DIPLOMARBEIT

# Automated Graphical User Interface Generation based on an Abstract User Interface Specification

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines
Diplom-Ingenieurs unter der Leitung von

Univ.Prof. Dipl.-Ing. Dr.techn. Hermann Kaindl
und
Univ.Ass. Dipl.-Ing. Dr.techn. Jürgen Falb
als verantwortlich mitwirkendem Assistenten am
Institutsnummer: 384
Institut für Computertechnik

eingereicht an der Technischen Universität Wien
Fakultät für Elektrotechnik und Informationstechnik

von

David Raneburger
Matr.Nr. 0125896
Forsthausgasse 16/2/7702, 1200 Vienna

17.12.2008

## Kurzfassung

Bei der Entwicklung neuer Software, bzw. dem raschen Erstellen von Prototypen, wird der Programmierer heute vielfach durch Tools, die eine teilweise automatische Generierung des Quellcodes ermöglichen, unterstützt. Durch den vermehrten Einsatz von graphischen Anzeigen bzw. Bildschirmen, basiert die Kommunikation zwischen System und Benutzer häufig auf einem Graphischen User Interface (GUI), vielfach auf einem sogenannte WIMP (window, icon, menu, pointer) UI. Ein WIMP UI besteht aus einer beschränkten Anzahl von *Widgets*, die je nach der zu erfüllenden Aufgabe kombiniert werden.

Im Zuge dieser Diplomarbeit wird ein Code Generator präsentiert, mit dessen Hilfe eine abstrakte User-Interface Spezifikation in Quellcode für ein WIMP UI übersetzt wird. Das als Ausgangspunkt dienende abstrakte Model ist nicht auf ein bestimmtes *GUI-toolkit* festgelegt, sehr wohl aber auf ein bestimmtes Zielgerät (z.B. PDA, Bildschirm, . . . ). Die Transformation des Modeles zu Quellcode wurde mittels *Templates* realisiert und Java Swing als *GUI-toolkit* gewählt.

Die Implementierung des Code Generators hängt von der Meta-Model Struktur der abstrakten UI-Spezifikation ab und ist deshalb auf die Transformation von Instanzen des Meta-Modells beschränkt. Durch die Verwendung von *Templates* kann die Anzahl der untertützten *GUI-toolkits* jedoch ohne größeren Aufwand erhöht werden.

## Abstract

Automated code generation tools support rapid prototyping and the development of software in various fields of application. As almost any kind of technological device is equipped with either a screen or a display, system-user communication frequently relies on graphical user interfaces, in the majority of cases on WIMP (window, icon, menu, pointer) UIs. Such interfaces consist of a limited number of widget types. What varies, due to the task that should be accomplished, is the way they are combined.

The generator presented in this work translates an abstract description of a graphical user interface into source code that implements a WIMP UI. The abstract UI models forming the inputs are independent of the target GUI toolkit, but not of the target device. The actual translation from model to target toolkit specific code involves the use of templates. The code generator has been implemented and tested, using Java Swing as the target toolkit.

The structure of the code generator depends on the meta-model definition of the abstract user interface specification. Therefore, the generator is not applicable to any other kind of input model. Due to the use of templates however, the number of supported target toolkits can be extended without much effort.

**Acknowledgements**

Da ich privat sehr gerne Bergsteige, freut es mich umso mehr, auch im übertragenen Sinn auf einem Höhepunkt des Weges angekommen zu sein, den ich vor sieben Jahren zu gehen begann. Die hier vorliegende Diplomarbeit stellt das Ende eines Studiums dar, das ich ohne der Hilfe und Motivation vieler Freunde und vor allem meiner Familie nie absolviert hätte.

Mein erster Dank gilt deshalb auch meinen Eltern und Geschwistern, die nicht müde wurden mich zu motivieren und sowohl finanziell als auch moralisch zu unterstützen. Sollte ich Erfolg haben, ist eure Pension gesichert, gern hab ich euch sowieso!

Ein jeder Gipfel ist mit fachkundiger Führung leichter zu erreichen und in diesem Sinne möchte ich mich vor allem bei meinen Betreuern Prof. Hermann Kaindl und Dr. Jürgen Falb recht herzlich bedanken. Ein großes Dankeschön auch an Roman Popp und Sevan Kavaldjian für die vielen wertvollen Tipps, eure Geduld und die gute Zusammenarbeit. Es freut mich sehr, euch seit Oktober Arbeitskollegen nennen zu dürfen.

Des weitern bedanken möchte ich mich bei allen Mitstudenten, die mir nicht nur als Mitstreiter bei Prüfungen, sondern im Laufe der Jahre vor allem als Freunde eine wertvolle Hilfe waren und sind. Nicht zu vergessen all jene Freunde und Innen, die mich, ganz unabhängig vom Studium, mit Motivation und Energie versorgt haben.

Nachdem zumindest meine Urlaubstage inzwischen gezählt sind hat, so wie es aussieht der Ernst des Lebens begonnen... und es macht Spaß!

David

# Contents

# Abbreviations and Acronyms

| | |
|---|---|
| **API** | Application Programming Interface |
| **AST** | Abstract Syntax Tree |
| **CDL** | Communication Definition Language |
| **CSS** | Cascading Style Sheets |
| **DOM** | Document Object Model |
| **EMF** | Eclipse Modeling Framework |
| **EMOF** | Essential Meta Object Facility |
| **GUI** | Graphical User Interface |
| **HTML** | Hypertext Markup Language |
| **JET** | Java Emitter Templates |
| **MDA** | Model Driven Architecture |
| **MVC** | Model View Controller |
| **OntoUCP** | Ontology-based Unified Communication Platform |
| **SWT** | Standard Widget Toolkit |
| **VTL** | Velocity Template Language |
| **W3C** | World Wide Web Consortium |
| **XMI** | XML Metadata Interchange |
| **XML** | Extensible Markup Language |
| **XSLT** | Extensible Stylesheet Language Transformations |
| **WIMP** | Window, Icon, Menu, Pointer |

# Chapter 1

# Introduction

With an increasing number of computer-based systems in our environment, a device's ability not only to interact with other devices but with humans as well becomes more and more important. Up to now describing machine-machine interaction requested profound technological knowledge. Widespread protocols are based on a simple request-response principle or on message passing metaphors. Building such systems would be much easier if higher semantics were introduced in communication protocols in a way that would make the interaction more human-like and therefore "natural". Such an interaction description would enable people with only limited technological knowledge to understand such a system easily and even to design it, with the help of a graphical editor for example.

Making the interaction mechanism independent from the other communication party involves the advantage that a device does not have to know whether it talks to a machine or a person. This unification of machine-machine interaction and human-machine interaction would make the inclusion of a system in an ubiquitous computing environment much easier.

Such a system can only be built, relying on the results of communication theoretical research. The goal of the Ontology-based Unified Communication Platform project is to take a model-driven approach to generate the communication component for a whole system, including the graphical user interface, automatically. The interaction between user and system is modelled using Communicative Acts [Sea69], linked via Rhetorical Speech Theory (RST) Relations [MT88] and Adjacency Pairs [LGF90].

Communicative Acts are based on Searle's theory, which states that speech is used to do something with a certain intention. A question is asked to obtain information, a command is given to make someone do something, a statement is made to convey information and so on. These acts can be seen as basic units of a communication and, therefore, offer a precise way to describe not only human communication but also communication between machines or any other type of communication between two parties. To model the flow of an interaction these Communicative Acts need to be linked somehow. This can be done using another linguistic theory.

The Rhetorical Structure Theory (RST) has its focus on the function of text. It offers a comfortable way to describe internal relationships between text-potions, their effects and associated constraints. Such relations can be a Condition, Background Information or a Joint. The objects linked by such relations are communicative acts. They encapsulate a certain intention each and form the basic building blocks for any discourse. Adjacency Pairs have been introduced as well to complete the meta-model for discourses. This theory describes typical turns in communications, which result in typical pairs of communicative acts within the model.

The combination of these three theories allows the description of a communication as a tree. The communicative acts represent the leaves that are linked by RST relations. Diagrams are used as a graphical representation of the communication (see Chapter 3).

Based on the above given theories, communication between a computer system and a user or even between two machines can be described. Considering the first type of interaction one more problem to be considered is how to interact with a user. One of the most established ways to display information and to communicate with a user is to do so via a graphical user interface. Up to now such interfaces offer the most "intuitional" way of interaction with a system. Nowadays almost all, even portable devices, contain a screen, differing only in the platform they use to display the graphical information. Two of the most widespread underlying technologies, due to their platform independence, are HTML and Java Swing.

Programming such an interface is still a time consuming and therefore expensive task. Automating the generation of the graphical user interface source-code would, therefore, save a lot of money and moreover ease the maintenance of the system as well. As soon as something concerning the structure of the system changes, it would not be necessary to look for changes to be made in the user interface but only to generate it anew. What would be needed as an input to the code-generator is an abstract description of the user interface, independent of the target toolkit but not of the target device. Issues like screen resolution must be considered right from the beginning to find an expedient representation of the information that needs to be conveyed or gathered by the system.

