuu 12, 2001	Sivumäärä:	88

Osasto: Sähkö- ja tietoliikennetekniikka
Professuuri: Tik-86 Käyttöliittymät ja käytettävyys

Työn valvoja: Professori Marko Nieminen
Työn ohjaaja: Gerry Callaghan, B.Sc.

User Interface Markup Language (UIML) on alustariippumaton käyttöliittymien luomiseen tarkoitettu kieli. Tämä diplomityö keskittyy siihen, miten hyvin käyttöliittymiä voidaan luoda UIML:llä loppukäyttäjän ja suunnittelijan perspektiivistä. Pääpaino on kuitenkin suunnittelijoiden puolella.

Ensimmäiseksi käydään läpi oleellista taustatietoa. Seuraavaksi luodaan kaksi paria testikäyttöliittymiä. Ensimmäinen pari on luotu Javalla ja WML:llä (Wireless Markup Language). Toinen pari on luotu UIML:llä. Molemmat parit sisältävät PC-käyttöliittymän ja WAP-käyttöliittymän (Wireless Application Protocol). Käyttö-liittymien arvosteluperusteet on määritelty ennen testauksen aloittamista.

Luodut käyttöliittymät ovat identtisiä, kun ottaa huomioon UIML-tulkin keskeneräisyyden. UIML-tulkki (eng. renderer) on ohjelma, joka luo käyttöliittymän UIML-dokumentin perusteella. Tämä tuo esille UIML:n suurimman ongelman. UIML on täysin riippuvainen UIML-tulkin laadusta.

Riippuen alustasta UIML:n vaatima koodimäärä verrattuna vastaavaan alustariippuvaisella kielellä tehtyyn käyttöliittymään vaihtelee paljonkin. Javan ollessa kyseessä suhde on 124% ja WML:n 230%, molemmissa tapauksissa alustariippuvaisten kielten eduksi. Suuri rakenteellinen ero Javan ja WML:n välillä esti käyttämästä samaa UIML-dokumenttia molempiin käyttöliittymiin.

Rajoituksistaan huolimatta UIML on melko hyvä kieli käyttöliittymien luomiseen. Jos UIML saa kunnon työkalutuen, mukaanlukien tulkit, siitä voi muodostua hyvä ratkaisu, kun tarvitaan saman ohjelman käyttöliittymiä monelle eri alustalle.

Avainsanat: User Interface Markup Language (UIML), käyttöliittymä, XML, käytettävyys, alustariippumattomuus.

Preface

I would like to thank Professor Marko Nieminen for his guideance through the entire thesis. His pointers guided me to the right path from the beginning.

Special thanks to Gerry Callaghan who presented the subject and patiently read through my thesis many times and gave invaluable input. Also, big thanks to Eimear Lyons, Danny Owens, Jennifer Allen and Denis Hackett for their help.

I am grateful to Ericsson and especially Harri Oikarinen for making my thesis possible.

Special thanks to Pia Lappalainen who helped me greatly by checking the grammar and spelling of this thesis.

Finally I would like to thank all the people that gave me advice and support. The completion of this work would have been impossible without you. Thank you.

Table Of Figures

Table Of Contents

Abbreviations

Abbreviation	**Explanation**
DTD	Data Type Definition
IT	Information Technology
PC	Personal Computer
PDA	Personal Digital Assistant
UI	User Interface
UIML	User Interface Markup Language
WAP	Wireless Application Protocol
WML	Wireless Markup Language
XML	EXtensible Markup Language
XSL	EXtended Style Language
HTML	HyperText Markup Language

1 Introduction

Nowadays people want to be able to access IT services, wherever they are. This means that the same services must be available on many different appliances, such as desktop PC, small PDAs, WAP phones and voice phones.

It is time consuming to create user interfaces for each of these appliances independently. Ideally an interface should be specified without any device-dependent information and mapped to different devices as necessary. This is where user interface markup language steps in. UIML is an appliance-independent language for describing user interfaces.

1.1 Background

Abrams (1999) defines three reasons for developing the UIML:

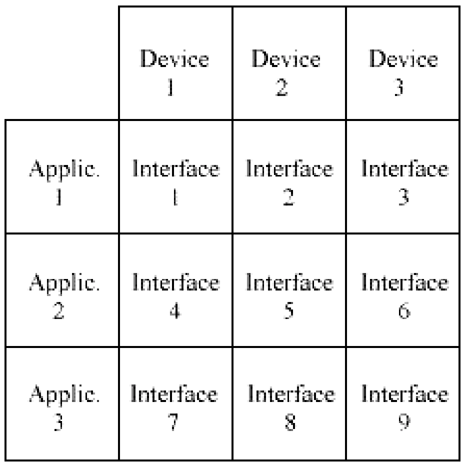
- Historical motivation or raising the abstraction.
- Managing the family of interfaces.
- Avoiding market risks when developing applications for new appliances.

These reasons are inspected in more detail in the following paragraphs.

In the early stages of computers, programs were written in binary machine code. Time passed and higher-level languages became available. Higher-level languages are more abstract than lower-level ones and allow less skilled people to program. This same trend can be seen in user interface creation. In the beginning there were no readily available user interface components and everything had to be made from scratch. As time passed, toolkits for user interfaces emerged. These toolkits were collections of readymade user interface components that anyone could use. Nowadays one can build the user interfaces using toolkits and graphical drag & drop tools. User interface designers come from many different backgrounds, from human factor specialists to cognitive psychologists, from graphic designers to programmers. It is necessary to have a simple and flexible process for creating user interfaces. Raised abstraction may allow non-programmers to create user interfaces.

Consider a situation where the same application is run on PC, handheld PC and cellular WAP phone. All of the appliances need their own user interfaces. All of these user interfaces have common parts, since they are just different views of the same application. Using appliance-independent user interface language would mean that there is no need to write the common code for appliances multiple times. Maintaining consistency across appliances is easy, since they all use the same data to create the user interfaces.

Appliance-dependent style sheets are used to map the user interfaces into the current appliances. All applications have appliance-independent code. The situation is even better when multiple applications are running on multiple appliances. As can be seen in Figure 1 and Figure 2, the amount of code needed to maintain 3 applications on three appliances is reduced dramatically by using appliance-independent user interface language. When using appliance-dependent language all applications need separate user interfaces for all devices. On the other hand, when using appliance-independent language, applications need one user interface description per application and one style sheet per device. Style sheets contain the appliance-dependent data. Style sheets are reusable for different applications.



	Device 1	Device 2	Device 3
Applic. 1	Interface 1	Interface 2	Interface 3
Applic. 2	Interface 4	Interface 5	Interface 6
Applic. 3	Interface 7	Interface 8	Interface 9

Figure 1. User interfaces needed with an appliance-dependent user interface language.

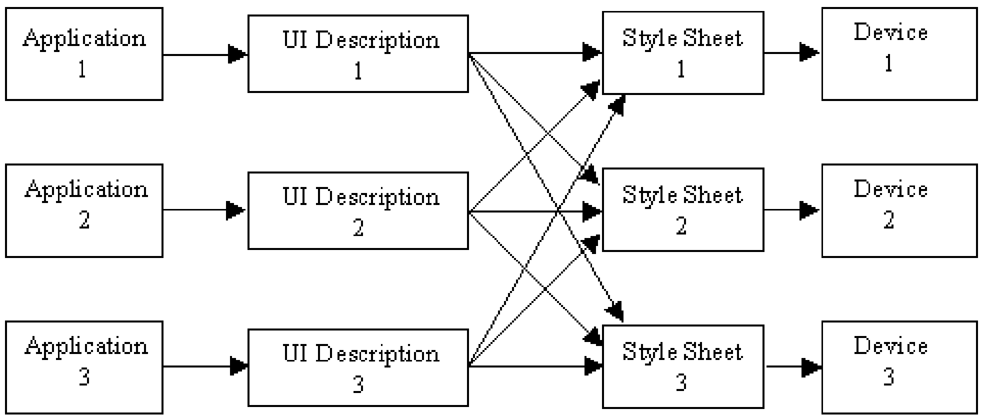


Figure 2. Files that are needed when using an appliance-independent user interface language.

There are always risks in developing applications for new platforms. Using an appliance-independent user interface language would allow the reuse of user interface description and lower the risks when adopting new technologies.

1.2 *Research problem*

A reduction in the amount of user interface code needed for a single application would reduce the workload of the developers. This would help the developers in creating new user interfaces and maintaining the old user interfaces. But to be able to find out if UIML really helps to reduce the amount of code, the following question needs to be answered. Is it possible to create an application that runs on multiple appliances with just one user interface definition and one style sheet per appliance?

If UIML is truly appliance-independent, then it probably reduces the amount of source code needed for the user interfaces. But some things are harder to accomplish than others. So it must be found out, if working with UIML is actually harder and more time-consuming than with appliance-dependent methods.

There is still one important issue to be considered. UIML is a generic way of describing user interfaces. User interfaces created with UIML may not be as good as user interfaces created with appliance specific tools. It must be verified if this is the case.

1.3 Objectives of the study

1. To find out UIML's limitations and problems in a multi-appliance environment.
2. To find out UIML's limitations and problems in user interface creation.
3. To evaluate the usability potential of UIML from the developers' perspective.

1.4 Scope of the study

The scope of the study is limited to one test application on two appliances. Analysis is based on expert opinions. Some usability testing is done, but most of the emphasis is on expert opinions. This is due to the fact that the thesis focuses on inspecting a user interface language and not user interfaces. There are no user interface building tools based on UIML yet and this restricts the ability to create user interfaces for testing.

The amount of appliances used in this study is limited to two. The appliances used are a standard PC environment and a WAP cellular phone, which is simulated with a program.

The most limiting factor of this study is that only one person can be used in the actual work of creating the user interfaces.

1.5 Structure of the thesis

The first section is called Introduction and it goes through the problem domain and defines the objectives for this thesis. The second section is called Review of Existing Knowledge and it describes the problem domain and the research background necessary for this thesis.

Third section, Construction of the Application, goes through the selection of the test application domain, the creation of scenarios and the actual construction of the test applications.

Evaluation of the application, which is the fourth section, defines the evaluation criteria and methods used during this study. The fifth section, Results of the Study, states the results based on the evaluation methods defined in the previous section.

The sixth section is called Conclusion And Discussion and in this section the results are inspected against the evaluation criteria. The validity and reliability of this thesis are evaluated in this section. Also, any other observations concerning the study are stated.

The seventh and the final section is called Summary and it gives an overview of the results and the conclusions.

2 Review of Existing Knowledge

2.1 Definition of usability

Usability is not just one easily quantifiable property. Usability is traditionally defined by five attributes (Nielsen 1993, 26).

- Learnability: The system should be easy to learn. This allows new users to quickly start productive work.
- Efficiency: Once the system has been learned, it should be efficient to use to allow high level of productivity.
- Memorability: The system should be easy to remember. This allows the casual user to return to the system without having to relearn the whole system.
- Errors: The system should be built so that the error rate is low. When errors occur it should be easy to recover from them.
- Satisfaction: The system should be pleasant to use.

The attributes that concern this thesis are learnability, efficiency and errors. Memorability and satisfaction do not fit into the scope of this study, as inspecting memorability would take more time than is available. Measuring satisfaction needs more than one test subject for the results to be valid and there is only one test subject available. In this case the test subject is the same as the user interface programmer. Similarly, it is not possible to study learnability of UIML with the normal usability methods within the boundaries set for this thesis.

The view taken on usability in this thesis is a little different from traditional usability engineering. This is due to the fact that this thesis is about inspecting a user interface language instead of a user interface. Here are the three attributes which concern the thesis, and slightly modified explanations that suit to the objectives of the thesis better.

- Learnability: UIML as a language should be easy to learn.

- Efficiency: UIML as a language should allow designers to create user interfaces efficiently and without limiting the designer’s options.
- Errors: Creating user interfaces with UIML should not be more difficult than appliance-dependent languages.

2.2 XML in a nutshell

This is a short introduction to the world of XML. The acronym XML stands for eXtensible Markup Language. XML is a markup language that is used to define new markup languages, in other words XML is a metalanguage.

World Wide Web Consortium is responsible for the development of XML and they have published a recommendation (1999), which defines the language.

2.2.1 XML notation in brief

The XML document consists of a number of elements. In Figure 3 you can see an example of an XML element. An element has a name and optional attributes. There can be any number of attributes in an element. As Figure 3 shows, an attribute consists of an attribute’s name and an attribute’s value.

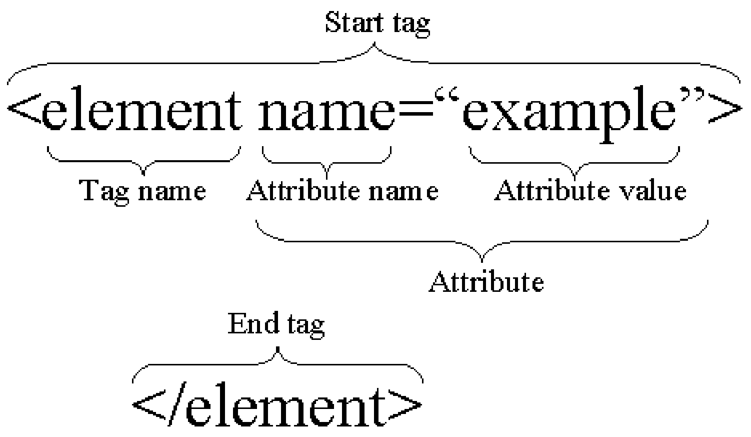


Figure 3. An XML element with a start tag and its corresponding end tag.

An element may have text or other elements placed between its start and end tags. Usually the element identifies the nature of the content they surround. If the element has no content, then a shorter notation, shown in Figure 4, can be used.

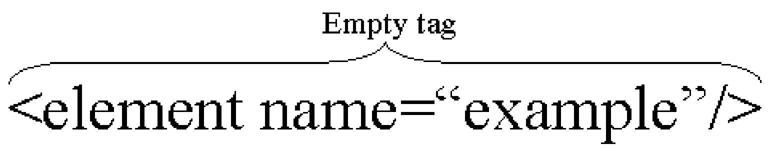


Figure 4. An empty element

2.2.2 Well formedness

Every XML document must be well formed. This means that the document must follow all of the notational and structural rules of XML. The most important of these rules are:

No unclosed tags.

Each and every element must be closed, that is, they must have an end tag. If the element is empty it can use the shorthand shown in Figure 4.

No overlapping tags.

An element that is inside another element must close before the containing element closes. This XML document, for example,

<sport>
<basketball></sport>
</basketball>

is not well formed, because the <basketball> element, which opens inside <sport> element, is not closed inside of the <sport> element. The correct way to do it would be like this:

<sport>
<basketball></basketball>
</sport>

Attribute values must be enclosed in quotes.

The correct way to assign values to the attributes is shown in Figure 3 and Figure 4.

The text characters <, > and ” must always be represented by ‘character entities’.

When these three characters are used in the text part of the XML, they must be represented by special character entities, as shown in Table 1. Note that this does not apply to the markup language itself.

Table 1. List of characters to be replaced when used in the text part of XML document.

Character	Replacing character entity
<	&lt;
>	&gt;
“	&quot;

2.2.3 Data type definition

Data Type Definition (DTD) is the grammar for a markup language. The designer of the markup language defines the DTD for that particular language. The DTD specifies the elements and which attributes the elements may have. It also specifies which elements may or must be found inside other elements and in which order.

If the XML document is well formed and follows the rules set in the DTD, then the document is valid XML.

2.2.4 Style sheets in XML

XML can be used with two different style sheets. The older one is cascading style sheets or CSS and it can also be used with HTML. The newer one is called extensible style sheet language or XSL and it can only be used with the XML.

CSS (World Wide Web Consortium, 1998) can be used to specify formatting semantics. For example, it could specify that the text contents of a certain element should always be displayed in a red font.

XSL (World Wide Web Consortium, 2000) is a language for expressing style sheets. It can be used to transform XML documents and to specify formatting semantics. In addition to being capable of performing the same actions as CSS, XSL could, for example, be used to transform a UIML document into an HTML document.

2.3 Introduction to UIML

This is a short introduction to a XML language called UIML. This section will go through UIML's structure and elements with the help of simple examples. Also, it will show how user interfaces are created from UIML documents. A company named Harmonia Inc. is responsible for the development of the UIML specification.

2.3.1 Basic structure

This is the basic structure for a UIML document:

```
<uiml>
<head>
    <meta> . . . </meta>
</head>
<peers> . . .  </peers>
<template> . . . </template>
<interface>
    <structure>
        <part> . . . </part>
    </structure>
    <style>
        <property> . . . </property>
    </style>
    <content>
        <constant> . . . </constant>
    <content>
    <behavior>
        <rule> . . . </rule>
    </behavior>
</interface>
</uiml>
```

2.3.2 UIML example

Here is an example of an UIML document. The first three lines define that this is an UIML document. All XML documents have similar lines. The part that is inside UIML tags is the actual UIML document.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE uiml PUBLIC "-//UIT//DTD UIML 2.0 Draft//EN"
"C:\UIML2\bin\UIML2_0d.dtd">
<uiml>
    <interface>
        <structure name="JavaSwing">
            <part name=" exampleFrame" class="JFrame">
                <part name="button" class="JButton"></part>
            </part>
        </structure>
        <style name="JavaSwing">
            <property part-name="exampleFrame" name="title"> Example </property>
            <property part-name="exampleFrame" name="size"> 300,100 </property>
            <property part-name="button" name="text"> Press me </property>
        </style>
        <behavior>
            <rule>
                <condition>
                    <event class="actionPerformed" part-name="button"/>
                </condition>
                <action>
                    <property part-name="exampleFrame" name="title">Button was pressed
                    </property>
                </action>
            </rule>
        </behavior>
    </interface>
</uiml>
```

This UIML document produces the user interface shown in Figure 5 and Figure 6.

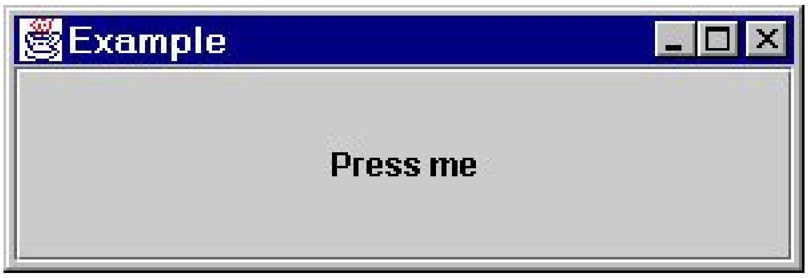


Figure 5. User interface created with UIML

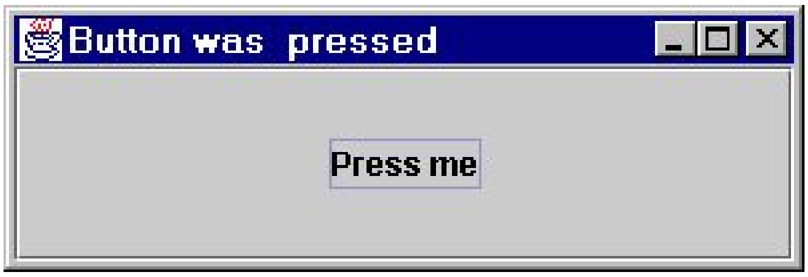


Figure 6. UIML user interface after button was pressed

2.3.3 UIML elements

This section presents the major UIML elements. Whenever possible the example in the subsection 2.3.2 is used as a basis.

Interface

An interface element is composed of four other elements. It is a container for these elements. The elements are structure, style, content and behavior.

Structure

The structure element is a hierarchical presentation of the user interface parts. Each part has an instance name and a class. The instance name uniquely identifies the part. The class identifies the part as being an element of a certain class. The user interface implementers can freely choose the names and classes. Multiple structure hierarchies can be used. It would look like this:

```
<structure name="JavaSwing">
    <part name=" exampleFrame" class="JFrame">
        <part name="button" class="JButton"></part>
    </part>
</structure>
<structure name=”secondStructure”>
     . . .
</structure>
```

The structure’s name attribute can later be used to choose which structure hierarchy is to be used. The JavaSwing structure defines that there is a JFrame component, which has one JButton inside it. JFrame and JButton are Java Swing components.

Style

The style element specifies the presentation style for the interface parts. The presentation is device-dependent and consequently each device needs its own style elements.

Below is shown two different ways for the JavaSwing structure to render the document. The difference is that if the BigJavaSwing style element is used, then the size of the exampleFrame is doubled.