Using a model-driven approach, the intention is to generate the graphical user interface, out of the discourse description, in two steps. At first an abstract user interface is generated. In this step the discourse model is transformed into widgets, considering the screen resolution of the target platform to find the right layout. The second step is the transformation of this abstract user interface specification into a specific language like Java Swing. This target language needs to be selected by the user in advance. The resulting code contains everything needed by the system during runtime.

These two steps are hidden from the designer, who does not know how the transition between discourse model and concrete user interface happens. Everything the designer has to do is define the discourse model, select a target device and toolkit, push the generate button which triggers the automatic generation of the source code for the graphical user interface, including all the runtime components.

The topic of this master thesis is the transition from an abstract user interface specification to generated source code files. Before the details on the developed device are presented, an outline concerning common code generation techniques is given. The second half of Chapter 2 introduces four widely used template engines, including Java Emitter Templates (JET), which have been used to implement the code generator. The subsequent chapter has its focus on the OntoUCP project and provides the information necessary to fully understand Chapter 4. Within this section the code generator, which has been implemented using JET, is presented in detail. Finally the pros and cons of the code generator implementation are pointed out and the essence of the presented work is repeated. The last chapter gives an outlook on future development as well as on the deployment of the developed technology.

# Chapter 2

# State of the Art

This chapter presents the most common techniques to generate source code automatically and reasons why it makes sense to do so. As templates have been used in the code generator implementation (presented in chapter 4) an overview over the most widely spread template engines is given. The Eclipse Modeling Framework is presented at the end of this chapter because major parts of the OntoUCP project rely on this framework.

## 2.1 Automated Code Generation

The model-driven approach is a promising technique to design software today. This signifies that systems are not developed writing source code right from the beginning, but describing the software's design on a higher level of abstraction. Models play an important role within the OntoUCP project, but instead of translating them to code directly they are transformed to another model first. Subsequently the code generator, which only depends indirectly on the actual input models, completes the model-to-code transformation.

One of the main reasons for using a code generator is to guarantee a certain performance without losing much flexibility. As soon as traditional object-oriented approaches like frameworks, polymorphism or reflection are not sufficient to reach the desired performance the actual configuration can be set in an abstract way and more efficient code is generated automatically.

Another point of consideration might be the size of the code. If the features needed during the runtime of the system are already known in advance, the generator only needs to include these parts at compilation time. This way the size of the code can be reduced efficiently.

Complex generic frameworks or runtime systems frequently work interpretive, which complicates the static analysis of program characteristics as well as the search for faults. Regarding this aspect, automatically generated code, using a standard programming language possesses the quality of classically written code. Many systems can only be realised combining manually written and automatically generated code, which requires the designer to find the right balance.

Weak type definitions are frequently used to enable the program to make decisions only during runtime. This increases the flexibility but transfers fault detection from compile to runtime as well, thus enlarging the difficulty of finding errors. Using static frameworks for code generation includes the advantage of being able to find errors already at compilation time, giving the compiler the possibility of uttering warnings as well.

One of the big benefits of using a model-driven approach in software development is the independence of the system architecture and logic from the actual implementation platform. This way the transition to newer or potentially better platforms can be achieved easily.

Another issue to point out would be that shortcomings of the actually used programming language can be compensated. Examples concerning Java would be its type genericity (at least for version 1.5 downwards) or the downcast to one variable class. Moreover, code generation eases the introduction of object-oriented concepts in procedural programming languages.

Finally introspection should be mentioned. It describes a program's read access to itself. Some programming languages support this mechanism dynamically (e.g. Java) and some do not support it at all (e.g. C++). If this feature is not supported dynamically, code generation can at least introduce this mechanism statically. Instead of analysing a program's structure and connecting it with a certain functionality during runtime, this task is done before hand. Based on the analysis of the system, code is generated that activates the requested functionality during the runtime of the system.

Stahl and Voelter [SV05, p. 175, p. 190] distinguishes the code generation techniques presented in the following paragraphs. These techniques differ strongly in many aspects but all of them do have at least three things in common:

1. There is always a meta-model or an abstract syntax (at least given implicitly).
2. There are always transformations based on this meta-model.
3. There is always some kind of frontend which reads the meta-model and triggers its transformation.

### 2.1.1 Templates and Filtering

This technique represents a very simple and direct access to code generation. The code is generated using templates that iterate over the relevant parts of a textually represented model (e.g. XSLT iteration over XML). The code to be generated is provided by the templates and the variables contained by the templates are bound to values provided by the model. The same model can easily be translated to various target platforms by using different templates.

XSLT is a typical language used to implement this technique in combination with XMI or XML. Therefore, this example is used to illustrate a typical drawback of this technology. The fact that the XSLT-Stylesheet's complexity increases with the amount of code to be generated makes this technique inapplicable for big systems, even more so if their specification is based on XMI. To resolve the XMI problem, a model-to-model transformation can be done before the actual code generation. This first step converts the XMI description to a domain-specific XML representation. Further steps generate the source code based on this XML description. This eases the code generation a lot because it separates the templates from the actual XMI syntax. The level of abstraction the generator works on is still the level of the XML meta-model. This problem can be solved by following the next generation approach.

### 2.1.2 Templates and Meta-Model

This technique solves some of the problems arising through direct code generation by introducing an approach that includes more than one step. The XML description is parsed by the generator in a first step. Subsequently an instance of the meta-model is created, which can be customized by the user. This model then forms the basis on which the actual output code is generated. The whole process is illustrated by Figure 2.1.

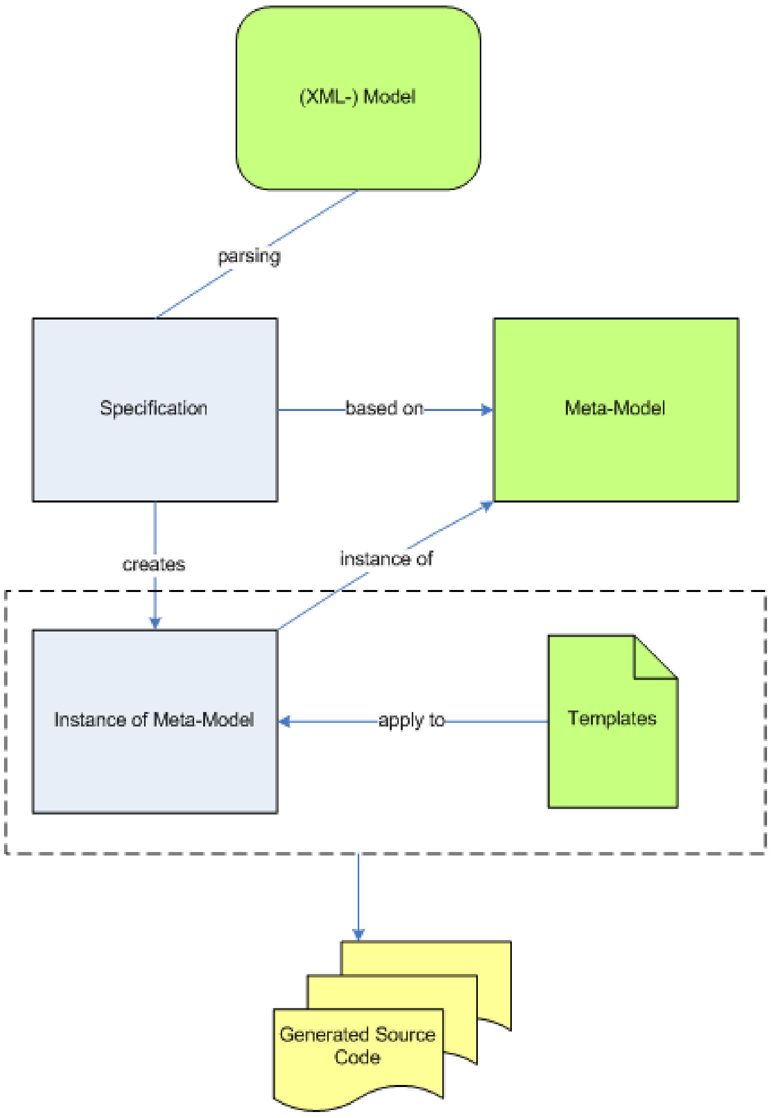


**Figure 2.1:** Templates and Meta-Model

An advantage of this approach is that the generator module is more or less independent from the concrete model syntax. This is due to the insertion of the parser, which itself is syntax-dependent. Frequently, the description is provided either in UML or any available XMI version.

Another benefit of this technique is that complex logical verification functions regarding the meta-model can be included in the meta-model itself. In contrast to templates, such functions can be implemented in common programming languages like Java.

This technique has been used in the implementation of the openArchitectureWare generator, which includes another aspect worth highlighting. From a compiler constructor's point of view, the transformation consists of the meta-model implementation (e.g. in Java) and the templates. The meta-model's function is once again the one of an abstract syntax. This syntax as well as the transformation are part of the compiler's parameters. The main issue is that these abstract syntax constructs (i.e. the elements of the meta-model) translate themselves. Therefore the compiler is called object oriented.