```
<style name="JavaSwing">
    <property part-name="exampleFrame" name="title"> Example </property>
    <property part-name="exampleFrame" name="size"> 300,100 </property>
    <property part-name="button" name="text"> Press me </property>
</style>
<style name=”BigJavaSwing”>
    <property part-name="exampleFrame" name="title"> Example </property>
    <property part-name="exampleFrame" name="size"> 600,200 </property>
    <property part-name="button" name="text"> Press me </property>
</style>
```

The part-name attribute sets the property’s value for the specified user interface part and the part-class sets the value for all the interface parts of that class. If the property is set for both the part and the class, then the value set for the part is used.

Content

UIML defines the content of the interface parts in a separate content element. There can be multiple content hierarchies. For example, each language might have its own content hierarchy or there might be separate novice and expert content hierarchies. The contents of the interface parts can also be set by the application logic.

The content element is simple to use as can be seen in the following example:

```
<content name=”english”>
  <constant name=”title”> Example </constant>
  . . .
  <constant . . .> . . . </constant>
</content>
<content name=”finnish”>
  <constant name=”title”> Esimerkki </constant>
  . . .
  <constant . . . > . . . </constant>
</content>
```

There can be constant elements with the same name attributes within other content hierarchies, as shown in the example above. Before the constant actually appears anywhere in the user interface it has to be referred to. This is done by writing the property element in a slightly different way, as shown in the example below.

This example sets the value directly:

```
<property part-name=”example” name=”title”> Example </property>
```

and this refers to the constant element:

```
<property part-name=”example” name=”title”>
  <reference constant-name”title”/>
</property>
```

Behavior

The behavior of the user interface, i.e., what happens when the user interacts with the user interface is specified in the behavior element. The behavior is defined by a set of conditions and associated actions.