Taking a conceptual point of view, the templates and the support functions implemented in Java, are means for translating the meta-model elements. Just like Java, the template language supports polymorphism and overwriting. Only the helper class definition is outsourced to the Java part. This is the reason why template languages are called object-oriented.

### 2.1.3 Frame Processing

The basis of this generation technique is frames. These frames contain the specification of the code to be generated and are processed by the frame processor.

Just like classes in object-oriented languages, these frames can be instanced. Every frame instance, more than one of the same frame is possible, binds the variables, in this case called slots, to concrete values. Every instance can be bound to its own set of values. Such values can be basic data types like `String` or `Double` or other frame instances. The resulting structure of the code to be generated is then represented by a tree that consists of the instantiated frames. In a second step, after the frames have been created, the code is generated according to the given tree structure.

### 2.1.4 API-based generation

The concept of this kind of code generator is to provide the programmer with an output-language-specific Application Programming Interface (API). This interface enables the programmer to create elements of the target platform. The abstract syntax the generator works with is defined by the meta-model of the target language. Therefore, the generator is bound to one specific target platform or target languages that share the same abstract syntax tree.

This mechanism is used by the .NET framework, which allows varying the output language through the change of the generator backend. Abstract syntax trees can be specified by creating a Code Document Object Model (CodeDOM). In .NET this model is based on the abstract syntax defined by the Common Language Specification (CLS). Through the selection of an appropriate ICodeGenerator, the concrete syntax for a chosen .NET language (e.g. C#, VB, C++) can be generated.

This technique offers a intuitive and easy way to generate code. Another advantage is that the compiler in combination with the API can force the user to write syntactically correct code quite easily. A slight drawback compared to using templates is that variable values need to be set by using a common method. This process requires more code to be written than if the values simply replace specified text parts.

Such generators can be used much more efficiently if domain-specific classes are used together with default types. Such classes can be derived by the generator through common object-oriented concepts. Flexibility can be reached through parameterisation of the generator class.

### 2.1.5 Inline Generation

Inline code generation stands for source code that contains constructs which generate further source, byte or machine code when compiled. Examples are C++ pre-processor statements or C++ templates.

There are few cases in which the use of such pre-processor statements is reasonable because, as they are based on simple text substitution, no type checks or preference rules are included. Templates, on the other hand, form a Turing-complete, functional programming language based on types and literals. This enables the user to create complete programmes that are executed at compile time.

Due to the unintuitive syntax, this technique is not in widespread use. It is not advisable to create large generation projects using this technique, because most compilers have not been optimised for this kind of task. See [CE00] for further details on this topic.

### 2.1.6 Code Attributes

This mechanism is widely known in Java environments and started out as JavaDoc. This tool allows the automatic generation of a HTML-documentation of the program via added comments. Due to JavaDoc's extensible architecture, custom tags and code generators can be hooked easily. The best known generator using this technique is XDoclet [Tea05], which lost significance since Java introduced similar features directly in version 1.5.

Using this technique the developer only writes the interface or class definition manually and attaches the commentary needed for the generator. Apart from the commentary, the generator accesses the source code syntax tree to extract additional information. The advantage of this method is that much information needed by the generator is already provided through the source code. The programmer can save a lot of work by adding commentaries. Typical applications would be the generation of EJB Remote/LocalInterfaces as well as Deployment Descriptors.

The same mechanism is deployed by .NET, which offers the attachment of attributes to source code elements like methods, attributes or classes. Apart from automatic code generation, runtime functionality can be added this way. Such "realtime" framework characteristics can be illustrated best by introducing a small example. Every class is given a certain priority in advance. Subsequently, every instance of such a class is equipped with such an attribute and every method of such a class carries a time limit within which it needs to complete its task. The idea is that services are executed within a framework that measures certain parameters (e.g. the execution time). If this limit is exceeded, a log entry can be made, no more requests can be accepted and an exception can be thrown to the user.

### 2.1.7 Code Weaving

This term names the technique of merging different independent code snippets. The definition of how to integrate the different parts is given by so-called Join Points or Hooks.

An implementation of this principle is AspectJ [Fou08b], which is available as Eclipse plugin as well. This generator combines regular object-oriented code with so-called aspects on source or byte code level. In this context the word aspects is used to describe functionalities that solve the problem of cross-cutting concerns. Such cross-cutting concerns are connections that can not be described locally using common object-oriented means. An example would be an aspect offering logging of calls on methods of a certain class. This aspect logs for example the name of the class as well as the name of the method that calls methods on the specified class.

### 2.1.8 Combinations

To reach the best performance for a given task, various combinations of the generation techniques presented above can be deployed. An example would be the generation of annotated source code using a template engine. Another frequently found case is the combination of code that was generated automatically, customized manually and now needs to be merged with new automatically generated code. Another possibility is to create API-based generators that offer the option to include templates.

### 2.1.9 Differences and Similarities

Generators that build an Abstract Syntax Tree (AST) of the application domain are frame processors and API-based ones. To raise the level of abstraction in contrast to pure template-based approaches and the efficiency, the addition of platform-specific constructs is quite popular (e.g. a Frame that creates a JavaBeans-Property). This way a general approach is tailored to a certain target platform. Due to this reason the starting point for such applications is normally the AST of the target language or platform.

The Templates and Meta-Model approaches, on the contrary, build the AST during runtime. This task is completed by the generator, relying on the meta-model as representation of the problem domain. Therefore, this approach starts on a higher level of abstraction and the translation to a certain target platform is completed by the templates. The deployment of this technique is widely used if a comprehensive meta-model of the application domain is available. Model driven architecture usually results in the use of this technique. Regarding the amount of similar code to be generated, this approach is normally preferred as well to the API-based generator. These generators are quite efficient on the other hand, if the granularity of the code to be generated is very fine.

Code attributes can be seen as a kind of inline generation as well. This would be the case if the code is generated where the specification in the base source code is found. Normally, attributes are used to generate code that is completely extern. As it covers mostly persistency or glue logic aspects, it does not need to be integrated with manually written code. These approaches are not appropriate if models are used, but can be quite convenient if already available code can be used as a basis. This way only attributes and specifications have to be added.

The difference between inline generation and code weaving is that cross-cutting concerns can be included more easily by the latter one. These aspects allow the modification of the existing code from the outside. Both technologies are appropriate if code instead of models forms the basis worked on.

## 2.2 Template Engines

As the tasks computer programs have to accomplish get more and more complex, the programs themselves get more complicated as well and the amount of code increases. Modelling languages like UML have been developed to support the designer, giving him an abstract way of illustrating the most important aspects of a system. As UML is a general modelling language it gives the designer the opportunity to concentrate on the design and the content of a system instead of worrying about the concrete technical realisation.

In a second step, these models need to be translated to source code of course. As this step involves tedious programming, it was soon the intention of programmers and scientists to automate this step. There is a wide range of applications offering the construction of UML models and as a second step the automated generation of class declarations and their associated method declarations. This generated code offers the skeleton of the system to the programmer which then needs to be filled with code to accomplish the given task. To obtain already executable code the amount of information contained by the model needs to be augmented or added some other way.

An established way to produce runable code is the use of template engines. By combining templates and additional information, they produce source code that only needs to be compiled and executed as Figure 2.2 illustrates. The nature of the output is determined by the templates and the additional information is frequently extracted from a detailed model of the system the code shall implement in the end.

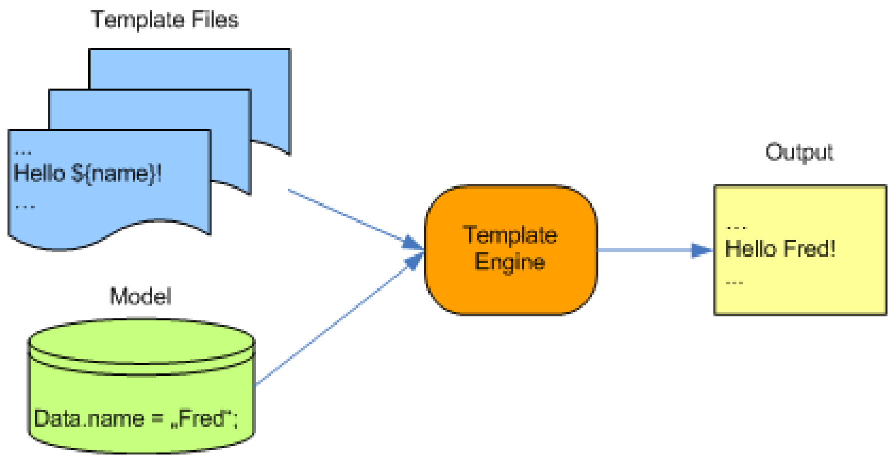


**Figure 2.2:** Template Engine

Automation of repeating steps eases the development of new systems. Taking a closer look at any source code file, it becomes obvious that the structure, apart from a component's source code, is always quite similar. Another important advantage that comes with the use of templates is that the amount of code to be written manually is reduced drastically. Furthermore the error rate of the resulting code is lower because only the templates need to be checked for faults instead of the whole program. One more thing that's worth to focus on is the appliance of changes. If either the model or the template is adjusted to fit new circumstances, the program does not have to be rewritten but only to be generated anew.