```
<behavior>
    <rule>
        <condition>
            <event class="actionPerformed" part-name="button"/>
        </condition>
        <action>
            <property part-name="exampleFrame" name="title">Button was pressed
            </property>
        </action>
    </rule>
</behavior>
```

The rule in the above example specifies that when the event actionPerformed happens to the part named button, the action specified inside the action element takes place. In this case the action is targeted at a part named exampleFrame. That is specified by the attribute part-name. The attribute name specifies the exampleFrame’s property which is to be set. The action means that the exampleFrame’s title is set to “Button was pressed”.

Peers

The peers element defines relations between items in UIML document and target platforms. For example, a part button in UIML document could correspond to a Java swing object called JButton.

Template

The template allows the reuse of UIML elements. Each template element can contain only one child element, but that child element can contain any number of child elements. The element that sources the template must have the same tag name as the template's only child has. There are three choices how to source the template element: replace, append and cascade.

Replace simply replaces the element, which sources the template, and it's child nodes with the template. Append attaches the template to the end of the element that sources the template. Cascade attaches the template nodes to the element, which sources the template, only if there is no node with the same name. If there is a node with the same name, cascade skips that node and tries the next one in the template.

Head

The head element contains metadata about the current UIML document. It is similar to HTML head element.

2.3.4 Rendering the user interface

Rendering is specified as follows in the UIML Draft Specification (2000) "Rendering is a process of converting a UIML document into a form that can be displayed (e.g. through sight or sound) to an end user, and with which an end user can interact."

There are two ways to render a UIML document. The first one is called compiling and the second one interpreting.

Compiling means that the UIML document is compiled into another language. Examples of possible languages are Java, WML and HTML.

Interpreting means that a program reads a UIML document and makes calls to an appliance-dependent toolkit, which displays the user interface. For example, web browsers do interpreting when presented with an HTML document.

In addition to the UIML document, a program called renderer is needed to run the UIML user interface on a device. The renderer, as the name suggests, does the rendering. When interpreting is used the renderer takes care of user interface events and function calls that are made from the rendered user interface.

When using compiling, the end result is independent of the renderer. The compiled document can be distributed and used without the renderer. On the other hand, when using interpreting both the UIML document and the renderer are needed to run the user interface. The advantage of interpreting is that in theory the user interface could be changed and adjusted run-time. The weaknesses are the need for the renderer and that the renderer introduces some overhead into user interface computations.

There are two ways of arranging remote access using UIML. In both cases the interface server creates UIML instances dynamically using the user profile as input and sends it to the renderer. The interface server is connected to the application's backbone.

The first case is shown in the Figure 7. A PC web browser requests a UIML instance from the interface server and gives possible user profile to the interface server. The user profile might include something like the user's language or personal preferences. In this case the renderer compiles the UIML document into a language that the PC web browser understands, most likely HTML.

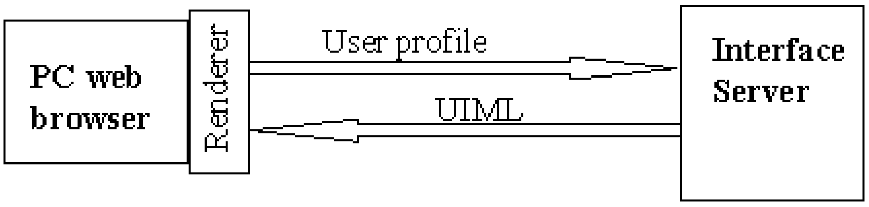


Figure 7. Remote access. Renderer is situated on the client computer.

The second case is shown in Figure 8. This is basically the same case as above with one exception. The renderer is on the interface server and it can, therefore, only be used as a compiler. The interface server returns the compiled file, in this case in WML, to the WAP cellular phone.

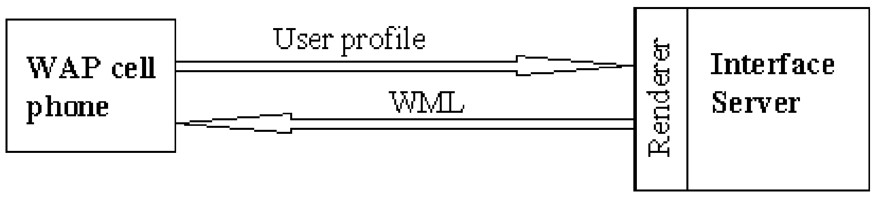


Figure 8. Remote access. Renderer is situated on the server.

When running a UIML application on a local device without remote connections, the interface server shown in Figure 7 and Figure 8 is not needed. Instead, the application itself is responsible for passing the UIML document to the renderer.

2.4 WML

WML stands for Wireless Markup Language. It is used to display information on WAP-capable cellular phones. WML is an XML-based language created and maintained by the WAP Forum. The WAP WML specification (2000) defines this language.

WML uses the metaphor of a deck of cards. A card shows as a single page on a WAP browser. A deck is a collection of cards. It is also the smallest downloadable unit of WML code.

2.5 Java and Swing

Java is an object-oriented programming language developed by Sun Microsystems. Swing is part of the Java foundation classes and it is used in creating graphical user interfaces.

2.6 Platform context

UIML can be used with a very wide range of devices and it can easily be extended to work with new devices.

2.6.1 Desktop and laptop computers

There is a wide variety of desktop computers available. They all have keyboards, pointing devices (e.g. mouse) and comparatively large view screens. They can be connected to network with broadband connections and can handle complex multiple window graphical user interfaces.

Applications on desktop and laptop computers can be divided roughly into two categories: traditional applications that run on the same computer, and browser-based applications that run on a server.

2.6.2 Handheld computer

Handheld computers are small enough to fit into a large pocket. They commonly have keyboards, styluses and touch screens. Screen resolution is limited to less than half of those available with desktop computers.

Their screen size limits the complexity of their user interfaces. Applications usually have only one window. Handheld computers can have Internet access with the help of cellular phones or with a direct link to a desktop computer.

2.6.3 PDA

Personal digital assistants (PDA) are small enough to fit into a pocket. They commonly have keypads, styluses and touch screens. Screen resolution is limited to less than half of those available with desktop computers.

Their screen size limits the complexity of their user interfaces. Applications usually have only one window. PDAs can have Internet access with the help of cellular phones or with direct links to a desktop computer.

2.6.4 WAP cellular phone

WAP cellular phones usually have small black and white screens. The size and resolution of the screen varies greatly among different phones. Applications are on computers in various locations and WAP phones connect to them through a radio link.

2.6.5 Voice phone

The ordinary voice phone, where the user listens to messages and gives commands with the keypad.

2.6.6 3rd generation products

These products are still on the drawing board. They will combine many of the properties of handheld computers, PDAs and WAP phones. They will also have broadband radio Internet accesses.

2.7 Research background

2.7.1 Research methods

Two sets of user interfaces will be created to inspect UIML. The first one is created with UIML and the second one with appliance-dependent languages as shown in Figure 9. Both sets of user interfaces are for the same application. The aim is to make the user interfaces identical.

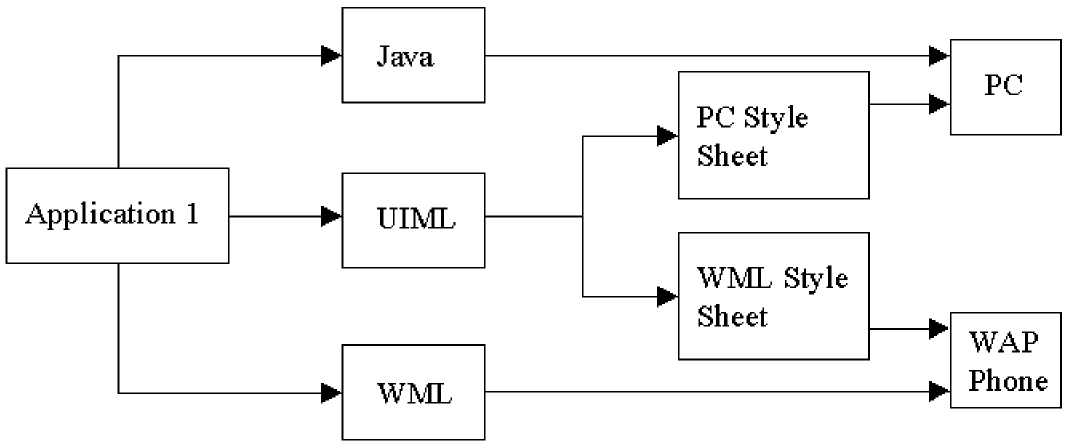


Figure 9. Applications and how they are implemented.

Java- and WML-based user interfaces are used as a basis for the evaluation of UIML. By comparing the user interfaces, one can investigate how well user interfaces can be created with UIML. By comparing the source files and the work done on them, one can see how much effort UIML takes compared to appliance-dependent languages.

Usability potential of UIML from the developer’s viewpoint is examined with expert evaluations.

2.7.2 Scenarios

Scenarios are stories of fictional characters using the system in fictional environment (Preece J. et al 1994, 462). Characters and the environment are realistic representations of real users and the environment. The stories try to capture the way the system would be used when it is completed. Multiple scenarios will be needed to reflect all the different situations that can occur.

Scenarios offer a concrete way for designers to evaluate the design in the early phases of a design process. Since scenarios describe the actual ways of using the system, they can easily be used as usability test tasks.

2.7.3 Usability testing

Usability testing with real people provides direct information about how they use the system. Usability tests consist of a test application, test subjects and test tasks. Test subjects are given tasks to do on the test application. The experimenter monitors how the test subject does these tasks.

Usability testing gives designers invaluable information on how the real users see the user interface and what are the problems with it.

2.7.4 Thinking-aloud method

Nielsen (1993, 195) says “Thinking aloud method may be the single most valuable usability engineering method.” Thinking-aloud method calls for the test subjects to continuously think aloud while they are using the test application.

Experimenters get a direct “view” in to the test subjects thoughts and thus can see which parts of the test application cause problems to them.

2.7.5 Pilot testing

Pilot testing is often used to verify that the usability test plan really functions as intended. It usually finds at least some of the bugs in the test plan, which can be fixed once they are found (Rubin 1994, 95). After the deficiencies are fixed the real tests will run much smoother. Also user interface problems found during the pilot tests can be fixed before the real testing begins. There are no reasons to test for known problems.

3 Construction of the Application

The test applications will have functional user interfaces and the minimum necessary application logic so they can be tested. Applications are not meant to be realistic. So they will not have all the features usually associated with real applications.

3.1 Choosing the application

The application selected for the project should be one that can and would be used on different appliances. UIML is most likely to be used in an area where the need to use several different appliances is great.

The application should have users who would use it on a PC and on mobile appliances. Mobile appliances include handheld computers, PDA's and WAP cellular phones.

Banks offer services, which can be of great use to mobile users as well as home users. Paying bills and checking account events from home or on the move can be very useful. This fulfills our requirements for the test application.

3.2 Scenario

Two slightly different scenario sets are to be used as the basis for building of the two applications. Both have two scenarios. The first set is used with the PC application and the second set is used with the WAP application.

Scenarios are also used as usability test tasks. They are used to make the test person feel that he's inside the problem and at the same time the scenario gives him goals for the test.

3.2.1 PC

Scenario 1:

You are at home and there is an electric bill to be paid. The bills due date is tomorrow, so it has to be paid soon. You have a home computer and on it you have remote banking application and Internet connection. You decide to use them to pay your bill.

Scenario 2:

When you are putting the bill in the binder, you notice that the sum is much larger than in the previous electric bills. You call the electric company and they admit that there has been a computer error and they'll send a new fixed bill to you. Now you need to cancel the bill before the money leaves your account.

3.2.2 WAP

Scenario 1:

You are leaving for a holiday trip with your family. On your way to the car you check the mailbox, not the electronic one, one last time. You find a phone bill and take it with you. In the evening, after long drive, you take out your WAP capable cellular phone and use it to pay the bill.

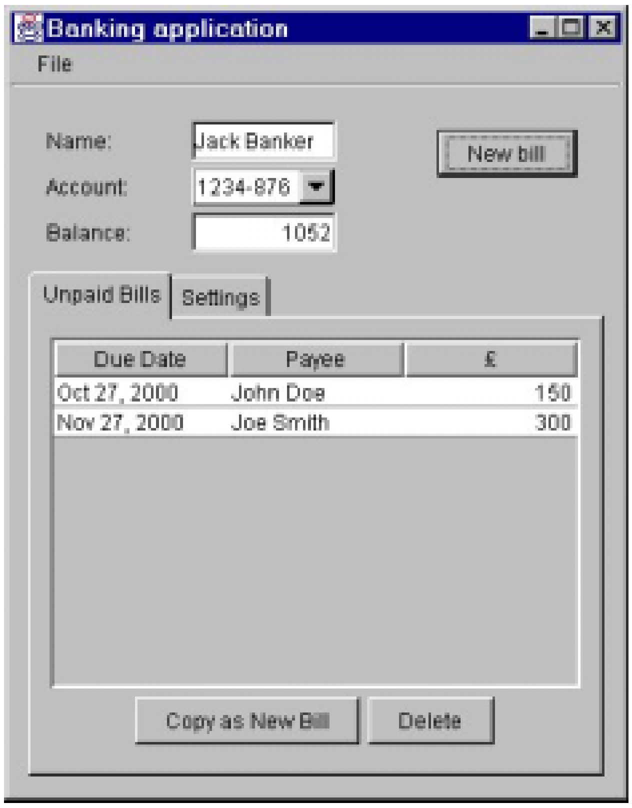
Scenario 2:

After giving the payment order via your WAP phone, you place the bill in the envelope for safekeeping until you get back to your house. While doing this you notice that the bill is not actually addressed for you but your neighbor. Postman must have made a mistake. You decide to cancel your payment and give the bill to your neighbor when you get back.

3.3 Building the application

3.3.1 Java application

The banking application has a main window that can be divided into two parts: user information and tab panel. User information part as shown in Figure 10 shows user's name, account number and account balance. There is also a button labeled "New bill" in user information part. This button opens a dialog for a new bill.



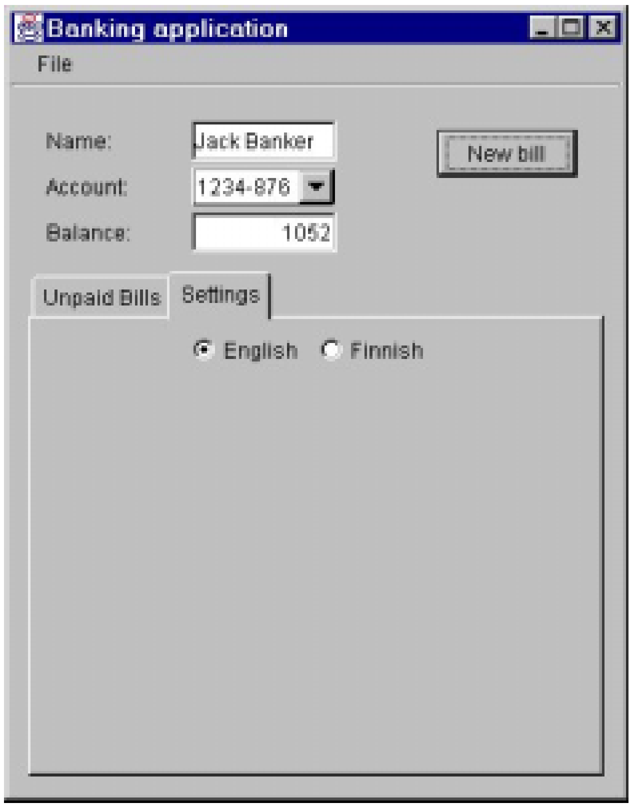
Name: Jack Banker
Account: 1234-876
Balance: 1052

Due Date	Payee	£
Oct 27, 2000	John Doe	150
Nov 27, 2000	Joe Smith	300

Figure 10. Banking application, unpaid bills.

The tab panel has two tabs. The first one shows the list of unpaid bills and it can be seen in the Figure 10. The list shows following information about the bills: Due date, Payee and amount. Bills can be chosen from the list and the chosen bill can be either deleted or copied as a new bill. These actions are launched from the two buttons at the bottom of the Figure 10.

The second tab is named "Settings" and it can be seen in Figure 11. In settings tab, there is only two radio buttons, which control the language used in the application. Possible choices are English and Finnish.



Name: Jack Banker
Account: 1234-876
Balance: 1052

[✓ English] Finnish

Figure 11. Banking application, settings tab.

There is a pull-down menu on the top of the application window and the pull-down menu is shown in Figure 12.

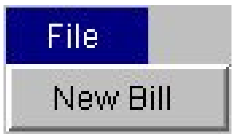


Figure 12. Banking application, pull-down menu.

When user wants to enter a new bill, he or she either presses the button labeled “New Bill” or chooses the “New Bill” action from the pull-down menu. This opens a new window, which is unsurprisingly called “New Bill”. This window can be seen in Figure 13.

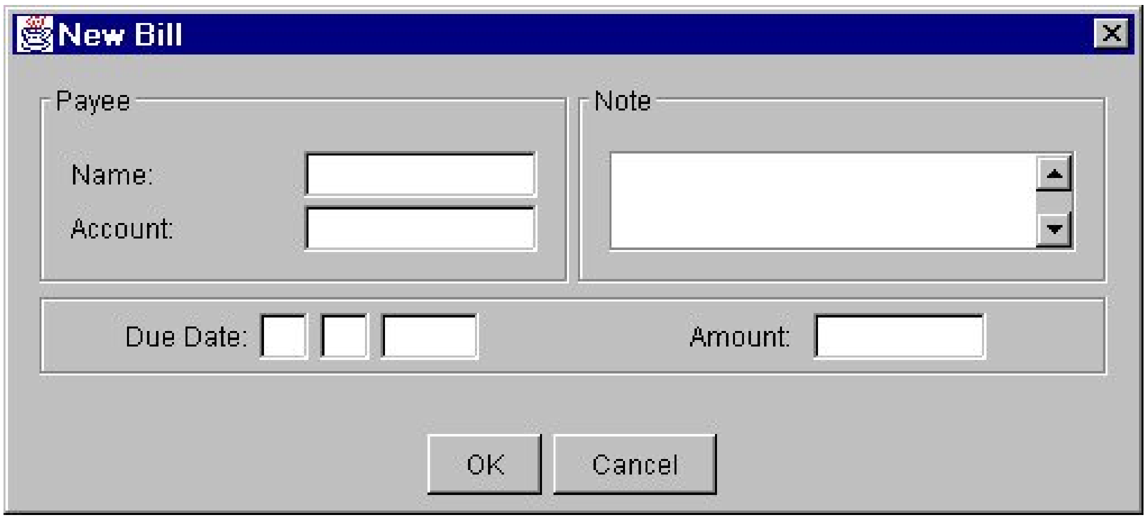
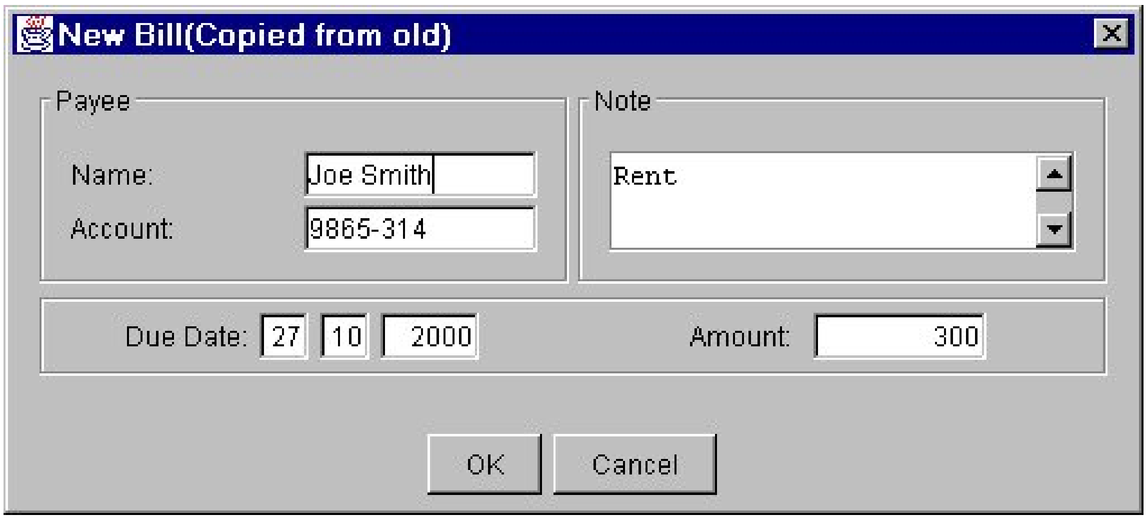


Figure 13. Banking application, new bill dialog.

If user presses "Copy as a New Bill" button, when a bill is highlighted on the list, a new window, which is shown in Figure 14, opens. The only differences from Figure 13 are, that the window title is different and that the input fields are filled.



Name: Joe Smith
Account: 9865-314
Rent
Due Date: 27 10 2000
Amount: 300

Figure 14. Banking application, new bill (copied from old) dialog.

The new bill dialogs have input fields for following information:

- Payee name
- Payee account
- Note
- Due date, separate fields for days, months and year

- Amount

The window has also two buttons, which are labeled OK and Cancel. As one can expect OK accepts the new bill and Cancel rejects it. Both buttons close the new bill window.

3.3.2 WML application

The WML application differs somewhat from the Java application. The main page or card shown in Figure 15 has three links:

- User information
- Unpaid bills
- New bill

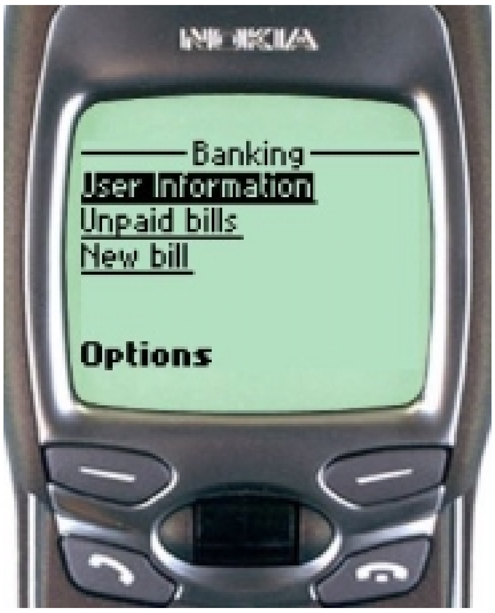


Figure 15. Banking application, main page

User information card shown in Figure 16 shows user's name, account and balance. Account can be selected, but it doesn't affect anything.

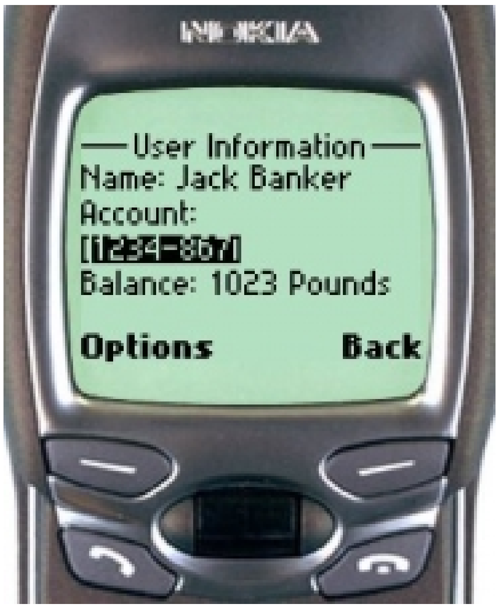
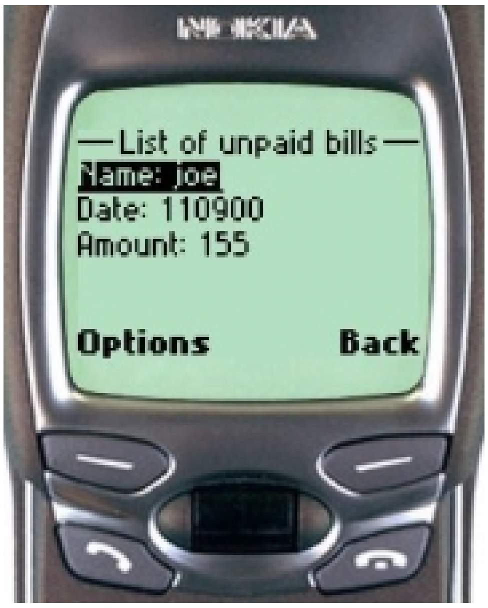


Figure 16. Banking application, user information

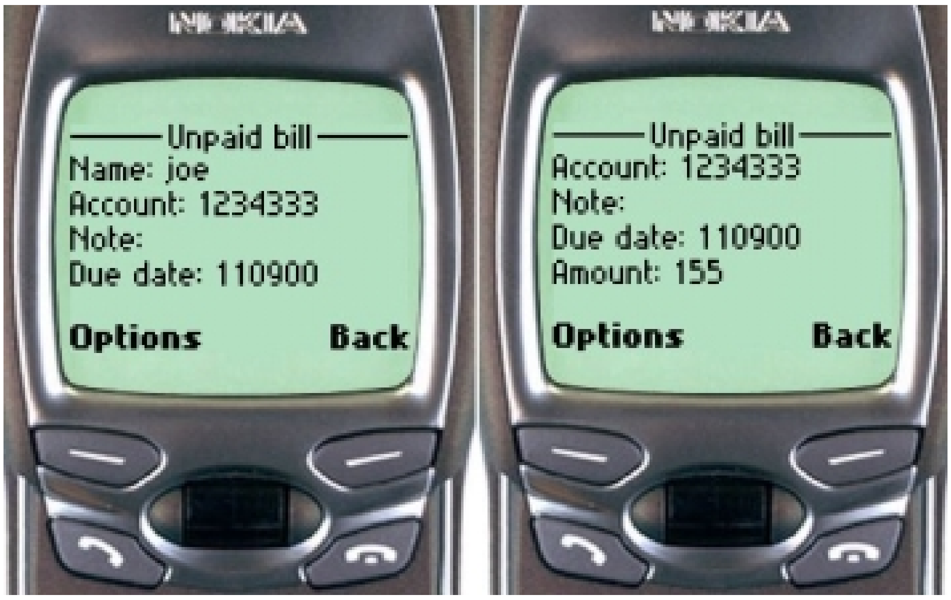
The list of unpaid bills is shown in Figure 17. The bill's information name contains a link to more specific information about the bill. The term list is a bit misleading for this card, since it only supports one bill. The reason for this is that if this were a real application, the WAP server would be responsible for creating the WML cards dynamically. Since the application backbone is not implemented, this approximation is adequate.


NOKIA

— List of unpaid bills —
Name: joe
Date: 110900
Amount: 155

Figure 17. Banking application, list of unpaid bills card.

Card unpaid bill shown in Figure 18 contains all the information about the chosen bill.


NOKIA

——— Unpaid bill ———
Name: joe
Account: 1234333
Note:
Due date: 110900

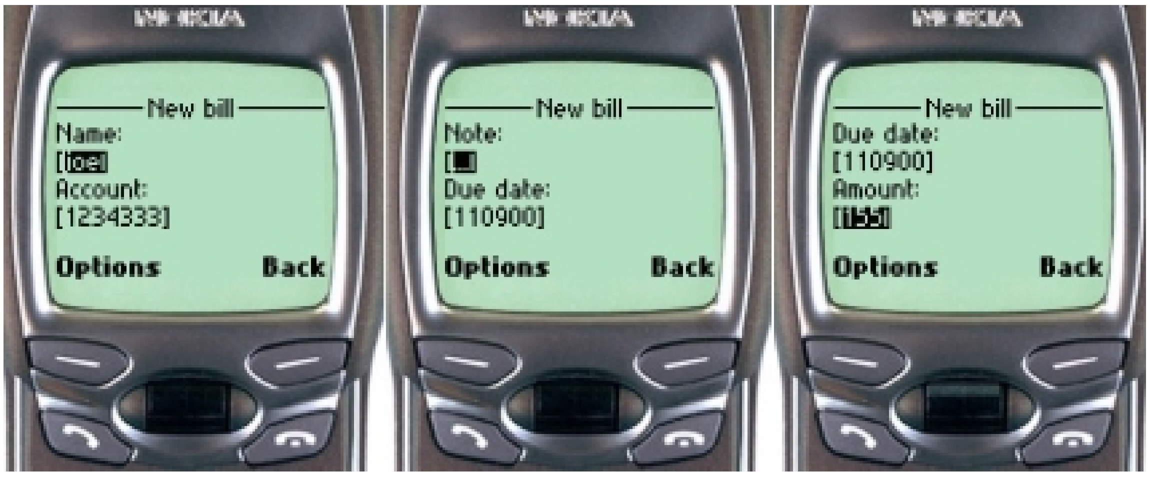
——— Unpaid bill ———
Account: 1234333
Note:
Due date: 110900
Amount: 155

Figure 18. Banking application, unpaid bill card.

New bill card shown in Figure 19 is used to enter new bill information. The card has input fields for following information:

- Payee name
- Payee account

- Note
- Due date
- Amount



New bill
Name:
[Joel]
Account:
[1234333]
Options Back

New bill
Note:
[]
Due date:
[110900]
Options Back

New bill
Due date:
[110900]
Amount:
[155]
Options Back

Figure 19. Banking application, new bill card.

3.3.3 UIML applications

UIML applications should look identical to those introduced in 3.3.1 and 3.3.2.

4 Evaluation of the Application

4.1 Evaluation criteria

For unambiguous results clearly stated success criteria is needed. Success criteria and methods for their verification are listed in the Table 2 below.

Table 2 Success criteria and methods of verification.

ID	Criteria	Importance	Method of verification
1	User interface components of the UIML user interface must be positioned according to the designer's plan	Mandatory	Compare the Java and the WML application to the UIML user interface. They should be very close to identical.
2	The usability of the UIML user interface should be as good as the usability of user interface created with appliance-dependent language. [a]	Desirable	Usability tests and interviews.
3	The UIML should not be significantly harder to use than appliance-dependent languages	Desirable	Expert opinion based on comparing the code needed for the user interfaces. "Not significantly harder" means approximately same amount of work

[a] Usability is a wide area and it is not fully covered by this thesis.

UIML and WML are both XML languages. Any XML language can be mapped into other XML language as long as they have all the relevant information. It may be possible that the WML code rendered from the UIML document is identical to the original WML code. In this case there is no need to run any tests on them. Comparing two identical user interfaces, which also have identical source code, is a waste of time.

4.2 Verifying criteria 1: Comparing the user interfaces

The user interfaces are compared by placing them side by side and examining them visually. All deviations are noted and their importance is evaluated. Any major deviancies, which can not be fixed, mean that the UIML won't pass the criteria 1 shown in Table 2.

4.3 Verifying criteria 2

4.3.1 Usability tests

The usability tests are used to evaluate the usability of the user interfaces. The goal is to compare the UIML user interfaces to the Java and WML based user interfaces. The input is not used to improve the user interfaces, but to evaluate the differences between the languages used in creating them.

The test subjects are chosen from the personnel of Ericsson Systems Expertise Ltd. in Athlone, Ireland. This is also the place where the tests are done. As tests are used in examining the UIML language and not the application itself, there is no need to get a good representation of the normal banking application users. Four test subjects are chosen from the experienced usability people.

The usability tests use the scenarios described in the subsection 3.2. All test subjects go through all the scenarios. Test subjects are not told which user interface they are using at the time of testing. Half of the test subjects go through the user interfaces in opposite order. This should prevent the learning effect from biasing the tests. After the usability tests, the test subjects are debriefed to find out their personal preferences and observations concerning the user interfaces. After the debriefing they are told which user interface was done with which user interface language. This is done to avoid any prejudices affecting the tests and debriefing.

The experimenter and the user interface designer are the same person. This creates some objectivity problems (Nielsen 1993, 180), which is unavoidable. No recording devices are used during the testing.

The usability tests are run on a following setup.

- Desktop PC
- Windows NT 4.0 operating system
- 500 MHz Intel Pentium 3 processor
- 128 Mb of main memory
- Java 2 SDK, Standard Edition Version 1.3.0
- Nokia WAP Toolkit 2.0
- Universal IT UIML Java renderer version 0.5d
- Universal IT UIML WML renderer version 0.3b

Nokia’s WAP toolkit is used to simulate a WAP phone on the PC.

Criteria 2 can be separated into two parts: functional usability and subjective usability. Functional usability is concerned with the correct use of a user interface and deviations from that. Subjective usability is concerned with the opinions of the test subjects and is consequently more concerned with the comfort of use of user interfaces.

UIML fails the functional part of the criteria 2 if the usability tests find that the UIML user interfaces do not function as specified. If there is failure and it can be fixed, then obviously the fault is in implementation and not with the UIML. The success of the subjective part is decided by the questionnaire administered in the debriefing phase.

Pilot testing will be used to test the whole usability test process from start to finish. That means that the pilot test subject will go through both the usability tests and debriefing. Only one pilot test subject is used. The pilot test subject is chosen from the same pool as the test subjects. The thinking - aloud method (Nielsen 1993, 195-198) is used during pilot testing. This helps to find out problems in the scenarios and applications that would cause unnecessary complications with the usability tests.

4.3.2 Debriefing

Each test subject is debriefed immediately after his or her usability tests are finished. This is to make sure that the tests are still fresh in the memory. Debriefing consists of a questionnaire and an interview.

The questionnaire is administered before any other discussion about the system. This helps avoid any biases caused by the comments of the experimenter (Nielsen 1993, 191). After the questionnaire is filled, there is an interview.

The questionnaire is used to compare the user interfaces created with UIML to those created with Java and WML. Semantic differentials are used for this, since they are scales on which the test subjects are asked to express the degree to which they favor one of two choices (Rubin 1994, 203). This questionnaire uses five point scales as shown in Table 3. Both ends of the scale read from 1-2 and 0 in the middle is a no-preference choice. This prevents any bias associated with higher versus lower numbers (Rubin 1994, 203).

Table 3. The scale used in the questionnaire.

Application 1	2	1	0	1	2	Application 2

Questionnaire has five questions on PC and WAP applications, which results in ten questions altogether. Both PC and WAP application parts have the same questions. The questions are:

1. Which one looks better?
2. Which one responds faster?
3. Which one has faster screen updates?
4. Which one is easier to use?
5. I prefer?

PC and WAP applications are separate cases when considering the criteria 2. No question should get mean value of more than 1 in favor of non-UIML user interfaces.

Also, the mean value of all five of the questions should not exceed 0,75 in favor of non-UIML user interfaces. If either one or both of these conditions fail, then UIML fails the subjective part of criteria 2.

Test subjects are also asked for the reasons behind their answers in the questionnaire. This may reveal consistent complaints about some aspects of either one of the user interfaces and thus help out in finding weaknesses about them.

During the interview the test subjects are asked for any comments they might have about the applications. The aim is to find what differences the test subjects noticed and which they thought affected the usability of the application. Attention is also given to the subjective seriousness of the usability problems.

After the debriefing is finished, the questionnaires and the experimenter's notes will be labeled with the test subject's number. Also, a short test report shall be written if necessary.

4.4 Verifying criteria 3: Expert opinions

The time spent on creating the user interfaces is not a good measuring unit to find out which interface is harder to use for following reasons:

- When creating the Java and WML interface all user interface components and their variables were defined. These same variables are used with UIML. So, the time needed with the UIML is not comparable, because there is less work to be done with it.
- UIML is a new language and therefore UIML programming skills are not as high as with already existing languages. Consequently a lot of time is spent on studying the UIML language and the studying is intermixed with user interface coding. This means that it is very difficult to estimate the time needed to code the user interfaces.

Because of these reasons some other measurement unit should be used to measure how hard the UIML is to use. An expert opinion based on observing the following points is used to evaluate the criteria 3:

- Amount of code needed for UIML based application versus Java and WML applications.
- Amount of appliance-independent code in UIML documents.
- Amount of appliance-dependent code needed to support UIML.
- How easily appliance-dependent code works with UIML.
- How difficult it is to implement an UIML application versus Java and WML applications. This includes an estimate of UIML learnability.
- Complexity of UIML document versus Java and WML code.

5 Results of the Study

5.1 Pilot test

The pilot test showed that the way the application works is not as clear as it should be. Subject thought that the bill is paid immediately after it is entered. In other words, the idea that the bill is on the list of unpaid bills and is paid when the due date arrives was not intuitive for the test subject. To avoid more problems of this type the idea will be explained to the test subjects. This does not disqualify the results from the usability tests, because it is the user interface language that is being tested, not the application logic.

No other problems were found during the pilot test.

5.2 Criteria 1

The WML renderer is not complete at the time of testing and the current version supports only part of the WML tags and UIML properties. When the supported tags were compiled, the resulting WML file is exactly the same as the target WML file minus the unsupported tags. Based on this, an assumption can be made that the unsupported tags would also be compiled correctly on later renderer versions. If the renderer supported all WML tags, the UIML-WAP and WML-WAP user interfaces would be identical. Therefore, usability testing and comparing the WAP user interfaces is unnecessary.

Although the Java renderer is not complete at the time of testing, the UIML user interface is identical with the Java user interface. Screenshots from the Java application can be found in section 3.3.1 and screenshots from the UIML PC application are in appendix 1.

5.3 Criteria 2

WAP applications were not tested for the reason given in the first paragraph of section 5.2.

The usability test did not find any functional problems specific to the UIML application. One functional problem was found outside of the tests. It concerns the table and row-selection. To use the “Copy as a new” feature in the UIML application one must select the wanted row twice, when using it the first time. After the first time everything works correctly. Also, either deleting a bill or changing the user interface language eliminates this problem. This problem is not evident in the Java application, which uses this table in the same way as the UIML application. Most likely this problem originates from the way the renderer handles Java events. This may change with the future releases of the Java renderer.

Figure 20 shows the questionnaire answers and average values for each question. Only one test subject favored one of the test applications over the other. The test subject favored Java application slightly on questions 2, 3 and 5. The test subject wrote in the questionnaire that it was likely that the preference was caused by learning effect making the second test application faster to use. Otherwise the test subjects noticed no differences between the test applications.

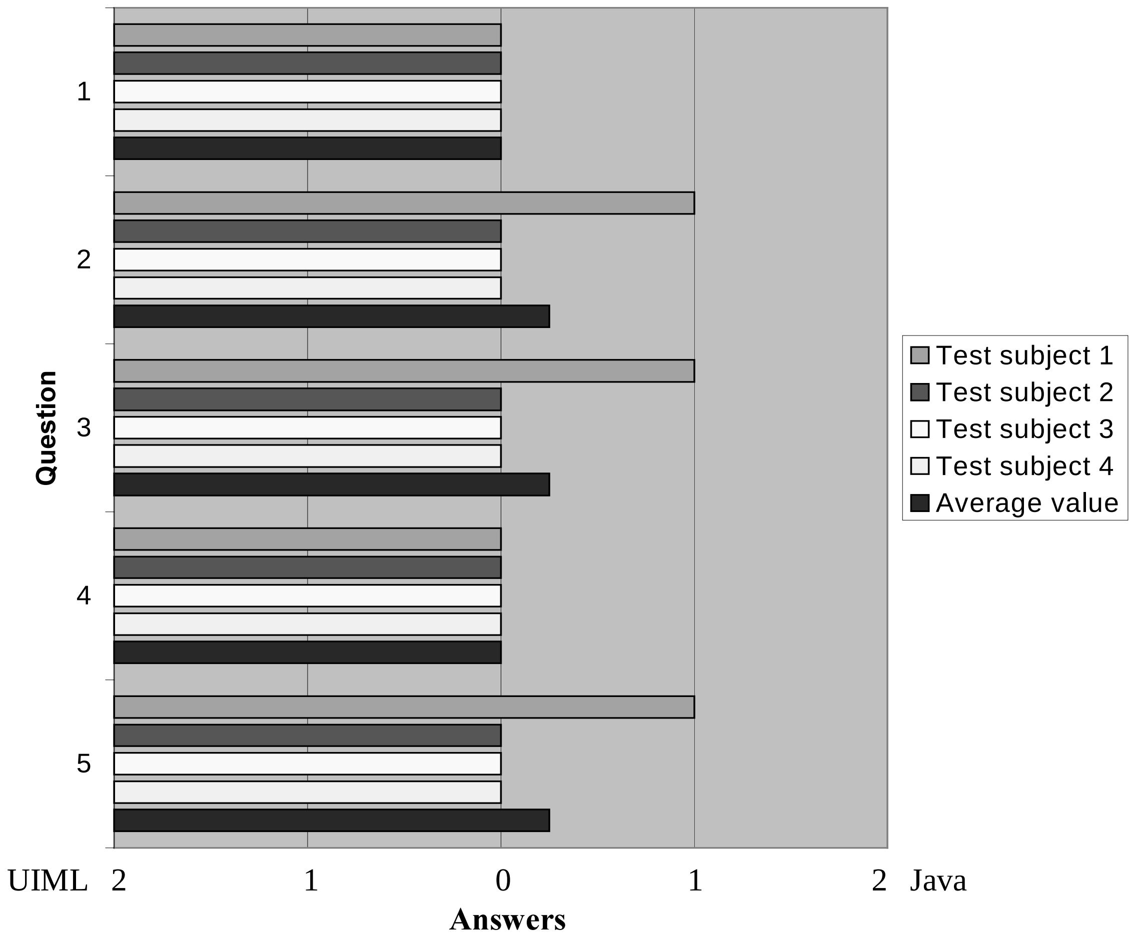


Figure 20. Questionnaire results and average values for results

The questionnaires found out all that was intended and the interviews did not reveal anything new. This was because the user interfaces are identical from the users' perspective.

5.4 Criteria 3

The amount of lines of code needed for the WAP user interfaces is shown in Figure 21. WML needs only 82 lines of code, where UIML needs 191, which does not even contain all the functionality of the WML user interface. If all functionality were included, it would increase the amount of lines to over 200. There are more than two times as many lines in the UIML document as in the WML file.

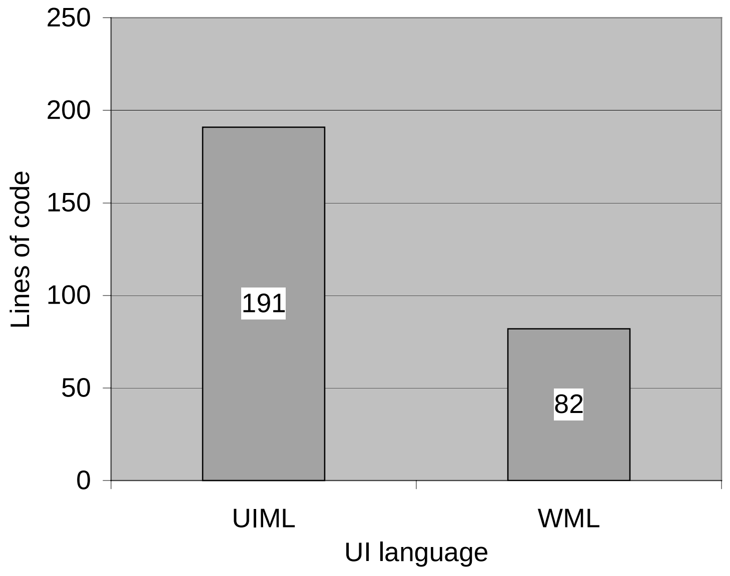


Figure 21. Lines of code for WAP application. Comments are included.

Figure 22 shows the amount and the distribution of lines of code for the PC applications. The application backbone is the same for both of the applications. As can be seen, the UIML application has 186 lines of code more than the Java application. It means that the UIML application is 24% larger than the Java application.

32% of the UIML application's actual user interface code is Java. This Java code consists of necessary methods for the UIML document to work with the application backbone and of code to use some user interface components and actions, which were not supported by UIML. These mainly consist of methods used in creating and updating border objects and the table.

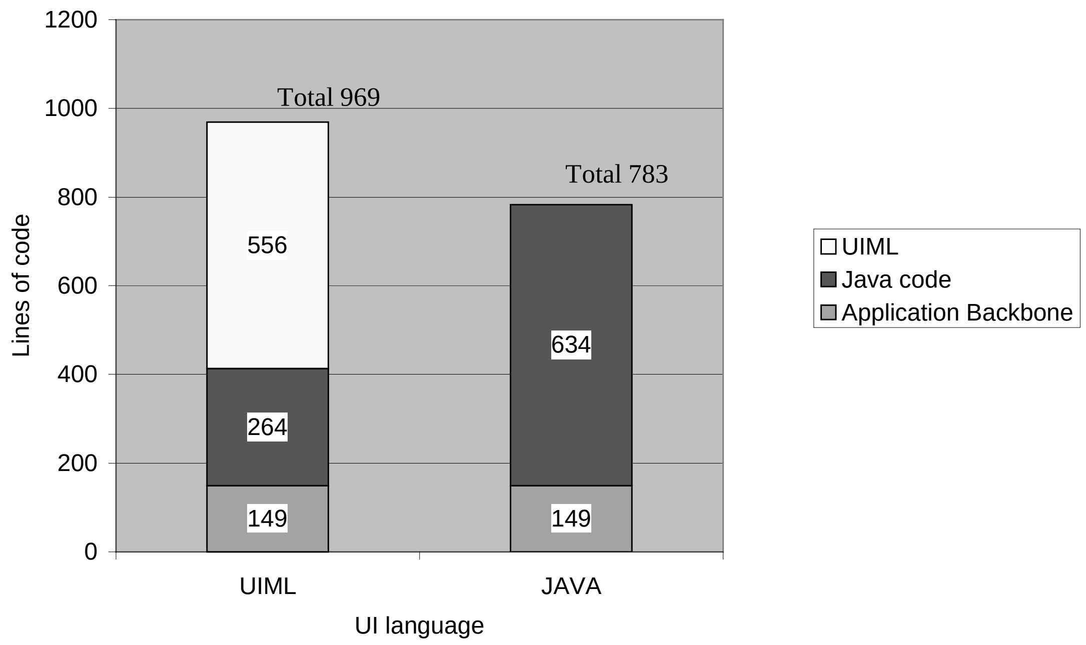


Figure 22. Lines of code for PC application. Comments are included.

The following example gives a good idea of how complex it is to use UIML in WAP user interface creation. It is an extreme case, but it demonstrates the complexity very well. These examples are from the test applications and rest of the UIML documents can be found in the appendix 2. First is the WML code:

```
<card id="new" title="New bill">
    <p>
        Payee: <input name="name"/>
        Payee account: <input name="account"/>
        Note: <input name="note"/>
        Due date:<input name="date"/>
        Amount: <input name="amount"/>
    </p>
</card>
```

The picture of the user interface that this piece of code creates can be seen in Figure 19 on page 38. The parts from UIML document that produce exactly the same WML code look like this:

```
<part name="new" class="Card">
    <part name="prev2" class="Do"/>
    <part name="newpara" class="P">
        <part name="new1" class="RichText"/>
        <part name="namefield" class="Input"/>
        <part name="new2" class="RichText"/>
        <part name="accountfield" class="Input"/>
        <part name="new3" class="RichText"/>
        <part name="notefield" class="Input"/>
        <part name="new4" class="RichText"/>
        <part name="datefield" class="Input"/>
        <part name="new5" class="RichText"/>
        <part name="amountfield" class="Input"/>
    </part>
</part>
.
.
.
<style>
    <property part-name="new" name="title">New bill</property>
    <property part-name="new1" name="content">Name:</property>
    <property part-name="new2" name="content">Account:</property>
    <property part-name="new3" name="content">Note:</property>
    <property part-name="new4" name="content">Date:</property>
    <property part-name="new5" name="content">Amount:</property>
    <property part-name="namefield" name="name">name</property>
    <property part-name="accountfield" name="name">account</property>
    <property part-name="notefield" name="name">note</property>
    <property part-name="datefield" name="name">date</property>
    <property part-name="amountfield" name="name">amount</property>
</style>
.
.
.
<rule>
    <condition>
         <event class="prev" part-name="new" name="do"/>
    </condition>
    <action>
        <property part-name="prev" name="visible">true</property>
    </action>
</rule>
```

This example does not include the definitions for the WML vocabulary, since the renderer provides it. This complexity comes from the fact that when using UIML each value and attribute needs to be set separately. In most parts the complexity of this example comes from the input elements that are inside the text paragraph. UIML provides no way to place an element inside an other element’s value. The solution is to break the latter element into two pieces and insert the former element between them.

Using Java Swing toolkit with UIML is more straightforward than using UIML to create WML code, since both Java Swing and UIML separate the structure and the content from each other. The Java code is in appendix 3. When comparing the Java code and the UIML document used in creating a user interface, there are many similarities. Swing uses a hierarchical way in user interface element placement, same as UIML. Swing uses methods to set element properties and content. UIML uses the property and content elements. These similarities lead to that working with UIML and Swing toolkit is similar to working with only Java Swing.

How well does the appliance-dependent code work with UIML? This is a renderer specific question, but the renderer proves that UIML can and does work well with Java. For example, the renderer does not support Java borders, but it was possible to add borders with Java code to the UIML user interface. Also, it proved to be possible to send and receive data between UIML user interface and Java program. A real problem with UIML and Java cooperation was found when using the table element. This problem is described in section 5.3 in greater detail. Accessing some Swing properties and methods from UIML is a bit different from what one would expect from Java coding experience. For example, when using Java some Swing properties have to be set by using method calls, but when using UIML they are set using attributes. Other small differences can be found between Java and UIML.

The amount of appliance-independent code in UIML could not be verified experimentally, since at the time of the testing the current version of the renderer did not support all of the necessary properties of UIML. Working with UIML brought to light some pros and cons concerning the appliance-independent code. These are discussed in the following paragraphs.

In section 1.1 it was claimed that only one common style sheet is needed per appliance for all applications. This claim is not achieved, since almost every user interface component needs it’s own settings in all applications. UIML has a concept called a vocabulary, which is used to map UIML code into a specific toolkit. It can be said that one appliance needs one style sheet, if the style sheet is the vocabulary for the appliance. But they still need style elements for describing the user interface elements. Creating a vocabulary is extra work that has to be done once for each appliance.

When comparing UIML documents for the PC and the WAP phone it is clear that they have structurally little in common, which may or may not be the result of poor design. But when one compares the complexities of both of the UIML documents, it is clear that the PC document’s structure is much more complex. This large difference in the complexities indicates that they would be structurally hard to match.

The contents of user interface elements are mostly appliance-independent. If the UIML document is used in creating user interfaces on multiple appliances, then there is some reduction in the amount of code needed. This is entirely dependent on the application. More importantly, the user interfaces will have consistent user interface element content across appliances, which is good news from the usability point of view.

6 Conclusions And Discussion

6.1 Criteria results

6.1.1 Criteria 1

The test applications show clearly that the UIML user interfaces are visually equal to the user interfaces created with Java and WML. Although we only compare user interfaces created with UIML with user interfaces created with two other languages, it is quite clear that user interfaces, regardless of the toolkit used, can be created with the UIML as well as with platform specific languages. Consequently criteria 1 is passed.

6.1.2 Criteria 2

The questionnaires show that the test subjects slightly favor the Java user interface, but the probability of this being caused by the learning effect is quite high. Three of the test subjects did not notice any differences between the applications and answered with the no-preference choice to all of the questions. One of the four test subjects favored the Java application on questions 2, 3 and 5. The test subject suspected that it just felt faster because the user interface was familiar when one performed the tasks on the second application. Questions 2 and 3 were about the speed of the user interface and favoring the later test application supports the learning effect theory. Question 5 asked which one of the user interfaces the test subject preferred. Clearly, if the test subject perceives that one of the applications is faster than the other, then he or she prefers the faster one and thus the answer to question 5 does not contradict the learning effect theory.

Since the majority of the test subjects favored neither of the user interfaces and the validity of the sole differing opinion is uncertain. It can be said that the UIML user interface, at least in the users' mind, is as good as the Java user interface. Usability tests showed no functional problems with the user interfaces. When all this is taken into consideration it can be said that the usability of a user interface does not suffer from using UIML. This means that criteria 2 is passed.

6.1.3 Criteria 3

There is little doubt in the fact that UIML needs more lines to accomplish the desired outcome than appliance-dependent languages in general. This is clearly shown in Figure 21 and Figure 22. The actual amount depends greatly on the toolkit used. In this study the UIML documents' sizes varied from 124% to 230% of the corresponding appliance-dependent application's size. These numbers are only indicative of the situation in general and they are statistically speaking invalid, since the numbers are gathered from one pair of user interfaces on two appliances.

XML languages differ greatly from functional programming languages. This makes comparing the sizes of an XML document and non-XML document difficult or even meaningless. For this reason the lines of code needed with UIML and Java applications cannot be compared easily. One has to take into account the differences between the languages. So the lower limit of 124%, which was with the Java application, does not tell us as much as the higher limit. The higher limit of 230% comes from when UIML was compiled into WML. As both, WML and UIML, are XML languages the numbers can be compared directly.

The PC applications are roughly the same size and the effort needed to create them was slightly in favor of Java application. It takes a bit more effort to work with UIML-Java combination than working with pure Java code. There is always more to know when using two languages instead of just one.

The amount of Java code needed to support the UIML application was 32% of the UIML PC applications user interface code. This does not include the application backbone. Once the renderer is complete, approximately one third of this Java code could be done with UIML. This would reduce the percentage to approximately 20%, which is quite acceptable, at least in this case. Here we made the assumption that one line in the UIML document correlates to one line of Java code. A more realistic ratio would be slightly in favor of Java as indicated by the amount of code in Java and

UIML applications. Figures available are not statistically valid and the UIML application uses some Java code, which makes it hard to find a solid Java-UIML lines of code ratio. This makes the one-to-one ratio assumption as good as any other ratio we can calculate on the basis of the available data.

The UIML WAP application is more than double the size of the WML WAP application. This translates directly into the effort needed to code the user interface. But since coding is only a part of the process and it can be facilitated with proper tools, this means that the total effort is somewhat less than the ratio calculated from the lines of code indicates. It is impossible to say exactly how much less, since we could not use time to measure the effort, and the effect of tools is not clear.

The structure of UIML document makes it complicated to use toolkits that can place a user interface component within other components' content. These toolkits or languages include, for example, HTML and WML. This does not mean that UIML cannot be used with them. It simply means that lots of lines are needed for something that can be done with far fewer lines if programmed with the target language. Good examples of this are the WAP applications created during this study.

The test applications were too different to be able to use a single UIML document for both user interfaces. It seems that UIML could be used in multiple appliance environment if the appliances are similar or at least not very different. For example, WAP phones could have different user interfaces for different phones. It should be studied in greater detail which appliances and what type of applications can benefit from multiple appliance UIML user interfaces.

Learning the UIML proved to be quite easy. At least for someone who has previous experience with functional languages and who is familiar with the concepts of XML beforehand. One does not need to learn as much as with the functional languages. This is based on subjective opinion and as such should be taken with a grain of salt. It should be studied how people with different backgrounds learn to use UIML.

When all this is taken into account it is clear that in general UIML is a little harder to use than appliance-dependent languages, but not significantly. This means that criteria 3 is passed.

6.2 Other observations

Non-programmers can use UIML, but knowledge of the user interface toolkit is very important for user interface developers. UIML only maps into the toolkit and does not hide the toolkit's user interface components' properties from the developer. So the developer cannot use toolkits effectively without first understanding them.

UIML does not limit what the user interface designers can do. On the other hand, the renderers limit what parts of toolkits are available for the designer. This means that the renderer is the first, and probably also the biggest, bottleneck for the UIML user interface development for new devices. Older devices should have third-party renderers available.

The only functional problem that was found in the applications was the modified JTable in the PC UIML application. This problem may go away with future releases of the Java renderer, but something similar might come up later. So it would be important to study customized user interface gadgets and how they work with UIML in greater detail.

The test applications showed no degraded performance due to the use of the UIML renderer. Larger and more resource-intensive applications may suffer from degraded performance because of the overhead of using the renderer in interpreting a UIML document into user interface. It should be studied how much overhead the renderer causes.

Harmonia Inc. has released a shorthand notation for UIML, which reduces the amount of elements needed for defining the properties of a user interface component. This makes the hand coding of UIML documents a more attractive option than the results of this thesis suggest.

6.3 Reliability Analysis

Visually comparing two graphical user interfaces is by its very nature an error free operation. As this study did not cover all possible user interface components, some user interface components may not be as presentable with UIML as with native toolkits. So the results are nearly 100% reliable.

Usability tests were executed in a way that is standard throughout the usability science. Very seldom are there enough test subjects in usability tests for statistically valid results. The extra reliability gained from having a large number of test subjects simply is not worth the time. The reliability gained from the limited amount of test subjects is adequate for usability purposes method (Nielsen 1993, 173-174). This means that the reliability is high enough to satisfy the needs of this study.

The results from the questionnaire are not statistically valid and cannot be regarded as completely reliable. The results give indication of what the situation really is. Since the variance of the answers is very small, the results can be used as a good estimate of the real situation.

The expert opinions are always opinions and thus can vary between different experts, even by a large margin. Also, in this study the test applications were evaluated and made by the same person. This may have caused a defensive attitude in the evaluation of the applications. But knowing this beforehand and the fact that the applications were created for the sole purpose of testing UIML and not to be used in any real world situation reduces the effects of the defensive attitude to insignificant levels.

Also, the fact that the expert opinions are from only one researcher reduces the validity by a small margin. Multiple opinions would have given more reliable results.

The opinions are based on measurable quantities wherever possible. The rationale for each opinion is clearly stated. This does not increase the reliability by itself but it gives the reader a way to see how the opinions were formed and thus each reader is able to check easily if they agree with the results.

The expert opinions given here cannot be viewed as the only truth, but they are valid from the author’s point of view. And since the paths to the conclusions are clearly stated, each and everyone can evaluate the reliability for themselves.

The articles and books used in this study are recent due the nature of the topic. The literature used is selected from respected professionals. The source information should represent reliable information, although the papers on UIML are from the authors of UIML.

6.4 Validity of the study

The objectives for this study were:

To find out UIML's limitations and problems in a multi-appliance environment.

In a way this objective was not fully achieved within this study, since it would have needed more appliances and test applications than were used in this study or even could have been used in optimal circumstances. But a complete coverage was not an objective for this study. This study found some problems for and limitations on UIML in a multi-appliance environment and thus completed this objective.

To find out UIML's limitations and problems in user interface creation.

To evaluate the usability potential of UIML from the developers' perspective.

These objectives were completed quite thoroughly. Much of the data that was collected during this study was used in comparing UIML to other languages that can be used in creating user interfaces. This means that the data and the testing that gave the data directly contributed to the completion of these two objectives.

7 Summary

UIML can be used in user interface creation almost as well as any other native toolkits. Almost, because it always needs a renderer and if the renderer does not allow a functionality that the target toolkit can do, then the UIML simply cannot perform these functions. If the renderer is up to the job, then the UIML user interface can be as good as the one directly created with the target toolkit.

The Java renderer used in this study proves that the UIML can work well with the target toolkits. Programming languages can be used to enhance the user interfaces created with UIML, as long as the renderer allows it.

UIML applications need more lines of code to accomplish the same things as with the native toolkits. The amount of lines needed is highly dependent on the target toolkit. When using Java toolkit with UIML, the UIML application needed 24% more lines than the pure Java application did, even though 32% of the UIML-Java application was Java code, not UIML. When using UIML with WML toolkit the application needed more than two times more lines. This is not an exact figure because not everything could be tested, as at the time of testing the renderer was not fully functional (pre-release version).

If the structures of the user interface components in the target toolkits are too different, then it can be very difficult to use one UIML document with multiple "style sheets" in creating the user interfaces. This means that it is possible to create multiple-appliance user interfaces if the target toolkits are similar enough. For example, Java and WML applications proved to be too hard to match structurally during this study.

8 References

Abrams, M., Phanouriou, C., Batongbacal, A., Williams, S. & Shuster, J. (1999): UIML: An Appliance-Independent XML User Interface Language. Paper presented at 8th International World Wide Web Conference (WWW8). Toronto, Canada. May 1999. http://www8.org/w8-papers/5b-hypertext-media/uiml/uiml.html. Also appeared in Computer Networks, Vol. 31, 1999, pp. 1695-1708.

Abrams, M., Phanouriou, C. (1999): UIML: An XML Language for Building Device-Independent User Interfaces. Paper presented at XML'99 conference. Philadelphia, USA. December 1999. Also appeared in XML '99 Conference Proceedings, available from GCA at http://www.gca.org/

Nielsen, J. (1993): Usability Engineering. Morgan Kaufmann Publishers, Inc. San Francisco, USA. ISBN 0-12-518406-9.

Preece, J., Rogers, Y., Sharp, H., Benyon, D., Holland, S. & Carey, T. (1994): Human-Computer Interaction. Addison-Wesley Publishing Company. Wokingham, England. ISBN 0-201-62769-8.

Rubin, J. (1994): Handbook of usability testing: how to plan, design, and conduct effective test. John Wiley & Sons, Inc. New York, USA. ISBN 0-471-59403-2.

Universal Interface Technologies (2000): White Paper: The UIML Vision. Available at http://www.universalit.com/uiml/whitepapers/index.htm

World Wide Web Consortium (1998): Cascading Style Sheets, level 2. W3C recommendation, available at http://www.w3.org/TR/REC-CSS2/.

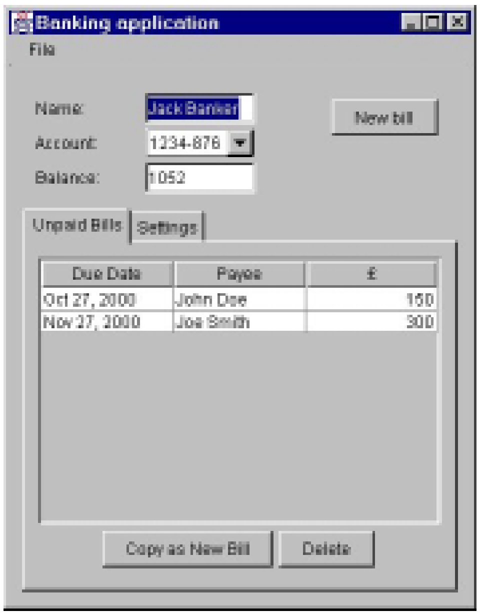
World Wide Web Consortium (1999): Extensible Markup Language (XML) 1.0. W3C recommendation, available at http://www.w3.org/TR/REC-xml.

World Wide Web Consortium (2000): Extensible Stylesheet Language (XSL) Version 1.0. W3C Working Draft, available at http://www.w3.org/TR/xsl/.

WAP Forum 2000. Wireless Application Protocol Wireless Markup Language Specification version 1.3. Available at http://www1.wapforum.org/tech/documents/WAP-191-WML-20000219-a.pdf

Appendixes

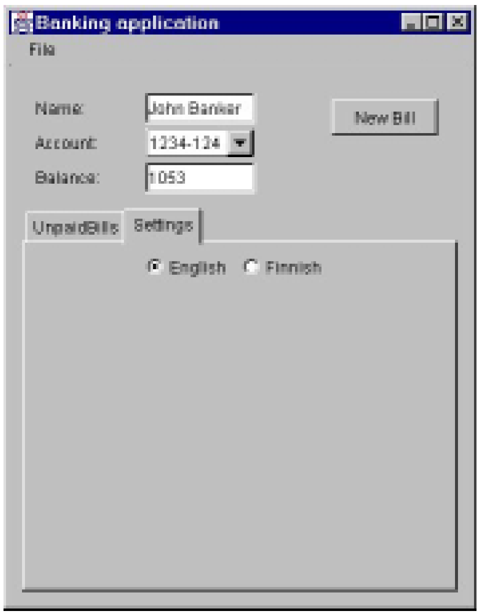
Appendix 1: UIML PC application screenshots



Name: Jack Banker
Account: 1234-876
Balance: 1052

Due Date	Payee	£
Oct 27, 2000	John Doe	150
Nov 27, 2000	Joe Smith	300

Figure 23 Unpaid bills panel, banking application



Name: John Banker
Account: 1234-134
Balance: 1053

[✓ English] Finnish

Figure 24 Settings panel, banking application

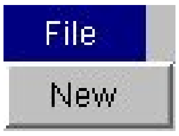


Figure 25 File menu, banking application

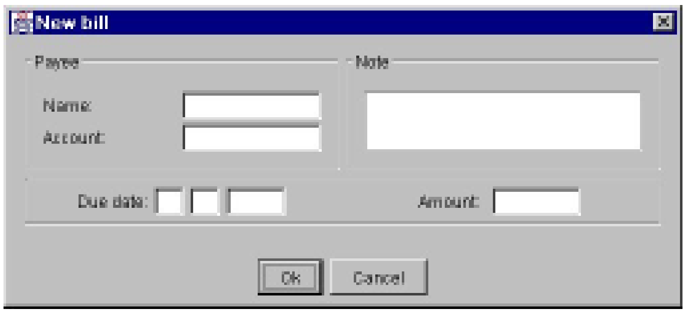
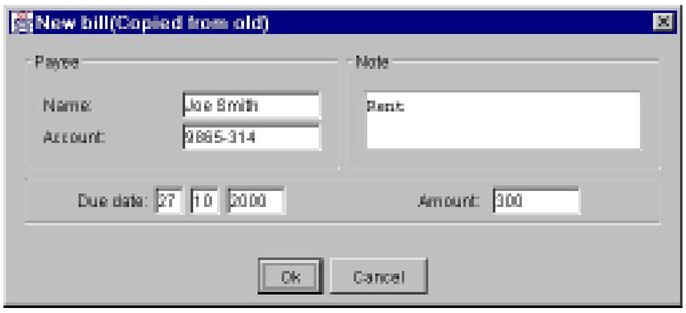


Figure 26 New bill dialog, banking application


New bill(Copied from old)

Payee

Name: Joe Smith

Account: 9865-314

Note: Rent

Due date: 27 10 2000

Amount: 300

Figure 27 New bill (copied from old), banking application

Appendix 2: UIML documents

PC application

```
<?xml version="1.0" encoding="ISO-8859-1"?><!DOCTYPE uiml PUBLIC "-//UIT//DTD UIML 2.0 Draft//EN"
"C:\UIML2\bin\UIML2_0d.dtd"><uiml>    <interface>
        <structure>
            <!-- Banking application's main frame -->
            <part name="Banking" class="JFrame">
                <style>
                    <property name="title">Banking application</property>
                    <property name="visible">false</property>
                </style>
                <part name="contentPanel" class="JPanel">
                    <style>
                        <property name="layout">java.awt.BorderLayout</property>
                    </style>
                    <!-- Menubar and it's contents -->
                    <part name="menubar" class="JMenuBar">
                        <style>
                            <property name="borderAlignment">North</property>
                        </style>
                        <part name="menu" class="JMenu">
                            <style>
                                <property name="text">File</property>
                            </style>
                            <part name="menuItem" class="JMenuItem">
                                <style>
                                    <property name="text">New</property>
                                </style>
                            </part>
                        </part>
                    </part>
                    <!-- Menubar ends -->
                    <part name="extraPanel" class="JPanel">
                        <style>
                            <property name="borderAlignment">Center</property>
                            <property name="layout">java.awt.BorderLayout</property>
                        </style>
                        <part name="topPanel" class="JPanel">
                            <style>
                                <property name="borderAlignment">North</property>
                                <property name="layout">java.awt.BorderLayout</property>
                            </style>
                            <part name="userInformationPanel" class="JPanel">
                                <style>
                                    <property name="borderAlignment">West</property>
                                    <property name="layout">java.awt.GridLayout</property>
                                </style>
                                <!-- labelPanel contains the labels for inputfields -->
                                <part name="labelPanel" class="JPanel">
                                    <style>
                                        <property name="layout">java.awt.GridLayout</property>
                                        <property name="layout_rows">3</property>
                                        <property name="layout_vgap">3</property>
                                    </style>
                                    <part name="nameLabel" class="JLabel">
                                        <style>
                                            <property name="text">Name:</property>
                                        </style>
                                    </part>
                                    <part name="accountLabel" class="JLabel">
                                        <style>
                                            <property name="text">Account:</property>
                                        </style>
                                    </part>
                                    <part name="balanceLabel" class="JLabel">
                                        <style>
                        <property name="text">Balance:</property>
                    </style>
                </part>
            </part>
            <!-- labelPanel ends -->
            <!-- fieldPanel contains three text fields -->
            <part name="fieldPanel" class="JPanel">
                <style>
                    <property name="layout">java.awt.GridLayout</property>
                    <property name="layout_rows">3</property>
                    <property name="layout_vgap">3</property>
                </style>
                <part name="nameField" class="JTextField">
                    <style>
                        <property name="text">John Banker</property>
                        <property name="editable">false</property>
                    </style>
                </part>
                <part name="accountField" class="JComboBox">
                    <style>
                        <property name="content">
                            <constant model="list">
                                <constant name="a" value="1234-124"/>
                                <constant name="b" value="1234-876"/>
                            </constant>
                        </property>
                        <property name="selectedIndex">0</property>
                    </style>
                </part>
                <part name="balanceField" class="JTextField">
                    <style>
                        <property name="text">1053</property>
                        <property name="editable">false</property>
                    </style>
                </part>
            </part>
            <!-- fieldPanel ends -->
        </part>
        <part name="newButtonPanel" class="JPanel">
            <style>
                <property name="borderAlignment">East</property>
            </style>
            <part name="newButton" class="JButton">
                <style>
                    <property name="text">New Bill</property>
                </style>
            </part>
        </part>
    </part>
    <!-- Tabpane has two panels -->
    <part name="tabPane" class="JTabbedPane">
        <!-- first tab panel -->
        <part name="tabPanel1" class="JPanel">
            <style>
                <property name="name">UnpaidBills</property>
                <property name="layout">java.awt.BorderLayout</property>
            </style>
            <!-- scrollPanel is scrollable panel which in this case has a JTable as a child -->
            <part name="scrollPanel" class="JScrollPane">
                <style>
                    <property name="borderAlignment">Center</property>
                    <property name="preferredSize">300,200</property>
                </style>
                <part name="table" class="JTable"/>
            </part>
            <!-- scrollPanel ends -->
            <!-- buttonPanel has two buttons -->
            <part name="buttonPanel" class="JPanel">
                <style>
                    <property name="borderAlignment">South</property>
                </style>
                <part class="JButton" name="copyasnewButton">
                    <style>
```

```
                            <property name="text">Copy As New</property>
                        </style>
                    </part>
                    <part class="JButton" name="deleteButton">
                        <style>
                            <property name="text">Delete</property>
                        </style>
                    </part>
                </part>
                <!-- buttonPanel ends -->
            </part>
            <!-- First tab panel ends -->
            <!-- Second tab panel starts -->
            <part name="tabPanel2" class="JPanel">
                <style>
                    <property name="name">Settings</property>
                </style>
                <!-- Radio button group -->
                <part class="ButtonGroup" name="buttongroup">
                    <part class="JRadioButton" name="English">
                        <style>
                            <property name="text">English</property>
                            <property name="selected">true</property>
                        </style>
                    </part>
                    <part class="JRadioButton" name="Finnish">
                        <style>
                            <property name="text">Finnish</property>
                        </style>
                    </part>
                </part>
                <!-- Radio button group ends -->
            </part>
            <!-- Second tab panel ends -->
        </part>
        <!-- Tabpane ends -->
    </part>
</part>
<!-- Banking application's invoice dialog -->
<part name="invoiceDialog" class="JDialog">
    <style>
        <property name="title">New bill</property>
        <property name="size">508,227</property>
        <property name="visible">false</property>
        <property name="modal">true</property>
    </style>
    <part name="basePanel" class="JPanel">
        <style>
            <property name="layout">java.awt.BorderLayout</property>
        </style>
        <!-- buttonPanel2 has two buttons: ok and cancel, which appear at
        the bottom of the GUI -->
        <part name="buttonPanel2" class="JPanel">
            <style>
                <property name="borderAlignment">South</property>
            </style>
            <part name="ok" class="JButton">
                <style>
                    <property name="text">Ok</property>
                </style>
            </part>
            <part name="cancel" class="JButton">
                <style>
                    <property name="text">Cancel</property>
                </style>
            </part>
        </part>
        <!-- buttonPanel2 ends -->
        <!-- inputPanel includes all the input fields and their labels -->
        <part name="inputPanel" class="JPanel">
            <style>
                <property name="layout">java.awt.BorderLayout</property>
                <property name="borderAlignment">North</property>
```