Today, various template engines are available for software engineers. Most of them are published under a public license and can either be used as off-the-shelf component or be customized to a user's special needs. Usually a reduced syntax set is offered, which allows the user to navigate through the input model described in XML or any other data description language. The requirements on such a template language are quite similar to the requirements on a programming language. The difference is that additionally to performance and ease of use it is important that the template language gives you an idea of what the output will look like even before the generation. This feature is frequently supported through the editor used to create the templates by providing some kind of color coding. The underlying of the template code with different colors helps the programmer to distinguish which code is executed during the time of generation and which code forms the resulting output.

### 2.2.1 Velocity

This Java-based template engine introduces a simple, yet powerful template language which allows the user to reference objects defined in Java code. The template writer includes markup statements called references in his file, which are filled with content during the rendering process. This content is stored in a context object, which basically consists of a hashtable. To set or retrieve values from the context object *get()* and *set()* methods are provided. These references do not only allow one to link properties but also to call methods from a given Java class. Apart from substituting the markups with the actual values the Velocity Template Language (VTL) offers basic control statements. A Foreach command can be used to loop over a set of values and if/else introduces the possibility to bind the inclusion of a text to a condition.
The #include directive allows the designer to include files containing text. The content of files included this way is not rendered by the template engine beforehand. If the given file needs to be rendered as well it must be included using the #parse directive. A #macro script element offers the opportunity to define a repeated segment of the VTL document. For a complete documentation of the VTL Syntax refer to [Fou08a].

The features presented above make Velocity not only attractive to Java programmers but to Web engineers as well. Main application fields of Velocity are:

- **Web applications:** In this case VTL directives act as place holders for dynamic information in a static HTML environment. To process such pages VelocityViewServlet or any framework that supports Velocity can be used. This approach results in a Model-View-Controller (MVC) architecture, which is intended to provide a viable alternative to applications built with Java Server Pages (JSP) or PHP.
- **Source code generation:** Seen as a standalone component, Velocity can, of course be used to generate Java source code, SQL or PostScript. See [Fou08a] for details on open source and commercial software packages that use Velocity this way.
- **Automatic emails:** Frequently, automatically generated emails are used as signup, password reminders or reports sent on a regular basis. Instead of including the email directly in the Java source code, Velocity offers the opportunity to store it in a separate text file.
- **XML transformation:** Anakia is an ant task provided by Velocity which makes an XML file available to a Velocity template. A frequently needed application is the conversion of a documentation stored in a generic "xdoc" format to a styled HTML document.

If the features already provided by Velocity are still insufficient to resolve a given problem, various possibilities are offered to extend its capabilities:

- **Tools:** They are a special type of object which contains methods instead of data. Placed in a Velocity context the template can call these methods. Such objects are frequently used to format numbers or escape HTML entities.
- **Resource Loaders:** If templates need to be retrieved from text files, the classpath, a database or even a custom resource, such loaders can be used.
- **Event Handlers:** In case of certain events, the Event Handler performs a specified custom action. An example would be the insertion of a reference into a text.
- **Introspectors:** To retrieve reference properties and methods, a custom Introspector has to be written. An application would be to retrieve data from Lucene or other search engine indexes.
- **Extend the Velocity Syntax:** The grammar of Velocity is processed in a parser generated by JavaCC (Java Compiler Compiler). The JJTree extension is used to create an abstract syntax tree. To change the Velocity syntax itself, only the JavaCC specification file has to be changed and a recompilation has to be done.

### 2.2.2 FreeMarker

Just like Velocity, FreeMarker can be used to generate text output based on templates. As it comes as a Java package, a class library for programmers, it is no application for the end user but needs to be embedded. It is quite easy to integrate as it does not require servlet environment.

FreeMarker has been developed to support the construction of HTML Web pages with dynamic content. Once more the MVC pattern has been implemented to enable everybody to work on what he is good at. This way the appearance of a Web page can be changed easily without having to recompile the code needed for the dynamic features.

Another advantage is that the templates used for the design are kept clean from complex code fragments and therefore easily maintainable. FreeMarker offers some programming capabilities as well, but the main data preparation is done by Java programs, leaving only the generation of the textual pages that display the data to the template engine.

Templates to be included or rendered can be loaded from any sources (e.g. local files or database) via a pluggable template loader. The template engine can produce any kind of textual output ranging from HTML, XML to RTF and any kind of source code. The output text can either be stored as a file, sent as email, sent back to a Web browser from a Web application etc.

As any template engine, FreeMarker offers its own template language. It consists of all usual directives (e.g. include), conditions like “if/elseif/else” and “loop” constructs. It allows the creation and changing of variables as well as the execution of a number of operations on them. These operations include `String` operations, `Boolean` arithmetic, decimal precision arithmetic calculations, reading array and associative array elements and the possibility of defining custom operations within self-written methods. The macro directives can be endued with named and positional parameters and modularization is supported by providing namespaces. This feature helps avoiding name clashes and eases the maintainability of macro libraries or big projects. Transformations like HTML escaping or syntax highlighting on the output is supported by the template engine via output transformation blocks, which can be customized as well by the programmer.

To ease the use of the Java objects providing the data needed to fill the template, FreeMarker exposes these objects to the template through a pluggable object wrapper class. This way complicated technical details can be hidden and only the information relevant for the template author is shown. Compared to Velocity, its template language is more complex but therefore more powerful. For detailed comparison of these two template engines see [Pro08].

As said before, FreeMarker has been designed to support the MVC pattern and offers a template language with built-in constructs that make it especially suitable for Web applications. Currently it supports JSP taglibs up to JSP 2.1 and can be integrated in Model2Web frameworks as direct JSP replacement. Contemplating the multinational character of the Web features like locale sensitive number, date and time formatting alongside with char set awareness and the possibility of using non-US characters in identifiers are very useful.

Since version 2.3 the directives <#recurse> and <#visit> enable the engine to traverse XML trees recursively. This version is also intended as an alternative to eXtensible Style Sheet Language Transformations (XSTL). Editor plugins offering at least syntax highlighting are available for the most common editors and IDEs.

### 2.2.3 AndroMDA

As suggested by the name, this extensible generator framework adheres to the Model Driven Architecture (MDA) paradigm. AndroMDA itself is basically a transformation engine which offers the possibility to directly transform UML models into deployable components. Various standard platforms like Spring, EJB 2/3, Hibernate, Java or XSD are supported and an Eclipse integration should be available soon. For each such transformation, a ready-made cartridge is provided which only needs to be plugged in as shown in Figure 2.3.

To implement custom transformations, a meta cartridge is provided, which can be adapted to the user's special needs and wishes. This way a custom code generator for any UML tool can be created. The UML 2.0 meta-model is supported completely. AndroMDA consists of various modules, which are pluggable and can easily be changed or customized. This gives great flexibility to the programmer to tailor the engine to his needs. Moreover, support for the most common UML tools like MagicDraw, Poseidon or Enterprise Architect is given. Using UML models to generate large portions of the source code makes them lose their static character and turns them into always up-to-date reflections of the system. Once again the engine's capabilities are not limited to UML but allow the programmer to generate code from any meta-model described in MOF XMI. As an additional feature AndroMDA validates these input models using OCL constraints which are related to the meta-model classes. Apart from offering constraints that protect the designer against the most common modelling mistakes, project-specific constraints can be added too.

Since AndroMDA is not only a template engine but a generator framework, it includes the possibility of model-to-model transformation. A model-to-model transformation foregoing a

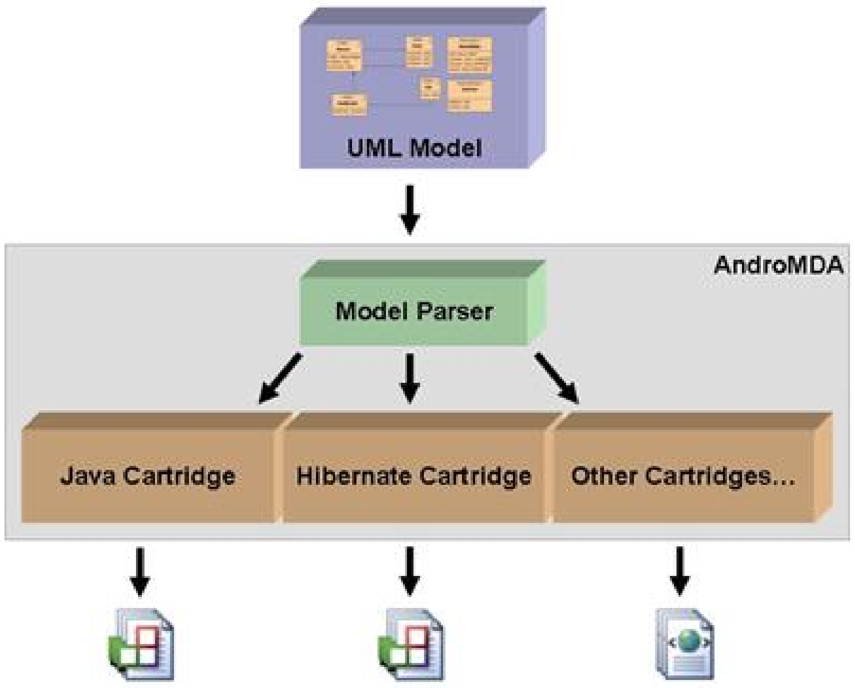


**Figure 2.3:** AndroMDA Architecture [Tea08]

model-to-code transformation is very useful to detail the model. The transformations needed for the first step can be written using Java or any other transformation language like the Atlas Transformation Language (ATL), for example. Templates are used to realize the model-to-code transformation. Relying on well-known template engines, currently Velocity and FreeMarker, any kind of textual output is producible. As mentioned above, arbitrary target architectures are supported via plug-in custom transformations named cartridges. Support can be found 24 hours a day by visiting the AndroMDA forum at [Tea08].

### 2.2.4 Java Emitter Templates

Java Emitter Templates (JET) and Java Merge (JMerge) are two powerful code generation tools included in the Eclipse Modeling Framework (EMF). JMerge allows the programmer to modify the generated code and preserve these changes if the generation process is repeated. This is done by tags that indicate which code can be replaced and which parts of the code should be left untouched.

JET is a generic template Engine which can be used to generate Java, SQL, XML or any other output from given templates. Furthermore, JET offers an easy way to generate source code from Ecore-based models. The syntax used by the JET-Engine is quite simple and similar to JSP, actually a JSP subset. A JET Template's structure always looks the same and does not contain more than four different kinds of elements.

The actual JET syntax is presented in detail, because this template engine has been used to implement the code generator presented in chapter 4.

#### 2.2.4.1 The JET Syntax

1. **JET Directive**
   Every JET Template starts with certain messages to the JET engine. These directives show the following syntax:

   ```
   <%@ directive { attr="value" }* %>
   ```

   Such directives are used to set the class name, the package name, to import Java libraries used in the scriptlets, to change the start and the end tag and to use a user-defined skeleton instead of the standard one for the Java file generation during the translation step. In other words, they affect the way the template is translated but do not produce any output.

   The JET directive must be included on the first line of the template. It defines a number of attributes and communicates these to the JET engine. Any unrecognized attributes result in fatal transition errors and any subsequent JET directives are ignored by the engine. The following attributes can be set to influence the way the template is translated:

   - **package:** This attribute defines to which package the Java implementation class is translated. If this attribute is left out the default package is chosen.
   - **class:** This field defines the class name of the implementation class. It is set to CLASS if not defined otherwise.
   - **imports:** Unlike in Java source code this list is space-separated and defines the packages or classes to import in the implementation class.
   - **startTag:** This string, used in a JET template, marks the beginning of a scriptlet, an expression or a directive. By default it is defined as "<%".
   - **endTag:** This tag signals the end of a scriptlet, an expression or a directive. It always needs to be combined with a startTag and is set to "%>" by default.
   - **skeleton:** If needed the skeleton of the implementation class to which the template is translated, can be changed by setting this attribute to the URI of the new skeleton. The URI is resolved in a similar way to the file attribute in the include directive. Its default value is "`public class CLASS\n{\n public String generate(Object argument)\n{\n return "";\n}\n}\n`".
     The name of the class in any skeleton definition must be CLASS because usually it is substituted by the template engine with the value defined by the class attribute.
   - **nlString:** This way the newline string to be used in the Java template class can be defined. It's default value is "`System.getProperties().getProperty("line.separator")`".

   A common header found similarly in each template would be:

   ```
   <%@ jet package="my.templates" class="LabelTemplate"
     imports="org.eclipse.emf.ecore.* java.util.*" %>
   ```

2. **Include Directive**
   The include directive offers the possibility of including another textfile, which is interpreted by the JET Engine during translation time and is included in the output file at

the location where the directive is situated in the template. Each file to be included needs its own include directive.

As the file is inserted in the template by the engine it may contain scriptlets, expressions or directives apart from simple text. This allows the programmer to avoid redundant programming as well as to encapsulate certain parts for better readability of the templates. As inducted above, the include directive has exactly one value and looks like the following example:

```
<%@ include file="copyright.jet" %>
```

The URI specified may be either absolute or relative. Relative URIs are always interpreted as relative to the folder of the template file that contains the include directive.

3. **Expressions**
This JET element evaluates the given Java expression at the invocation time of the template and appends its result to the `StringBuffer` object returned by the generate method. Its content must be a complete Java expression that returns a `String`. In case the engine fails to execute the expression correctly a fatal translation error occurs. Expressions can be used to insert strings directly as well. Their syntax is as follows:

```
<%= (new java.util.Date()).toLocaleString() %>
```

If the expression includes the character sequence <% or %> it needs to be escaped. This can be done by writing <\% and %\>. White spaces before the start and the end tag are optional.

4. **Scriptlets**
Scriptlets are short parts of logic implemented in Java. This enables the programmer to encapsulate certain functionality in the template and therefore makes them perfectly appropriate for modular programming. They are executed at the template invocation time, and it depends on the actual code whether they produce a string output or not. Scriptlets as well as expressions may have side effects as they modify the objects visible in them. To embed a scriptlet the following syntax has to be used in the *.txtjet file:

```
<% this Java code fulfils the required functionality %>
```

For detailed information on the JET Syntax and getting started with the engine refer to [Pop07a].

#### 2.2.4.2 The JET Model

Any template file can be the input of the JET engine as long as its suffix ends with "jet". The JET conventions are to use the extension of the file to be generated with this suffix. "javajet" implies the generation of Java source code, "xmljet" and "sqljet" are other frequently found template file extensions.

The transition from a *.javajet file to a java file consists of two steps. In a first step the *.javajet file is translated to a Java implementation class. This implementation class has a method that can be called to obtain the result of the generation process as a String. If the skeleton of the implementation class has not been changed by the programmer this method is called *generate()*. The following two objects are contained implicitly by the Java implementation class and can be referenced from the template:

- **stringBuffer:** This ``java.lang.StringBuffer'' object is used to create the resulting string as soon as the *generate()* method is called.
- **argument:** this ``java.lang.Object'' can be an array of objects that contains all the arguments passed to the *generate()* method.

Any further skeleton attributes need to be specified in the jet directive.

The separation of the generation process in two steps allows JET not only to offer a reduced syntax set but the complete Java functionality. The first step is automatically completed by the JETNature and JETBuilder modules, included in the JET package. As these classes operate on Eclipse workspace projects only, they have to be called manually for plug-in transition. For more information on this topic see [Pop07b].

To receive the output source code in the end, the second step of the transition needs to be completed by calling the implementation class' *generate()* method and handing over the appropriate arguments. Figure 2.4 illustrates the two-step nature of the process. The transition is composed of a Translation followed by the actual Generation. Both steps are combined by the JETEmitter class which is part of Eclipse Modeling Framework and situated alongside the other classes needed in the org.eclipse.emf.codegen package.

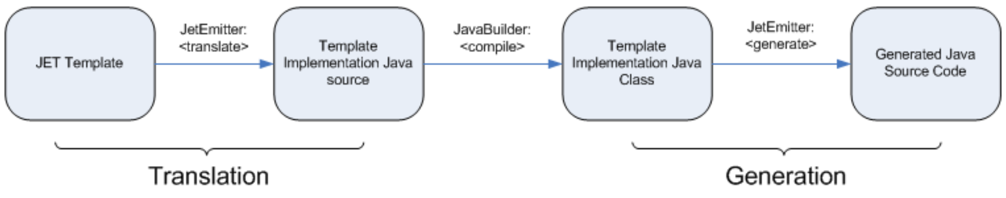


**Figure 2.4:** JET Transition

## 2.3 The Eclipse Modeling Framework

As a Java-based framework, the Eclipse Modeling Framework supports automatic code generation based on structured models as well as the construction of tools and other applications based on such models. Being part of the Eclipse project, which is developed as an open source initiative, it offers a low cost entry to converting object-oriented models to efficient, syntactically correct and easily customizable Java code.

Today object-oriented modelling frequently implies the use of diagrams specified in the Unified Modelling Language (UML). Using a combination of these standard notations allows the complete specification of an application. Apart from easing the understanding of a system, these models become part of the development process as input to code generation tools like JET.