```
</style>
<!-- dateAndAmountPanel has input fields conserning
dates and amount and their labels -->
<part name="dateAndAmountPanel" class="JPanel">
    <style>
        <property name="layout">java.awt.GridLayout</property>
        <property name="borderAlignment">South</property>
    </style>
    <!-- dateBasePanel contains the fields and label conserning the date-->
    <part name="dateBasePanel" class="JPanel">
        <style>
            <property name="layout">java.awt.GridBagLayout</property>
        </style>
        <part name="dateLabel" class="JLabel">
            <style>
                <property name="text">Due date:</property>
                <property name="gridx">0</property>
                <property name="gridy">0</property>
            </style>
        </part>
        <part name="dateFieldPanel" class="JPanel">
            <style>
                <property name="gridx">1</property>
                <property name="gridy">0</property>
            </style>
            <part name="dayField" class="JTextField">
                <style>
                    <property name="toolTipText">Day</property>
                    <property name="columns">2</property>
                </style>
            </part>
            <part name="monthField" class="JTextField">
                <style>
                    <property name="toolTipText">Month</property>
                    <property name="columns">2</property>
                </style>
            </part>
            <part name="yearField" class="JTextField">
                <style>
                    <property name="toolTipText">Year</property>
                    <property name="columns">4</property>
                </style>
            </part>
        </part>
    </part>
    <!-- dateBasePanel ends -->
    <!-- amountBasePanel has label and input field conserning the amount -->
    <part name="amountBasePanel" class="JPanel">
        <style>
            <property name="layout">java.awt.GridBagLayout</property>
        </style>
        <part name="amountLabelPanel" class="JPanel">
            <style>
                <property name="gridx">0</property>
                <property name="gridy">0</property>
            </style>
            <part name="amountLabel" class="JLabel">
                <style>
                    <property name="text">Amount:</property>
                </style>
            </part>
        </part>
        <part name="amountFieldPanel" class="JPanel">
            <style>
                <property name="gridx">1</property>
                <property name="gridy">0</property>
            </style>
            <part name="amountField" class="JTextField">
                <style>
                    <property name="columns">6</property>
                </style>
            </part>
        </part>
```

```
                        </part>
                        <!-- amountBasePanel ends-->
                    </part>
                    <!-- dateAndAmountPanel ends -->
                    <!-- northPanel has fields and labels conserning the payee -->
                    <part name="northPanel" class="JPanel">
                        <style>
                            <property name="layout">java.awt.GridLayout</property>
                            <property name="borderAlignment">North</property>
                            <property name="layout_columns">2</property>
                        </style>
                        <!-- payeePanel has fields for payee's name and account and labels for them -->
                        <part name="payeePanel" class="JPanel">
                            <style>
                                <property name="layout">java.awt.GridLayout</property>
                                <property name="layout_columns">2</property>
                            </style>
                            <part name="payeeLabelPanel" class="JPanel">
                                <style>
                                    <property name="layout">java.awt.GridLayout</property>
                                    <property name="layout_rows">2</property>
                                    <property name="layout_vgap">3</property>
                                </style>
                                <part name="payeeNameLabel" class="JLabel">
                                    <style>
                                        <property name="text">Name:</property>
                                    </style>
                                </part>
                                <part name="payeeAccountLabel" class="JLabel">
                                    <style>
                                        <property name="text">Account:</property>
                                    </style>
                                </part>
                            </part>
                            <part name="payeeFieldPanel" class="JPanel">
                                <style>
                                    <property name="layout">java.awt.GridLayout</property>
                                    <property name="layout_rows">2</property>
                                    <property name="layout_vgap">3</property>
                                </style>
                                <part name="payeeNameField" class="JTextField">
                                    <style>
                                        <property name="columns">10</property>
                                    </style>
                                </part>
                                <part name="payeeAccountField" class="JTextField">
                                    <style>
                                        <property name="columns">10</property>
                                    </style>
                                </part>
                            </part>
                        </part>
                        <!-- payeePanel ends -->
                        <!-- notePanel has input field for note -->
                        <part name="notePanel" class="JPanel">
                            <style>
                                <property name="layout">java.awt.GridLayout</property>
                            </style>
                            <part name="noteScrollPane" class="JScrollPane">
                                <part name="noteTextArea" class="JTextArea">
                                    <style>
                                        <property name="columns">10</property>
                                    </style>
                                </part>
                            </part>
                        </part>
                        <!-- notePanel ends -->
                    </part>
                    <!-- northPanel ends -->
                </part>
                <!-- inputPanel ends -->
            </part>
        </part>
```

```
          </part>
     </structure>
     <behavior>
          <rule>
<!--
This rule sets the action for pull-down menu item, which opens new bill dialog
-->
               <condition>
                    <event class="actionPerformed" part-name="menuItem"/>
               </condition>
               <action>
                    <call name="bank.setBillTitle"/>
                    <property part-name="payeeNameField" name="text"/>
                    <property part-name="payeeAccountField" name="text"/>
                    <property part-name="noteTextArea" name="text"/>
                    <property part-name="dayField" name="text"/>
                    <property part-name="monthField" name="text"/>
                    <property part-name="yearField" name="text"/>
                    <property part-name="amountField" name="text"/>
                    <property part-name="invoiceDialog" name="visible">true</property>
               </action>
          </rule>
          <rule>
<!--
This rule sets the action for newButton, which opens new bill dialog
-->
               <condition>
                    <event class="actionPerformed" part-name="newButton"/>
               </condition>
               <action>
                    <call name="bank.setBillTitle"/>
                    <property part-name="payeeNameField" name="text"/>
                    <property part-name="payeeAccountField" name="text"/>
                    <property part-name="noteTextArea" name="text"/>
                    <property part-name="dayField" name="text"/>
                    <property part-name="monthField" name="text"/>
                    <property part-name="yearField" name="text"/>
                    <property part-name="amountField" name="text"/>
                    <property part-name="invoiceDialog" name="visible">true</property>
               </action>
          </rule>
          <rule>
<!--
This rule sets the action for deleteButton, which deletes selected message.
It does this by calling Java method deleteBill
-->
               <condition>
                    <event class="actionPerformed" part-name="deleteButton"/>
               </condition>
               <action>
                    <call name="bank.deleteBill"/>
               </action>
          </rule>
          <rule>
<!--
This rule sets the action for copyasnewButton, which opens new bill dialog and
copies the selected bill's information into the new bill.
It does this by calling Java method copyAsNewBill
-->
               <condition>
                    <event class="actionPerformed" part-name="copyasnewButton"/>
               </condition>
               <action>
                    <call name="bank.copyAsNewBill"/>
               </action>
          </rule>
          <rule>
<!--
This rule set the action for part ok, which saves the new bill.
It does this by calling Java method insertBill. After the method call
it clears the fields and makes the dialog invisible.
-->
               <condition>
```

```
                                        <event class="actionPerformed" part-name="ok"/>
                                </condition>
                                <action>
                                        <call name="bank.insertBill"/>
                                        <property part-name="invoiceDialog" name="visible">false</property>
                                        <property part-name="payeeNameField" name="text"/>
                                        <property part-name="payeeAccountField" name="text"/>
                                        <property part-name="noteTextArea" name="text"/>
                                        <property part-name="dayField" name="text"/>
                                        <property part-name="monthField" name="text"/>
                                        <property part-name="yearField" name="text"/>
                                        <property part-name="amountField" name="text"/>
                                </action>
                        </rule>
                        <rule>
<!--
This rule set the action for part cancel, which cancels the new bill and makes
the dialog invisible.
-->
                                <condition>
                                        <event class="actionPerformed" part-name="cancel"/>
                                </condition>
                                <action>
                                        <property part-name="invoiceDialog" name="visible">false</property>
                                        <property part-name="payeeNameField" name="text"/>
                                        <property part-name="payeeAccountField" name="text"/>
                                        <property part-name="noteTextArea" name="text"/>
                                        <property part-name="dayField" name="text"/>
                                        <property part-name="monthField" name="text"/>
                                        <property part-name="yearField" name="text"/>
                                        <property part-name="amountField" name="text"/>
                                </action>
                        </rule>
                        <rule>
<!--
This rule set the action for part English, which changes the user interface language
in to english.
-->
                                <condition>
                                        <event class="itemStateChanged" part-name="English"/>
                                </condition>
                                <action>
                                        <property part-name="Banking" name="title">Banking application</property>
                                        <property part-name="menu" name="text">File</property>
                                        <property part-name="menuItem" name="text">New Bill</property>
                                        <property part-name="nameLabel" name="text">Name:</property>
                                        <property part-name="accountLabel" name="text">Account:</property>
                                        <property part-name="balanceLabel" name="text">Balance:</property>
                                        <property part-name="newButton" name="text">New Bill</property>
                                        <property part-name="tabPanel1" name="name">Unpaid Bills</property>
                                        <property part-name="tabPanel2" name="name">Settings</property>
                                        <property part-name="copyasnewButton" name="text">Copy as New</property>
                                        <property part-name="deleteButton" name="text">Delete</property>
                                        <property part-name="English" name="text">English</property>
                                        <property part-name="Finnish" name="text">Finnish</property>
                                        <property part-name="invoiceDialog" name="title">New Bill</property>
                                        <property part-name="ok" name="text">Ok</property>
                                        <property part-name="cancel" name="text">Cancel</property>
                                        <property part-name="dateLabel" name="text">Due Date:</property>
                                        <property part-name="amountLabel" name="text">Amount:</property>
                                        <property part-name="payeeNameLabel" name="text">Name:</property>
                                        <property part-name="payeeAccountLabel" name="text">Account:</property>
                                        <call name="bank.setLanguage">
                                                <param>0</param>
                                        </call>
                                </action>
                        </rule>
                        <rule>
<!--
This rule set the action for part Finnish, which changes the user interface language
in to finnish.
-->
                                <condition>
                        <event class="itemStateChanged" part-name="Finnish"/>
                    </condition>
                    <action>
                        <property part-name="Banking" name="title">Pankki sovellus</property>
                        <property part-name="menu" name="text">Tiedosto</property>
                        <property part-name="menuItem" name="text">Uusi lasku</property>
                        <property part-name="nameLabel" name="text">Nimi:</property>
                        <property part-name="accountLabel" name="text">Tilinumero:</property>
                        <property part-name="balanceLabel" name="text">Saldo:</property>
                        <property part-name="newButton" name="text">Uusi Lasku</property>
                        <property part-name="tabPanel1" name="name">Maksamattomat laskut</property>
                        <property part-name="tabPanel2" name="name">Asetukset</property>
                        <property part-name="copyasnewButton" name="text">Kopio laskuksi</property>
                        <property part-name="deleteButton" name="text">Poista</property>
                        <property part-name="English" name="text">Englanti</property>
                        <property part-name="Finnish" name="text">Suomi</property>
                        <property part-name="invoiceDialog" name="title">Uusi lasku</property>
                        <property part-name="ok" name="text">Ok</property>
                        <property part-name="cancel" name="text">Peruuta</property>
                        <property part-name="dateLabel" name="text">Eräpäivä:</property>
                        <property part-name="amountLabel" name="text">Summa:</property>
                        <property part-name="payeeNameLabel" name="text">Nimi:</property>
                        <property part-name="payeeAccountLabel" name="text">Tilinumero:</property>
                        <call name="bank.setLanguage">
                            <param name="language">1</param>
                        </call>
                    </action>
                </rule>
            </behavior>
    </interface>
    <peers>
        <logic>
<!--
This logic element maps the UIML method calls into the real Java methods and classes.
-->
            <d-component name="bank" maps-to="thesis.uimlbanking.Banking2">
                <d-method name="deleteBill" maps-to="deleteBill"/>
                <d-method name="copyAsNewBill" maps-to="copyAsNewBill"/>
                <d-method name="insertBill" maps-to="insertBill"/>
                <d-method name="setBillTitle" maps-to="setBillTitle"/>
                <d-method name="setLanguage" maps-to="setLanguage">
                    <d-param type="int" name="language"/>
                </d-method>
            </d-component>
        </logic>
    </peers>
</uiml>
```

WAP application

```
<?xml version="1.0"?>
<!DOCTYPE uiml PUBLIC "-//UIT//DTD UIML 2.0 Draft//EN" "C:\UIML2\bin\UIML2_0d.dtd">
<uiml>
    <interface>
        <structure>
            <part name="Banking" class="Wml">
                <!-- startpage is the main card and starting place for the application -->
                <part name="startpage" class="Card">
                    <part name="startpara" class="P">
                        <part name="starttext" class="RichText"/>
                    </part>
                </part>
                <!-- user card has the user information-->
                <part name="user" class="Card">
                    <part name="prev1" class="Do"/>
                    <part name="userpara" class="P">
                        <part name="username" class="RichText"/>
                        <part name="ownaccount" class="Select">
                            <part name="option1" class="Option">
                                <part name="option1txt" class="RichText"/>
                            </part>
                            <part name="option2" class="Option">
                        <part name="option2txt" class="RichText"/>
                    </part>
                </part>
                <part name="userbalance" class="RichText"/>
            </part>
        </part>
        <!-- new card is used for inputing a new bill -->
        <part name="new" class="Card">
            <part name="prev2" class="Do"/>
            <part name="newpara" class="P">
                <part name="new1" class="RichText"/>
                <part name="namefield" class="Input"/>
                <part name="new2" class="RichText"/>
                <part name="accountfield" class="Input"/>
                <part name="new3" class="RichText"/>
                <part name="notefield" class="Input"/>
                <part name="new4" class="RichText"/>
                <part name="datefield" class="Input"/>
                <part name="new5" class="RichText"/>
                <part name="amountfield" class="Input"/>
            </part>
        </part>
        <!-- old card shows the information concerning the selected bill-->
        <part name="old" class="Card">
            <part name="prev3" class="Do"/>
                <part name="p1" class="P">
                    <part name="unpaid1" class="RichText"/>
                </part>
                <part name="p1" class="P">
                    <part name="unpaid2" class="RichText"/>
                </part>
                <part name="p1" class="P">
                    <part name="unpaid3" class="RichText"/>
                </part>
                <part name="p1" class="P">
                    <part name="unpaid4" class="RichText"/>
                </part>
                <part name="p1" class="P">
                    <part name="unpaid5" class="RichText"/>
                </part>
        </part>
        <!-- bills card is a list of bills, can show only one
        bill so list is bit ambitious name for it-->
        <part name="bills" class="Card">
            <part name="prev4" class="Do"/>
            <part name="delete" class="Do"/>
            <part name="listofbills" class="P">
                <part name="listtext" class="RichText"/>
            </part>
        </part>
    </part>
</structure>

<style>
    <!-- properties for startpage card -->
    <property part-name="startpage" name="title">Banking application</property>
    <!-- property starttext is used to input wml code directly, because current version of
    the renderer does not support the tags we want -->
    <property part-name="starttext" name="content">
        &lt;anchor&gt; User Information &lt;go href="#user"/&gt;&lt;/anchor&gt; &lt;br/&gt;
        &lt;anchor&gt; List of unpaid bills &lt;go href="#bills"/&gt;&lt;/anchor&gt; &lt;br/&gt;
        &lt;anchor&gt; New bill &lt;go href="#new"/&gt;&lt;/anchor&gt;
    </property>

    <!-- properties for user card -->
    <property part-name="user" name="title">User Information</property>
    <property part-name="username" name="content">name: Jack Banker</property>
    <property part-name="ownaccount" name="title">Account:</property>
    <property part-name="option1txt" name="content">1234-867</property>
    <property part-name="option2txt" name="content">1234-124</property>
    <property part-name="userbalance" name="content">Balance: $(pounds)</property>

    <!-- properties for new card -->
```

```
          <property part-name="new" name="title">New bill</property>
          <property part-name="new1" name="content">Name:</property>
          <property part-name="new2" name="content">Account:</property>
          <property part-name="new3" name="content">Note:</property>
          <property part-name="new4" name="content">Date:</property>
          <property part-name="new5" name="content">Amount:</property>
          <property part-name="namefield" name="name">name</property>
          <property part-name="accountfield" name="name">account</property>
          <property part-name="notefield" name="name">note</property>
          <property part-name="datefield" name="name">date</property>
          <property part-name="amountfield" name="name">amount</property>

          <!-- properties for old card -->
          <property part-name="old" name="title">Unpaid bill</property>
          <property part-name="unpaid1" name="content">name: $name</property>
          <property part-name="unpaid2" name="content">Account: $account</property>
          <property part-name="unpaid3" name="content">Note: $note</property>
          <property part-name="unpaid4" name="content">Due Date: $date</property>
          <property part-name="unpaid5" name="content">Amount: $amount</property>

          <!-- properties for bills card -->
          <property part-name="bills" name="title">List of unpaid bills</property>
          <!-- property listtext is used to input wml code directly, because current version of
          the renderer does not support the tags we want -->
          <property part-name="listtext" name="content">
             &lt;anchor&gt;Name: $name&lt;go href="#old"/&gt;&lt;/anchor&gt; &lt;br/&gt;
             Date: $date &lt;br/&gt;
             Amount: $amount
          </property>
       </style>

       <behavior>
     <rule>
          <!--this rule does nothing, rendered want's a rule for Option tags -->
        <condition>
           <event class="Onpick" part-name="option1" name="itemselect"/>
             </condition>
      <action>
           </action>
     </rule>
     <rule>
          <!--this rule does nothing, rendered want's a rule for Option tags -->
        <condition>
           <event class="Onpick" part-name="option2" name="itemselect"/>
             </condition>
      <action>
           </action>
          </rule>
     <rule>
       <!--this rule moves the WAP browser to previous page-->
             <condition>
          <event class="prev" part-name="user" name="do"/>
           </condition>
           <action>
                 <property part-name="prev" name="visible">true</property>
             </action>
          </rule>
     <rule>
       <!--this rule moves the WAP browser to previous page-->
      <condition>
          <event class="prev" part-name="new" name="do"/>
           </condition>
           <action>
                 <property part-name="prev" name="visible">true</property>
             </action>
          </rule>
          <rule>
       <!--this rule moves the WAP browser to previous page-->
      <condition>
          <event class="prev" part-name="old" name="do"/>
           </condition>
           <action>
                 <property part-name="prev" name="visible">true</property>
```