Regarding the modelling aspect, the low cost entry argument can be applied once more. This is due to the fact that an EMF model requires just a small subset of things that are supported by UML. To be precise, this subset is called Essential Meta Object Facility (EMOF) and offers the core features of MOF, with an infrastructure similar to UML. As only simple definitions of classes, their attributes and relations are used a full scale graphical modelling tool is not needed. The EMF's meta-model is called Ecore. Any Ecore model is defined and stored as an XML Metadata Interchange (XMI) description. Several possibilities are given to create such a model:

- An XMI model can be written manually using any XML or text editor.
- Most graphical modelling tools offer a way to export an XMI model from a graphically created model.
- If Java interfaces are annotated with model properties, the XMI description can be generated automatically.
- Last but not least, XML Schema can be used to describe the model.

With little effort the first approach, though the most direct one, appeals only to programmers with very sophisticated XML skills. Having great tool support in combination with the fact that diagrams are most frequently easier to understand than code, the second approach is the most popular one. Java annotations are most frequently used by programmers to get the benefits of EMF and its code generator without the use of a graphical editor. The description of the model using XML Schema is very appealing if the application to be created needs to read or write a particular XML file format.

The EMF generator subsequently generates the corresponding set of Java classes out of any of these descriptions. Manual changes like the addition of classes or variables are preserved even if the code is generated anew. Other features coming along with EMF are a framework for model validation, model change notification, persistence support including XMI and schema-based XML serialization and a very efficient reflective API for the manipulation of generic EMF objects. Another important aspect is that EMF provides a foundation for interoperability with other EMF-based tools or applications.

Basically EMF consists of three fundamental frameworks: the EMF core framework, EMF.Edit and EMF.Codegen. Most of the above mentioned features are contained by the core framework. Moreover, the Ecore or meta-model used for the description of models and their runtime support are part of this module. EMF.Edit extends the core framework's functionality by including reusable adapter classes that offer the possibility of viewing and command-based editing of a model as well as a basic model editor. The EMF.Codegen represents EMF's code generation facility and distinguishes three levels of code generation. Firstly, the Java interfaces and implementation classes specified by the model as well as a factory and a package (meta-data) are created. Consequently implementation classes, so called ItemProviders, are added that adapt the model classes for viewing and editing. On the third level, the source code needed for the EMF model editor is generated. This editor includes a GUI, which allows the specification of options as well as the invocation of the generation process. Downloads, Tutorials and other information regarding EMF can be found at [Fou08c].

Chapter 3

# OntoUCP – Ontology-based Unified Communication Platform

Developing complex software today would be much more effort if there were not dozens of programmes supporting the creator. These programmes contain features ranging from simple syntax highlighting to model-to-code transformation tools. To give an example, the Eclipse software project, which is one of the most powerful software development frameworks, is worth to be mentioned. Relying on this already developed and tested software, new programmes can turned into functioning prototypes very quickly.

As automatically generated code is hardly ever optimized this step, if needed, still has to be completed manually later on. Concerning user interface programming, an automated source code generation is very interesting, because user interfaces normally consist of a small number of recurring widgets, that are combined in different ways to fulfil a certain task. If supported by automated source code generation the programmer does not only save a lot of work and can reduce the probability of syntax errors, but is able to create a prototype fastly. This is due to the fact that the amount of code needed for the generation is most of the time much smaller than the amount of code that is generated.

One further advantage is that the amount of time needed till a functioning prototype is available is reduced drastically, not to forget that changes in the interface conception just lead to a new generation instead of tedious changes in the source code itself. Such code generation is most frequently based on an abstract user interface description. The OntoUCP project[1] takes this development one step further and aims to generate WIMP UIs out of an interaction description. This interaction description is provided using a specially developed Communication Definition Language (CDL). From this description an abstract user interface description is generated automatically, which is then turned into source code in a second step.

This project aims to enable people whose programming skills are limited to create systems with WIMP UIs. The description of such a system is per se independent of the target device and of the graphical toolkit used.

One more thing to point out is the fact that such a system does not even care whether it communicates with a person or another system, which makes it applicable for Human-Machine as well as for Machine-Machine interaction. This results in great flexibility and

[1]http://www.ontoucp.org

widens the range of applications which can be realized using the Communication Platform. Figure 3.1 illustrates which parts are needed to create a working system, using this newly

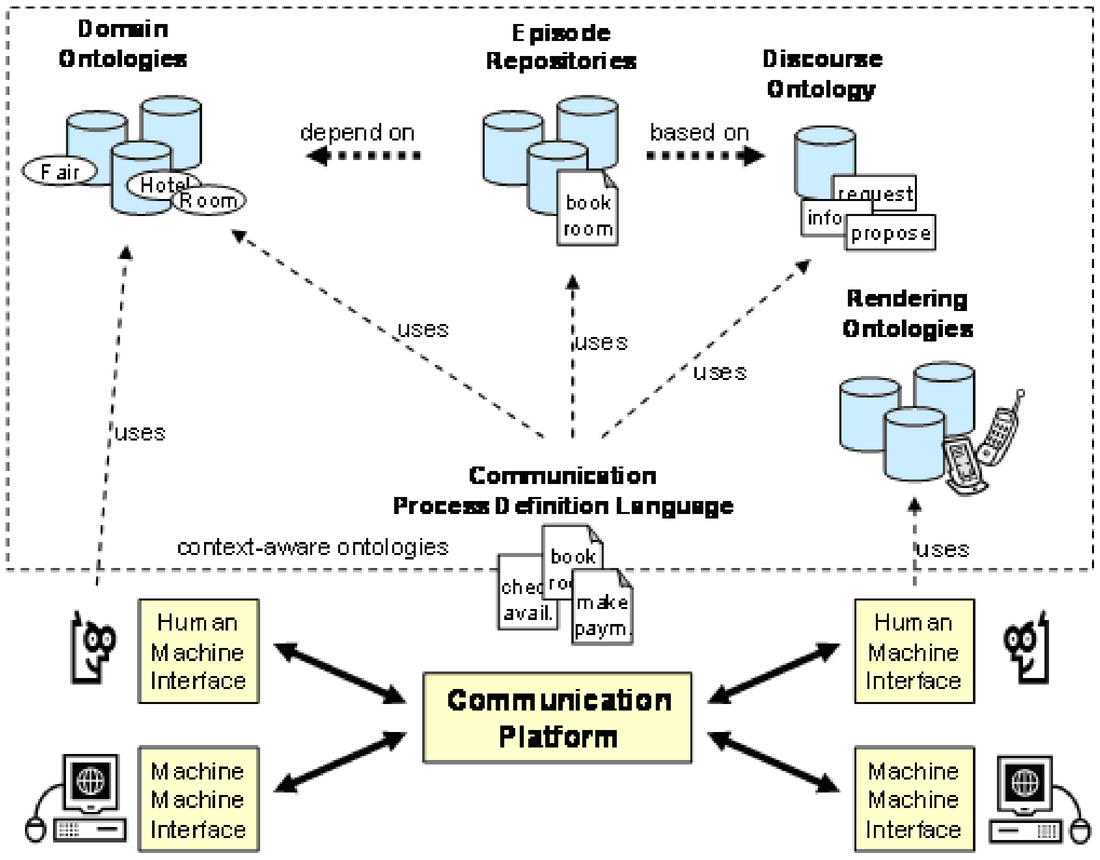


**Figure 3.1:** The Communication Platform [KFAP06]

developed technology.

The central part is the above mentioned Communication Definition Language. This language is needed to model the communication between the user and the system. It is based on speech act theory, which makes its use much more intuitive than common programming languages. The CDL relates communicative acts with knowledge about domain and workflow. This represents a combination of the content and intention of the communication. The intention is represented by the type of communicative act (e.g. Question or Informing) whose content is modelled by objects defined in the domain specification.

The discourse model repository holds sequences of communication, needed to achieve some goal (e.g. retrieve or gather information). Typical combinations of related communicative acts are called adjacency pairs (e.g. Question-Answer, Offer-Accept or Request-Inform). The communication, resulting through the exchange of communicative acts between two parties, can be described as a number of resulting interaction scenarios. The synchronisation between the two communication parties is related to the exchanged communicative acts. As the number of interaction scenarios is limited it can be seen as a finite state-machine. A transition is triggered according to the sending and receiving of communicative acts.

## 3.1 The Discourse Model

One of the major advantages of a system like OntoUCP is that the system designer does not have to care about how things are displayed but can entirely focus on the interaction with the system. Using a specially developed description language, which is based on results of human speech theory, he describes the communication between the system and its user on a high level of abstraction. This user can either be human or another electronic system. As this description language is based on language theory results, its use is far more intuitive than common programming languages. The theoretical background needed to understand the elements used to create a discourse model is presented throughout the following paragraphs.

### 3.1.1 Communicative Acts

One of the key ideas in the OntoUCP project is to use speech acts [Sea69] to model human machine communication on a high level of abstraction. Speech, or more generally communicative acts are based on the fact that speaking or communicating is not only used to describe something or to give a statement, but also to do something with intention – to act. So-called performatives, expressions to perform an action are illustrating examples. These are sentences like "I nominate John to be President" or "I promise to pay you back", whose act is performed by the sentence itself. The speech is equal to the act it triggers.

According to Searle, not words or sentences but these speech acts form the basic units of communication. This means that speaking a language is performing speech acts like making statements, asking questions, giving commands, making promises, etc. Any of these speech acts can be represented in the form $F(P)$ where $F$ is the illocutionary force and $P$ its propositional content. Three speech acts with the same proposition $P$ would be:

Sam smokes habitually.

Does Sam smoke habitually?

Sam smokes habitually!

As said before the proposition $P$ (Sam smokes habitually) stays the same, what differs is the illocutionary force $F$. The first sentence is an assertion whereas the second one is a question and the third one poses a command. Additional ways to characterise speech acts are:

- **Degree of strength**, describing the intensity and importance of the speech act. Speech acts like a command and a polite request may have more or less the same illocutionary force but different strength.
- **Propositional content**, describing the content of the speech act.
- **Preparatory condition**, ensuring that the receiver is able to perform the action required by the speech act.

Searle's work offers a clean and formal way to computer scientists to describe communication even apart from speech or natural language. The name communicative act was introduced to lay emphasis on their general applicability. Such communicative acts simply abstract from the corresponding speech act as they do not rely on natural language as a basis. So far they have been used successfully in several applications like Knowledge Query and Manipulation Language [FFMM94] or electronic commerce and information systems [KM97]. These systems use communicative acts on a low semantic level whereas OntoUCP embeds them in a Communication Definition Language which describes communication on a high level of abstraction. Figure 3.2 shows a selection of the most frequently used communicative

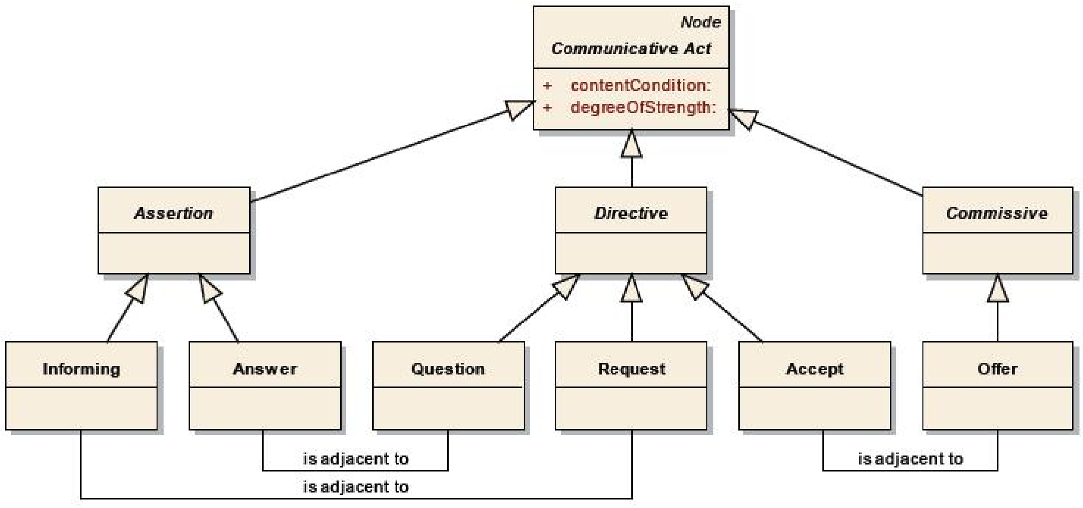


**Figure 3.2:** Communicative Act Taxonomy [BFK+08]

acts in OntoUCP. They are used to model the intention of the communication, referring to elements of the domain of discourse. As stated before, these elements are defined in the domain-of-discourse model.

### 3.1.2 Conversation Analysis

To describe the relationship between communicative acts, a further theoretical device is needed. Conversation Analysis offers theoretical background to describe typical turns in human conversation. Using the CDL, any communication is represented through a composition of communicative acts. Consequently patterns found in human communication can be mapped to typical combinations of communicative acts.

As can be seen in Figure 3.2, two communicative acts related via an is adjacent to relation are called adjacency pair. Such corresponding communicative acts describe a typical dialogue fragment between the user and the system (e.g. Question-Answer or Offer-Accept). Patterns such as "noticing" or "teasing" have not been considered in this model as they are still too subtle for the current state of human-machine interaction [ABF+06b].

### 3.1.3 Rhetorical Structure Theory (RST) Relations

As shown above, communicative acts are quite handy as basic basic blocks of communication. Typical patterns can be described using adjacency pairs, but what is still missing, are means to describe the connection of communicative acts in a more general way.

Rhetorical Structure Theory offers the possibility to describe internal relationships among text portions and associated constraints and effects. It represents the functionality of an underlying text in the form of a tree structure. These basic blocks, or in our case communicative acts, are associated with non-leaf nodes, which are the rhetorical relations. Their purpose is to describe the function of the associated sentences, or communicative acts. RST started out as an empirical approach with three simplifying assumptions that have to be considered:

1. The elementary units joined by rhetorical relations are clauses. These units are called elementary text spans and can be simple sentences, main or subordinate clauses. This constraint has no scientific background but has been introduced by the authors [MT88] to make analysis manageable and to strengthen agreement among human speech analysts.
2. Two elementary text spans or sub-trees are connected by exactly one rhetorical relation. In case of more connections only the most prominent one has to be considered.
3. A tree constructed using rhetorical relations does always result in a coherent text. This is due to the assumption that rhetorical relations do not imply any implicit text spans.

The above given assumptions ease the use of RST in general but they do not hold for every application. In our case some refinements need to be done to make RST applicable as part of the CDL.

The first assumption can easily be challenged by the fact that the same thing can be expressed using different formulations. An example would be a causal relation which can be expressed using a “because”, which connects two clauses, or by a “because of” which attaches a noun phrase to a clause. These variations are considered as semantically equivalent, hence their rhetorical representations should be the same. To make a system handle both text variants equally, one of two measures should be taken:

1. Considering textual analysis such variants must be mapped to a generalized representation or, regarding the generation, either variant must be producible.
2. Another possibility would be the generalisation of the assumption, so that a rhetorical relation might also hold between two text spans within the same clause.

As variations in natural language expression are quite common, many systems follow one of the above given strategies to resolve this problem. This problem appears in a slightly different form as far as human-computer interaction is concerned. In this case rhetorical relations that might be obvious for a human observer might be inaccessible for the system. Consequently, special attention needs to be paid to decisions about information for the interface. A different partitioning of the information leads almost certainly to a different user interface.

The second assumption can be challenged by taking a closer look at the virtue of perspectives of rhetorical relations. Following systemic functional linguistics [BDRS97], three categories of rhetorical relations can be defined:

1. **interpersonal relations**
   They define the intentional structure of the discourse (e.g. the motivation relation).

2. **ideational relations**
   Through them the informational structure of the discourse is defined (e.g. sequence or cause relation).

3. **textual relations**
   They refer to the textual presentation itself (e.g. announcement of enumeration or explicit reference to a piece of text).

Since there are cases in which the first two categories appear competitive they are considered to be the more important ones. The current view among scientists is that both structures are necessary but there is still no agreement whether a system must build on both of them explicitly. For detailed information on how the relations have been classified within the OntoUCP project see [ABF+06b].

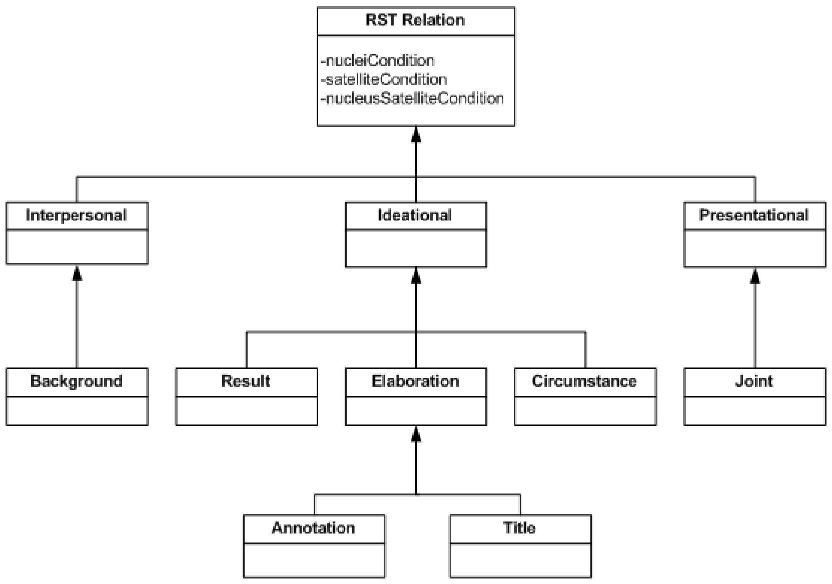


**Figure 3.3:** Taxonomy of used RST relations

The third assumption has been challenged rarely so far. This is due to the fact that computational approaches deal most frequently with narratives and fact-reports, whereas the problem most frequently arises in inference-rich discourses. This problem might occur for texts which are composed of several parts which extend the text's structure in a similar way. A frequently found example are mathematical texts in which a chain of equations may be preceded or followed by explanation or justification for the different steps without an explicit reference to the precise position of the text. This connection then can only be discovered in the presentation and, therefore by the addressee.