```
                    </action>
                </rule>
                <rule>
            <!--this rule moves the WAP browser to previous page-->
            <condition>
                <event class="prev" part-name="bills" name="do"/>
                  </condition>
                  <action>
                        <property part-name="prev" name="visible">true</property>
                    </action>
                </rule>
                <rule>
            <!--this rule moves the WAP browser to previous page-->
            <condition>
                <event class="accept" part-name="bills" name="do"/>
                  </condition>
                  <action>
                        <property part-name="prev" name="visible">true</property>
                    </action>
                </rule>
            </behavior>
        </interface>
</uiml>
```

Appendix 3: Java code

Common code for PC applications

Class InfoCollection

```
//Title:       UIML Banking application
//Version:     1.0
//Author:      Sami Rantala
//Company:     L M Ericsson Ltd.
//Description: This program is used to compare Swing and UIML interfaces.
package thesis.common;

import java.util.Vector;

public class InfoCollection {
  Vector store;

  /* InfoCollection()
   * Class constructor, places 2 bills into unpaid list.
   */
  public InfoCollection() {
    store = new Vector(10,4);
    InvoiceInfo start1 = new InvoiceInfo("John Doe", "1234-234", "Rent",150, 27,9,2000);
    InvoiceInfo start2 = new InvoiceInfo("Joe Smith", "9865-314", "Rent",300, 27,10,2000);
    store.addElement(start1);
    store.addElement(start2);
  }

  /* getInvoice(int i)
   * This method returns requested invoiceInfo instance from the collection.
   */
  public InvoiceInfo getInvoice(int i) {
    return (InvoiceInfo)store.elementAt(i);
    }

  /* deleteRow(int i)
   * This method removes the requested invoiceInfo from the collection.
   */
  public void deleteRow(int i){store.removeElementAt(i);}

  /* addInvoice(Object o)
   * This method adds an Object o into the collection
   */
  public void addInvoice(Object o) {
    store.addElement(o);
  }

  /* getSize()
   * This method returns collections size
   */
  public int getSize() {return store.size();}
}
```

Class InvoiceInfo

```
//Title:       UIML Banking application
//Version:     1.0
//Author:      Sami Rantala
//Company:     L M Ericsson Ltd.
//Description: This program is used to compare Swing and UIML interfaces.
package thesis.common;

import java.util.*;
import java.lang.Integer;

public class InvoiceInfo {
  private String payee;
  private String payeeAccount;
  private String note;
  private Integer amount;

  private Calendar date = Calendar.getInstance();

  public InvoiceInfo() {
  }
  public InvoiceInfo(String a,String b, String c, int d, int e, int f, int g) {
  payee = new String(a);
  payeeAccount = new String(b);
  note = new String(c);
  amount = new Integer(d);
  date.set(g,f,e);
  }

  /*Methods for getting variables from this class*/
  public Object getPayee(){return payee;}
  public Object getPayeeAccount(){return payeeAccount;}
  public Object getNote(){return note;}
  public Object getAmount(){return amount;}
  public Object getDay(){return java.lang.Integer.toString(date.get(date.DAY_OF_MONTH));}
  public Object getMonth(){return java.lang.Integer.toString(date.get(date.MONTH));}
  public Object getYear(){return java.lang.Integer.toString(date.get(date.YEAR));}
  public Object getInvoiceDate(){return date.getTime();}

  /*Methods for setting variables*/
  public void setPayee(String a){payee = new String(a);}
  public void setPayeeAccount(String b){payeeAccount = new String(b);}
  public void setNote(String c){note = new String(c);}
  public void setAmount(int d){amount = new Integer(d);}
  public void setInvoiceDate(int y, int m, int d){date.set(y,m,d);}
}
```

Class MyTableModel

```
//Title:        UIML Banking application
//Version:      1.0
//Author:       Sami Rantala
//Company:      L M Ericsson Ltd.
//Description:  This program is used to compare Swing and UIML interfaces.
package thesis.common;

import java.text.*;

public class MyTableModel extends javax.swing.table.AbstractTableModel {
    String[] columnNames = {"Due Date", "Payee", "£"};
    String[] columnNames2 = {"Eräpäivä", "Saaja", "£"};
    InfoCollection cont;
    InvoiceInfo info;
    DateFormat df = DateFormat.getDateInstance(DateFormat.DEFAULT);
    int language;

    public MyTableModel(InfoCollection c){
      cont = c;
    }

    public void setLanguage(int language){
      this.language = language;
    }

    /*gets the table column count*/
    public int getColumnCount() {
      return columnNames.length;
    }
    /* gets the table row count  */
    public int getRowCount() {
      return cont.getSize();
    }

    /*gets the tables column names*/
    public String getColumnName(int col) {
```

```
    if(language == 0){
     return columnNames[col];
    } else {return columnNames2[col];}
  }

  /*Gets unpaid bills table information from invoiceInfo*/
  public Object getValueAt(int row, int col) {
   info = cont.getInvoice(row);
    switch(col){
      case 0:  return df.format(info.getInvoiceDate());
      case 1:  return info.getPayee();
      case 2:  return info.getAmount();
      default: return null;
    }
  }

  /*Gets the class of table entries*/
  public Class getColumnClass(int c) {
    return getValueAt(0, c).getClass();
  }
}
```

UIML PC application

Class Banking2

```
//Title:        UIML Banking application
//Version:      1.0
//Author:       Sami Rantala
//Company:      L M Ericsson Ltd.
//Description:  This program is used to compare Swing and UIML interfaces.
package thesis.uimlbanking;

import com.universalit.renderer.Renderer;
import javax.swing.*;
import java.awt.event.*;
import javax.swing.event.*;
import javax.swing.border.*;
import thesis.common.*;

public class Banking2 {
  static String uimlFileName = "banking.uiml";
  static Renderer r;
  private MyTableModel myModel;
  private InfoCollection unpaidBills;
  private JTable table;
  private ListSelectionModel rowSM;
  private int selectedRow = -1;
  private Border emptyBorder10pt;
  private Border etchedBorder;
  private TitledBorder noteBorder = BorderFactory.createTitledBorder(etchedBorder);
  private TitledBorder payeeBorder = BorderFactory.createTitledBorder(etchedBorder);
  private int lan =0;

  public Banking2() {
    emptyBorder10pt = BorderFactory.createEmptyBorder(10,10,10,10);
    table = (JTable) r.getPartByName("table");
    unpaidBills = new InfoCollection();
    myModel = new MyTableModel(unpaidBills);

    if(table != null){
      table.setModel(myModel);
      table.setAutoResizeMode(table.AUTO_RESIZE_ALL_COLUMNS);
      table.sizeColumnsToFit(table.AUTO_RESIZE_ALL_COLUMNS);
      rowSM = table.getSelectionModel();
      rowSM.addListSelectionListener(new ListSelectionListener() { // this part of the finds out
       public void valueChanged(ListSelectionEvent e) {      // which row is selected
         //Ignore extra messages.
         if (e.getValueIsAdjusting()) return;
         ListSelectionModel lsm = (ListSelectionModel)e.getSource();
         if (lsm.isSelectionEmpty()) {
           //no rows are selected
```

```
        selectedRow = -1;
       } else {
        selectedRow = lsm.getMinSelectionIndex();
        //row selectedRow is selected
       }
      }
     });
    }
    setBorders();
    setBorders2();
    JFrame j = (JFrame)r.getPartByName("Banking");
    j.pack();
    j.setVisible(true);
  }


  /*  insertBill(...)
   *  This method inserts a new bill into the unpaidBills list
   */
  public void insertBill() {
    String payeeName = new String();
    String payeeAccount = new String();
    String noteText = new String();
    String amount = new String();
    String year = new String();
    String month = new String();
    String day = new String();
    InvoiceInfo info = new InvoiceInfo();
    // first get the input components and then their contents
    JTextField a = (JTextField)r.getPartByName("payeeNameField");
    if(a!=null)
      payeeName = a.getText();
    JTextField b = (JTextField)r.getPartByName("payeeAccountField");
    if(b!=null)
      payeeAccount = b.getText();
    JTextArea c = (JTextArea)r.getPartByName("noteTextArea");
    if(c!=null)
      noteText = c.getText();
    JTextField d = (JTextField)r.getPartByName("amountField");
    if(d!=null)
      amount = d.getText();
    JTextField e = (JTextField)r.getPartByName("yearField");
    if(e!=null)
      year = e.getText();
    JTextField f = (JTextField)r.getPartByName("monthField");
    if(f!=null)
      month = f.getText();
    JTextField g = (JTextField)r.getPartByName("dayField");
    if(g!=null)
      day = g.getText();

    // Set the bill information in to instance of InvoiceInfo class
    if(payeeName!=null)
      info.setPayee(payeeName);
    if(payeeAccount!=null)
      info.setPayeeAccount(payeeAccount);
    if(noteText!=null)
      info.setNote(noteText);
    if(amount!=null)
      info.setAmount(java.lang.Integer.parseInt(amount));
    if(year!=null && year!=null && day!=null)
      info.setInvoiceDate(java.lang.Integer.parseInt(year),
                    java.lang.Integer.parseInt(month),
                    java.lang.Integer.parseInt(day));
    if(info!=null)
      unpaidBills.addInvoice(info);
    refreshTable(unpaidBills.getSize());
    cancelNewBill();
  }


  /*  refreshTable(int i)
   *  This method fires an event that updates the table.
  */
 public void refreshTable(int i) {
   myModel.fireTableRowsInserted(i,i);
 }


 /*  deleteBill(...)
  *  This method deletes a bill from the unpaidBills list
  */
 public void deleteBill() {
   if(selectedRow != -1){
     unpaidBills.deleteRow(selectedRow);
     myModel.fireTableRowsDeleted(selectedRow,selectedRow);
     selectedRow = -1;
   }
 }


 /*  copyAsNewBill()
  *  This method copies the currently selected bill as a new bill
  */
 public void copyAsNewBill() {
   if(selectedRow != -1){
     // If selectedRow is larger than -1, then
     // we copy selected bill's info into the new one.
     // if it's -1 then no row is selected.
     InvoiceInfo invoice = unpaidBills.getInvoice(selectedRow);
     ((JTextField)r.getPartByName("payeeNameField")).setText((String)invoice.getPayee());
     ((JTextField)r.getPartByName("payeeAccountField")).setText((String)invoice.getPayeeAccount());
     ((JTextArea)r.getPartByName("noteTextArea")).setText((String)invoice.getNote());
     ((JTextField)r.getPartByName("dayField")).setText((String)invoice.getDay());
     ((JTextField)r.getPartByName("monthField")).setText((String)invoice.getMonth());
     ((JTextField)r.getPartByName("yearField")).setText((String)invoice.getYear());
     String a = ((Integer)invoice.getAmount()).toString();
     ((JTextField)r.getPartByName("amountField")).setText(a);
     JDialog d = (JDialog)r.getPartByName("invoiceDialog");
     if(lan == 0){
       d.setTitle("New bill(Copied from old)");}
     if (lan == 1){
       d.setTitle("Uusi lasku(kopio)");}
     d.setVisible(true);
   }
 }

 /* setBillTitle()
  * Sets the new bill dialogs title
  */
 public void setBillTitle() {
   JDialog d = (JDialog)r.getPartByName("invoiceDialog");
   if(lan == 0){
     d.setTitle("New bill");}
   else{
     d.setTitle("Uusi lasku");}
 }

 /*  cancelNewBill()
  *  This method is used to cancel the new bill dialog.
  */
 public void cancelNewBill() {
   JDialog d = (JDialog)r.getPartByName("invoiceDialog");
   d.setVisible(false);
 }

 /*  setLanguage(int language)
  *  This method changes the languages of those parts that cannot
  *  be changed directly from UIML code
  */
 public void setLanguage(int language){
   lan = language;
   JTabbedPane TabbedPanel = (JTabbedPane)r.getPartByName("tabPane");
   if(lan == 0){
     TabbedPanel.setTitleAt(0, "Unpaid Bills");
     TabbedPanel.setTitleAt(1, "Settings");
```

```
      payeeBorder.setTitle("Payee");
      noteBorder.setTitle("Note");
      myModel.setLanguage(0);
      // Make sure the table is painted correctly
      myModel.fireTableStructureChanged();
      table.sizeColumnsToFit(table.AUTO_RESIZE_ALL_COLUMNS);
    }
    if(lan == 1){
      TabbedPanel.setTitleAt(0, "Maksamattomat laskut");
      TabbedPanel.setTitleAt(1, "Asetukset");
      payeeBorder.setTitle("Saajan tiedot");
      noteBorder.setTitle("Viesti");
      myModel.setLanguage(1);
      // Make sure the table is painted correctly
      myModel.fireTableStructureChanged();
      table.sizeColumnsToFit(table.AUTO_RESIZE_ALL_COLUMNS);
    }
  }


  /*  setBorders()
   *  This method sets the borders for the Banking application JFrame
   */
  private void setBorders() {
    JPanel a = (JPanel)r.getPartByName("extraPanel");
    a.setBorder(emptyBorder10pt);
    JPanel b =(JPanel)r.getPartByName("userInformationPanel");
    b.setBorder(emptyBorder10pt);
    JPanel c = (JPanel)r.getPartByName("newButtonPanel");
    c.setBorder(emptyBorder10pt);
    JPanel d = (JPanel)r.getPartByName("tabPanel1");
    d.setBorder(emptyBorder10pt);
  }


  /*  setBorders2()
   *  This method sets the borders for invoiceDialog
   */
  private void setBorders2(){
    etchedBorder = BorderFactory.createEtchedBorder();

    JPanel d = (JPanel)r.getPartByName("inputPanel");
    if(d!= null)
      d.setBorder(emptyBorder10pt);
    JPanel e = (JPanel)r.getPartByName("dateAndAmountPanel");
    if(e!= null)
      e.setBorder(BorderFactory.createCompoundBorder(BorderFactory.createEmptyBorder(4,2,0,2),etchedBorder));
    JPanel f = (JPanel)r.getPartByName("payeePanel");
    if(f!= null)
      f.setBorder(BorderFactory.createCompoundBorder(payeeBorder,emptyBorder10pt));
    JPanel g = (JPanel)r.getPartByName("notePanel");
    if(g!= null)
      g.setBorder(BorderFactory.createCompoundBorder(noteBorder,emptyBorder10pt));
    payeeBorder.setTitle("Payee");
    noteBorder.setTitle("Note");
  }


      public static void main(String[] args) {
    try {
      UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
      } catch (Exception e) { }
       r = new Renderer();

        boolean renderOK = r.renderUIML(uimlFileName);
    if(!renderOK){
      System.exit(1);
    }

    Banking2 banking2 = new Banking2();
    r.registerComponentInstance("bank.Banking2.", banking2);
  }
}
```

Java PC application

Class Banking

```
//Title:       Banking application
//Version:     1.0
//Author:      Sami Rantala
//Company:     L M Ericsson Ltd.
//Description: This is used to compare Swing and UIML interfaces.

package thesis.javabanking;
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

public class Banking extends WindowAdapter {
  MainFrame GUI;

  /*  Banking()
   *  This is the constructor.
   */
  public Banking() {
    GUI = new MainFrame();
    GUI.addWindowListener(this);
    GUI.setVisible(true);
    GUI.pack();
  }

  /*  windowClosing(WindowEvent e)
   *  This method closes the window.
   */
  public void windowClosing(WindowEvent e) {
  System.exit(0);
  }

  public static void main(String[] args) {
    try {
    UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
      } catch (Exception e) { }
      Banking banking = new Banking();
  }
}
```

Class Invoice

```
//Title:       Banking application
//Version:     1.0
//Author:      Sami Rantala
//Company:     L M Ericsson Ltd.
//Description: This is used to compare Swing and UIML interfaces.

package thesis.javabanking;

import java.awt.*;
import javax.swing.*;
import javax.swing.border.*;
import java.awt.event.*;
import java.lang.String;
import java.lang.Integer;
import thesis.common.*;

public class Invoice extends JDialog {
  JPanel BasePanel = new JPanel();
  JPanel DateAndAmountPanel = new JPanel();
  JPanel NotePanel = new JPanel();
  JPanel PayeeLabel = new JPanel();
  JPanel Payee = new JPanel();
  JPanel NorthPanel = new JPanel();
  JPanel PayeeField = new JPanel();
  JPanel InputPanel = new JPanel();
  JPanel AmountBasePanel = new JPanel();
JPanel DateBasePanel = new JPanel();
JPanel DateFieldPanel = new JPanel();
JPanel AmountLabelPanel = new JPanel();
JPanel AmountFieldPanel = new JPanel();
JPanel ButtonPanel = new JPanel();

JScrollPane jScrollPane2 = new JScrollPane();

JLabel PayeeNameLabel = new JLabel();
JLabel PayeeAccountLabel = new JLabel();
JLabel AmountLabel = new JLabel();
JLabel DateLabel = new JLabel();

JTextArea NoteTextArea = new JTextArea();

JTextField DayField = new JTextField();
JTextField YearField = new JTextField();
JTextField MonthField = new JTextField();
JTextField AmountField = new JTextField();
JTextField PayeeAccountField = new JTextField();
JTextField PayeeNameField = new JTextField();

JButton OkButton = new JButton();
JButton CancelButton = new JButton();

BorderLayout borderLayout1 = new BorderLayout();
BorderLayout borderLayout3 = new BorderLayout();

GridLayout gridLayout1 = new GridLayout();
GridLayout gridLayout3 = new GridLayout();
GridLayout gridLayout4 = new GridLayout();
GridLayout gridLayout5 = new GridLayout();
GridLayout gridLayout6 = new GridLayout();
GridLayout gridLayout7 = new GridLayout();

GridBagLayout gridBagLayout1 = new GridBagLayout();
GridBagLayout gridBagLayout2 = new GridBagLayout();

Border etchedBorder;

InvoiceInfo info = new InvoiceInfo();
InfoCollection unpaidBills;
MainFrame fatherFrame;
Border emptyBorder10pt;
TitledBorder noteBorder;
TitledBorder payeeBorder;

int language;

/*  Invoice(Frame frame, String title, boolean modal, InfoCollection unpaid)
 *  This is one of the constructors. It forwards calls to the main constructor.
 */
public Invoice(Frame frame, String title, boolean modal, InfoCollection unpaid) {
this(frame, title, modal, unpaid, null);
}

/*  Invoice(Frame frame, String title, boolean modal, InfoCollection unpaid, InvoiceInfo invoiceInfo)
 *  This is the main constructor.
 */
public Invoice(Frame frame, String title, boolean modal, InfoCollection unpaid, InvoiceInfo invoiceInfo) {
  super(frame, title, modal);
  try {
   jbInit();
   unpaidBills = unpaid;
   fatherFrame = (MainFrame)frame;
   if(invoiceInfo != null){
    PayeeNameField.setText((String)invoiceInfo.getPayee());
    PayeeAccountField.setText((String)invoiceInfo.getPayeeAccount());
    NoteTextArea.setText((String)invoiceInfo.getNote());
    DayField.setText((String)invoiceInfo.getDay());
    MonthField.setText((String)invoiceInfo.getMonth());
    YearField.setText((String)invoiceInfo.getYear());
    AmountField.setText(((Integer)invoiceInfo.getAmount()).toString());
```

```
      }
      pack();
    }
    catch(Exception ex) {
      ex.printStackTrace();
    }
  }

  /* jbInit()
   * This method intitializes Invoice dialog GUI.
   */
  void jbInit() throws Exception {
    etchedBorder = BorderFactory.createEtchedBorder(Color.white,new java.awt.Color(134, 134, 134));
    emptyBorder10pt = BorderFactory.createEmptyBorder(10,10,10,10);
    noteBorder = BorderFactory.createTitledBorder(etchedBorder);
    payeeBorder = BorderFactory.createTitledBorder(etchedBorder);
    BasePanel.setLayout(borderLayout1);
    NotePanel.setLayout(gridLayout5);
    NotePanel.setBorder(BorderFactory.createCompoundBorder(noteBorder,emptyBorder10pt));
    PayeeLabel.setLayout(gridLayout6);
    Payee.setLayout(gridLayout4);
    Payee.setBorder(BorderFactory.createCompoundBorder(payeeBorder,emptyBorder10pt));
    NorthPanel.setLayout(gridLayout3);
    NoteTextArea.setWrapStyleWord(true);
    gridLayout3.setColumns(2);
    gridLayout4.setColumns(2);
    gridLayout6.setRows(2);
    gridLayout6.setVgap(3);
    jScrollPane2.setHorizontalScrollBarPolicy(JScrollPane.HORIZONTAL_SCROLLBAR_NEVER);
    jScrollPane2.setVerticalScrollBarPolicy(JScrollPane.VERTICAL_SCROLLBAR_ALWAYS);
    gridLayout7.setRows(2);
    gridLayout7.setVgap(3);
    PayeeField.setLayout(gridLayout7);
    InputPanel.setLayout(borderLayout3);
    DateAndAmountPanel.setLayout(gridLayout1);
    DateBasePanel.setLayout(gridBagLayout1);
    AmountBasePanel.setLayout(gridBagLayout2);

    YearField.setColumns(4);
    MonthField.setColumns(2);
    AmountField.setColumns(7);
    DayField.setColumns(2);

    OkButton.addActionListener(new java.awt.event.ActionListener() {

      public void actionPerformed(ActionEvent e) {
        OkButton_actionPerformed(e);
      }
    });

    CancelButton.addActionListener(new java.awt.event.ActionListener() {

      public void actionPerformed(ActionEvent e) {
        CancelButton_actionPerformed(e);
      }
    });

    InputPanel.setBorder(emptyBorder10pt);
    BasePanel.setMinimumSize(new Dimension(380, 185));
    BasePanel.setPreferredSize(new Dimension(500, 200));

DateAndAmountPanel.setBorder(BorderFactory.createCompoundBorder(BorderFactory.createEmptyBorder(4,2,0,2),etchedB
order));
    PayeeNameField.setColumns(20);
    getContentPane().add(BasePanel);
    BasePanel.add(InputPanel, BorderLayout.NORTH);
    InputPanel.add(NorthPanel, BorderLayout.NORTH);
    NorthPanel.add(Payee, null);
    Payee.add(PayeeLabel, null);
    PayeeLabel.add(PayeeNameLabel, null);
    PayeeLabel.add(PayeeAccountLabel, null);
    Payee.add(PayeeField, null);
    PayeeField.add(PayeeNameField, null);
    PayeeField.add(PayeeAccountField, null);
    NorthPanel.add(NotePanel, null);
    NotePanel.add(jScrollPane2, null);
    InputPanel.add(DateAndAmountPanel, BorderLayout.SOUTH);
    DateAndAmountPanel.add(DateBasePanel, null);
    DateBasePanel.add(DateLabel, null);
    DateBasePanel.add(DateFieldPanel, null);
    DateFieldPanel.add(DayField, null);
    DateFieldPanel.add(MonthField, null);
    DateFieldPanel.add(YearField, null);

    DateAndAmountPanel.add(AmountBasePanel, null);
    AmountBasePanel.add(AmountLabelPanel,null);
    AmountLabelPanel.add(AmountLabel, null);
    AmountBasePanel.add(AmountFieldPanel,null);
    AmountFieldPanel.add(AmountField, null);
    BasePanel.add(ButtonPanel, BorderLayout.SOUTH);
    ButtonPanel.add(OkButton, null);
    ButtonPanel.add(CancelButton, null);
    jScrollPane2.getViewport().add(NoteTextArea, null);
  }

  /*  setLanguage(int i)
   *  This method sets the language for the user interface.
   */
  public void setLanguage(int i) {
    language = i;
    setContents(language);
  }

  /*  setContents(int l)
   *  This method sets the content of the UI components according to the
   *  language setting.
   *  0 is english
   *  1 is finnish
   */
  private void setContents(int l){
    if(l==0) {
      OkButton.setText("OK");
      CancelButton.setText("Cancel");
      AmountLabel.setText("Amount:");
      DateLabel.setText("Due Date:");

      PayeeNameLabel.setText("Name:");
      PayeeAccountLabel.setText("Account:");

      DayField.setToolTipText("Day");
      MonthField.setToolTipText("Month");
      YearField.setToolTipText("Year");
      payeeBorder.setTitle("Payee");
      noteBorder.setTitle("Note");
    }
    else if(l==1){
       OkButton.setText("Hyväksy");
       CancelButton.setText("Peruuta");
       AmountLabel.setText("Määrä:");
       DateLabel.setText("Eräpäivä:");

       PayeeNameLabel.setText("Nimi:");
       PayeeAccountLabel.setText("Tilinumero:");

       DayField.setToolTipText("Päivä");
       MonthField.setToolTipText("Kuukausi");
       YearField.setToolTipText("Vuosi");
       payeeBorder.setTitle("Saajan tiedot");
       noteBorder.setTitle("Viesti");
    }else{System.out.println("Illegal language choice");}
  }

  /*  CancelButton_actionPerformed(ActionEvent e)
   *  This method disposes this dialog without saving any information.
   */
  void CancelButton_actionPerformed(ActionEvent e) {
    this.dispose();
  }

  /*  OkButton_actionPerformed(ActionEvent e)
   *  This method saves the bill data and disposes this dialog.
   */
  void OkButton_actionPerformed(ActionEvent e) {
    info.setPayee(PayeeNameField.getText());
    info.setPayeeAccount(PayeeAccountField.getText());
    info.setNote(NoteTextArea.getText());
    info.setAmount(java.lang.Integer.parseInt(AmountField.getText()));
    info.setInvoiceDate(java.lang.Integer.parseInt(YearField.getText()),
                    java.lang.Integer.parseInt(MonthField.getText()),
                    java.lang.Integer.parseInt(DayField.getText()));
    unpaidBills.addInvoice(info);
    fatherFrame.refreshTable(unpaidBills.getSize());
    this.dispose();
  }
}
```

Class MainFrame

```
//Title:        Banking application
//Version:      1.0
//Author:       Sami Rantala
//Company:      L M Ericsson Ltd.
//Description:  This is used to compare Swing and UIML interfaces.

package thesis.javabanking;

import java.awt.*;
import javax.swing.*;
import javax.swing.border.*;
import java.awt.event.*;
import javax.swing.JTable;
import javax.swing.ListSelectionModel;
import javax.swing.event.ListSelectionListener;
import javax.swing.event.ListSelectionEvent;
import javax.swing.event.TableModelListener;
import javax.swing.event.TableModelEvent;
import thesis.common.*;

public class MainFrame extends JFrame{
  JMenuBar menuBar = new JMenuBar();
  JMenu menu = new JMenu();
  JMenuItem menuItem;
  JPanel UserInformationPanel = new JPanel();
  JPanel FieldPanel = new JPanel();
  JPanel LabelPanel = new JPanel();
  JPanel UnpaidBillsPanel = new JPanel();
  JPanel SettingsPanel = new JPanel();
  JPanel ButtonPanel = new JPanel();
  JPanel TableButtonPanel = new JPanel();

  JTabbedPane TabbedPanel = new JTabbedPane();

  JScrollPane TableScrollPane = new JScrollPane();

  JTextField NameField = new JTextField();
  JTextField BalanceField = new JTextField();

  String[] accounts = {"1234-876","1234-124"};
  JComboBox AccountComboBox = new JComboBox(accounts);

  JButton Delete = new JButton();
  JButton CopyAsNewButton = new JButton();
  JButton NewButton = new JButton();

  JRadioButton englishLanguage = new JRadioButton();
  JRadioButton finnishLanguage = new JRadioButton();
  ButtonGroup group = new ButtonGroup();

  JLabel NameLabel = new JLabel();
JLabel BalanceLabel = new JLabel();
JLabel AccountLabel = new JLabel();

BorderLayout borderLayout2 = new BorderLayout();

GridLayout gridLayout1 = new GridLayout();
GridLayout gridLayout2 = new GridLayout();
GridLayout gridLayout8 = new GridLayout();

int language = 0;
int selectedRow = -1;

InfoCollection unpaidBills = new InfoCollection();
MyTableModel myModel = new MyTableModel(unpaidBills);
JTable UnpaidBillsTable = new JTable(myModel);
ListSelectionModel rowSM = UnpaidBillsTable.getSelectionModel();
JLabel jLabel1 = new JLabel();
JPanel TopPanel = new JPanel();
BorderLayout borderLayout3 = new BorderLayout();
JPanel contentPanel = new JPanel();
BorderLayout borderLayout4 = new BorderLayout();
Border EmptyBorder10pt;
Border UserInfoBorder;

String a= new String();
String b= new String();

/*  MainFrame()
 *  This is the constructor.
 */
public MainFrame() {
  try  {
  jbInit();
  this.pack();
  }
  catch(Exception e) {
    e.printStackTrace();
  }
}

/*  jbInit()
 *  This method initializes the UI.
 */
private void jbInit() throws Exception {
  EmptyBorder10pt = BorderFactory.createEmptyBorder(10,10,10,10);
  setJMenuBar(menuBar);
  TopPanel.setLayout(borderLayout3);
  contentPanel.setLayout(borderLayout4);
  contentPanel.setBorder(EmptyBorder10pt);
  UserInformationPanel.setBorder(EmptyBorder10pt);
  UnpaidBillsPanel.setBorder(EmptyBorder10pt);
  ButtonPanel.setBorder(EmptyBorder10pt);
  menuBar.add(menu);
  menuItem = new JMenuItem();

  menuItem.addActionListener(new java.awt.event.ActionListener() {
    public void actionPerformed(ActionEvent e) {
      menuItem_actionPerformed(e);
    }
  });

  menu.add(menuItem);
  UserInformationPanel.setLayout(gridLayout8);
  NameField.setEditable(false);
  FieldPanel.setLayout(gridLayout1);
  LabelPanel.setLayout(gridLayout2);
  gridLayout1.setColumns(1);
  gridLayout1.setRows(3);
  gridLayout1.setVgap(3);
  gridLayout2.setColumns(1);
  gridLayout2.setRows(3);
  gridLayout2.setVgap(3);
  this.setEnabled(true);
  BalanceField.setEditable(false);

  NewButton.addActionListener(new java.awt.event.ActionListener() {
    public void actionPerformed(ActionEvent e) {
      NewButton_actionPerformed(e);
    }
  });

  AccountComboBox.setSelectedItem(this);
  TableScrollPane.setDoubleBuffered(true);
  TableScrollPane.setPreferredSize(new Dimension(300, 200));
  TabbedPanel.add(UnpaidBillsPanel, "UnpaidBillsPanel");
  TabbedPanel.add(SettingsPanel, "SettingsPanel");
  UnpaidBillsPanel.setLayout(borderLayout2);

  Delete.addActionListener(new java.awt.event.ActionListener() {
    public void actionPerformed(ActionEvent e) {
      Delete_actionPerformed(e);
    }
  });

  CopyAsNewButton.addActionListener(new java.awt.event.ActionListener() {

    public void actionPerformed(ActionEvent e) {
      CopyAsNewButton_actionPerformed(e);
    }
  });

//    UnpaidBillsTable.setShowHorizontalLines(false);
  UnpaidBillsTable.setAutoResizeMode(UnpaidBillsTable.AUTO_RESIZE_ALL_COLUMNS);
  this.getContentPane().add(contentPanel, BorderLayout.CENTER);
  contentPanel.add(TopPanel, BorderLayout.NORTH);
  TopPanel.add(UserInformationPanel, BorderLayout.WEST);
  UserInformationPanel.add(LabelPanel, null);
  LabelPanel.add(NameLabel, null);
  LabelPanel.add(AccountLabel, null);
  LabelPanel.add(BalanceLabel, null);
  UserInformationPanel.add(FieldPanel, null);
  FieldPanel.add(NameField, null);
  FieldPanel.add(AccountComboBox, null);
  FieldPanel.add(BalanceField, null);
  TopPanel.add(ButtonPanel, BorderLayout.EAST);
  ButtonPanel.add(NewButton, null);
  contentPanel.add(TabbedPanel, BorderLayout.CENTER);
  UnpaidBillsPanel.add(TableScrollPane, BorderLayout.NORTH);
  UnpaidBillsPanel.add(TableButtonPanel, BorderLayout.SOUTH);
  TableButtonPanel.add(CopyAsNewButton, null);
  TableButtonPanel.add(Delete, null);

  englishLanguage.setActionCommand("english");
  finnishLanguage.setActionCommand("finnish");
  group.add(englishLanguage);
  group.add(finnishLanguage);
  englishLanguage.setSelected(true);

  englishLanguage.addActionListener(new java.awt.event.ActionListener() {
    public void actionPerformed(ActionEvent e) {
      radioButton_actionPerformed(e);
    }
  });

  finnishLanguage.addActionListener(new java.awt.event.ActionListener() {
    public void actionPerformed(ActionEvent e) {
      radioButton_actionPerformed(e);
    }
  });

  SettingsPanel.add(englishLanguage,null);
  SettingsPanel.add(finnishLanguage,null);
  TableScrollPane.getViewport().add(UnpaidBillsTable, null);

  rowSM.addListSelectionListener(new ListSelectionListener() {
    public void valueChanged(ListSelectionEvent e) {
     //Ignore extra messages.
     if (e.getValueIsAdjusting()) return;
     ListSelectionModel lsm = (ListSelectionModel)e.getSource();
     if (lsm.isSelectionEmpty()) {
       //no rows are selected
       selectedRow = -1;
     } else {
       selectedRow = lsm.getMinSelectionIndex();
       //selectedRow is selected
       }
     }
     });
   setContents(getLanguage());
 }

 /*  setContents(int i)
  *  This method sets the content of the UI components.
  *  int i is the language code
  *  0 is english
  *  1 is finnish
  */
 private void setContents(int i){
 if(i==0){
   menu.setText("File");
   menuItem.setText("New Bill");
   CopyAsNewButton.setText("Copy as New Bill");
   BalanceLabel.setText("Balance:");
   AccountLabel.setText("Account:");
   Delete.setText("Delete");
   BalanceField.setText("1052");
   NewButton.setText("New bill");
   NameLabel.setText("Name:");
   NameField.setText("Jack Banker");
   jLabel1.setText("This space is intentionally left blank");
   this.setTitle("Banking application");
   TabbedPanel.setTitleAt(0, "Unpaid Bills");
   TabbedPanel.setTitleAt(1, "Settings");
   englishLanguage.setText("English");
   finnishLanguage.setText("Finnish");
   a = "New Bill";
   b = "New Bill(Copied from old)";

   } else {
     menu.setText("Tiedosto");
     menuItem.setText("Uusi lasku");
     CopyAsNewButton.setText("Kopioi uudeksi laskuksi");
     BalanceLabel.setText("Saldo:");
     AccountLabel.setText("Tilinumero:");
     Delete.setText("Poista");
     BalanceField.setText("1052");
     NewButton.setText("Uusi lasku");
     NameLabel.setText("Nimi:");
     NameField.setText("Jack Banker");
     this.setTitle("Pankki sovellus");
     TabbedPanel.setTitleAt(0, "Maksamattomat laskut");
     TabbedPanel.setTitleAt(1, "Asetukset");
     englishLanguage.setText("Englanti");
     finnishLanguage.setText("Suomi");
     a = "Uusi lasku";
     b = "Uusi lasku (kopio)";
   }
 }

 /*  refreshTable(int i)
  *  This method fires an event that updates the table.
  */
 public void refreshTable(int i) {myModel.fireTableRowsInserted(i,i);}

 /*  getLanguage()
  *  This method returns the language code.
  *  0 is english
  *  1 is finnish
  */
  public int getLanguage(){return language;}

  /*  NewButton_actionPerformed(ActionEvent e)
   *  This method opens a dialog for creating new bill.
   */
  void NewButton_actionPerformed(ActionEvent e) {
    Invoice newBill = new Invoice(this, a , true, unpaidBills);
    newBill.setLanguage(language);
    newBill.setVisible(true);
  }

  /*  Delete_actionPerformed(ActionEvent e)
   *  This method deletes selected bill from the table.
   */
  void Delete_actionPerformed(ActionEvent e) {
    if(selectedRow != -1){
      unpaidBills.deleteRow(selectedRow);
      myModel.fireTableRowsDeleted(selectedRow,selectedRow);
      selectedRow = -1;
    }
  }

  /*  CopyAsNewButton_actionPerformed(ActionEvent e)
   *  This method copies the selected bill's information into the
   *  new bill dialog.
   */
  void CopyAsNewButton_actionPerformed(ActionEvent e) {
    System.out.println(selectedRow);
    if(selectedRow != -1){
    Invoice copiedBill = new Invoice(this, b, true,unpaidBills, unpaidBills.getInvoice(selectedRow));
    copiedBill.setLanguage(language);
    copiedBill.setVisible(true);
    }
  }

  /*  menuItem_actionPerformed(ActionEvent e)
   *  This method forwards alls calls to NewButton_actionPerformed(e).
   */
  void menuItem_actionPerformed(ActionEvent e) {
    this.NewButton_actionPerformed(e);
  }

  /*  RadioButton_actionPerformed(ActionEvent e)
   *  This method changes the language of the UI according the radio buttons.
   */
  void radioButton_actionPerformed(ActionEvent e){
    if(e.getActionCommand().compareTo("english") == 0){
      language = 0;
      setContents(0);
      myModel.setLanguage(0);
      myModel.fireTableStructureChanged();
      UnpaidBillsTable.sizeColumnsToFit(UnpaidBillsTable.AUTO_RESIZE_ALL_COLUMNS);

    }
    if(e.getActionCommand().compareTo("finnish") == 0){
      language = 1;
      setContents(1);
      myModel.setLanguage(1);
      myModel.fireTableStructureChanged();
      UnpaidBillsTable.sizeColumnsToFit(UnpaidBillsTable.AUTO_RESIZE_ALL_COLUMNS);
    }
  }

}
```