A reasonable solution to this conflict would be to create and maintain a precise rhetorical structure and to leave the appropriate presentation to the rendering engine. This approach is supported by a rich repertoire of expressive means on most devices. A composed construct could be represented as list or table for example, where extensions are included as clickable links.

What can be learned for human-computer interaction is that a certain level of explicitness needs to be chosen as a design decision and to be maintained throughout the project. This will make the result more predictable but will not be critical for most applications.

The following RST relations are suitable for human-computer as well as for machine-machine interaction. In total, there are five basic relations: Joint, Background, Elaboration, Circumstance and Result. These relations are divided into the three categories introduced above as is illustrated in Figure 3.3. Their difference lies in the constraints they place on each participant. The seven above named relations consist of symmetric (multi-nuclear) as well as of asymmetric (nucleus-satellite) RST relations. Multi-nuclear relations link RST structures or communicative acts that are independent of each other and offer the same kind of support to their parent relation.

Nucleus-Satellite relations link different types of RST structures or communicative acts. The main intention is linked via the nucleus, whereas the satellite links a structure that supports the nucleus. All specified nucleus-satellite relations are interpersonal or ideational relations. This signifies that they represent relationships of the content, which is referred to by the associated communicative acts. Sequence and Joint are presentational relations. This means that they do not correlate the content of the associated communicative acts but communicative acts themselves based on the communicative goal of the discourse.

Each RST relation has at least two children, the nucleus and the satellite. The children can be either RST relations or communicative acts. As every RST relation is followed by at least two children, the leaves of the end of the discourse tree have to be communicative acts. An additional constraint has been introduced which limits the number of following objects to exactly two for nucleus-satellite relations.

### Joint

Through the Joint relation associated objects are only related through their presentation. As stated above, there does not have to be a connection between the content of the communicative acts at the leaves.

### Elaboration

The elaboration relation provides details on the information presented by the nucleus. These details can be properties, parts, actions or other elaborations on the nucleus. The constraint posed on the satellite sub-tree is that it must contain only assertive communicative acts. The elaboration relation requires exactly one nucleus and at least one satellite.

### Annotation

The annotation relation is a specialisation of the elaboration where the detailed information is marked as additional comments. This relation mainly serves as a hint for rendering the elaboration.

#### Title

The title relation is a specialisation of the elaboration as well. In this case the detailed information is marked as a restatement of the nucleus information. This relation mainly serves as a hint for rendering the elaboration.

#### Circumstance

This relation describes situations in which the nucleus is embedded. The nucleus gets valid as soon as the satellite situation is fulfilled (e.g. a visitor of an art exhibition only gets information about the pieces of art that are in the room he is in). This relation requires exactly one nucleus and one satellite.

#### Result

The satellite sub-tree of the result relation describes the result caused by the nucleus. It is possible to have more than one result. As the result consists of information conveyed to the receiver, the communicative acts contained by the satellite can only be assertive. There needs to be exactly one nucleus and one satellite.

#### Background

The so far only interpersonal relation is used to provide the system designer with the possibility to include information that he considers as necessary for understanding or further acting. This relation is based on the designers believes. An example would be that the designer believes that the user has to get the content delivered by the satellite to be able to understand or act upon the content represented by the nucleus. Due to this reason the Background relation requires exactly one nucleus and one satellite. This satellite provides details on the information conveyed by the nucleus. Such details can be properties, parts, actions or other elaborations of the nucleus. According to this fact the satellite sub-tree has been constraint to contain only assertive communicative acts.

### 3.1.4 Procedural Constructs

Most RST relations describe a subject-matter relationship between the branches they relate (e.g. the dialogue situated in the nucleus branch of an Elaboration is elaborated by its satellite branch). These relations only suggest a particular execution order but do not imply one. In some cases it turned out to be useful to define sequences and repetitions, based on the evaluation of some condition. This led to the introduction of prcedural constructs. By determining the order of when to display information to the user, they determine which information can not be presented together in the same screen of the WIMP UI as well.

#### Sequence

This procedural construct defines an order for the execution of its sub-trees. Like any discourse relation, it relates two sub-trees and implies the constraint that the type of the highest level communicative act has to be the same in both.

### IfUntil

The `IfUntil` construct allows the interaction designer to define the order of execution related to conditions. Its statechart is shown in Figure 3.4. The combination of an if-statement and a conditional loop make this construct more complex compared to procedural statements found in typical procedural programming languages.

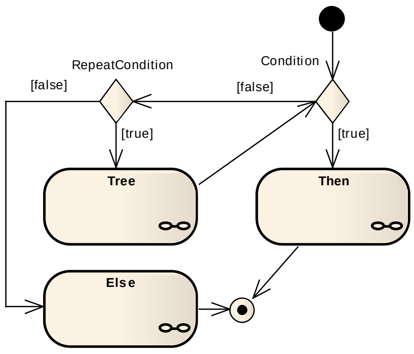


**Figure 3.4:** `IfUntil` Statechart

As soon as the `If Condition` is fulfilled, the `Then` Branch is executed. In case it turns out false two different scenarious are possible:

1. the `Tree` branch is executed again and again until the `Condition` turns true, or
2. the optional `Else` branch is executed if the `RepeatCondition` is false. As the `Else` branch is optional it might be missing as well. In this case the sub-tree with the `IfUntil` as root node, is considered as completely executed. The dialogue continues either at the next higher level or stops if the `IfUntil` node is the root of the dialogue itself.

Combining communicative acts, adjacency pairs, RST relations and procedural constructs, the CDL was developed. Its purpose is to describe the interaction between a user and a system. Every such discourse is represented by a tree structure. The leaf nodes are the Communicative Acts, which are connected via RST-relations and procedural constructs. Graphically these relations are represented through inner nodes. Some types of communicative acts have been related as adjacency pairs using the results of conversation analysis. This structure is illustrated by the UML-class diagram in Figure 3.5. What needs to be considered is that this class diagram allows modelling other graphs besides tree structures as well and therefore is not as restrictive as its interpretation used to create a discourse model. For further details on the content specification language see [PFA+09] and [ABF+06a].

## 3.2 The Abstract User Interface

The abstract UI model is an intermediate step between the discourse model and the generated code. Basically, it is a widget-based tree which represents the user interface structure. It is based on an abstract WIDGET class, from which all the other WIDGETS are derived. Basically, all WIDGETS can be divided into two classes. The OUTPUTWIDGETS' only task is to represent

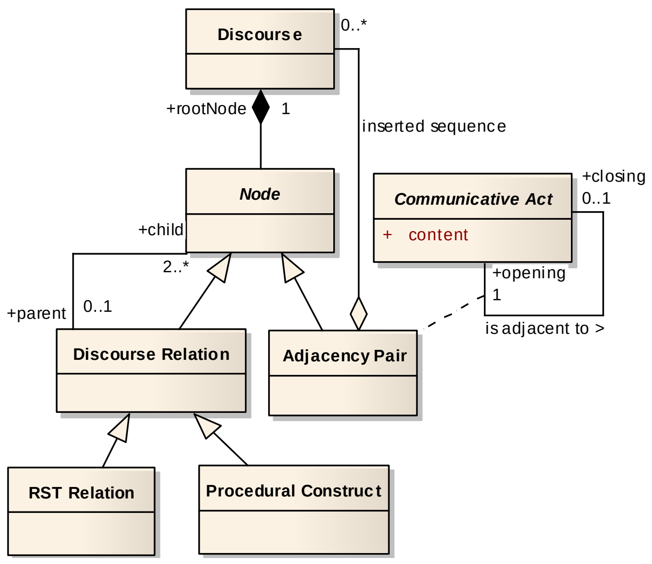


**Figure 3.5:** Discourse Metamodel [PFA+09]

information, whereas the InputWidgets are used to gather information from the user. A short description of the most important Widgets found in the abstract UI meta-model is given below. For more details see [ABF+07b]. To separate cleary between Widgets defined in the abstract UI meta-model and Java widgets the abstract ones are written with small capitals.

### Widget Class

Widget is the abstract representation of a widget capturing its basic characteristics. All Input as well as Output Widgets are derived from this class. Its properties are:

- ***visible*** – the value of this attribute defines whether the widget is visible or not.
- ***style*** – each widget can be assigned a certain style, which effects the way it is displayed.
- ***name*** – the name is used as a unique reference to the widget.
- ***enabled*** – this property enables or disables the widget in the actual screen representation.
- ***tracesTo*** – this field relates the widget to a communicative act. This communicative act delivers the information needed to be displayed by the widget.
- ***layoutData*** – contains the data needed to set constraints imposed by a certain layout type (e.g. GridBagConstraints).
- ***parent*** – in case the widget is contained by another widget this field specifies the container widget.
- ***content*** – this property field holds the data or a reference to the data the widget should display.

- ***contentSpecification*** – this property is relevant only for the mapping of the discourse model to the abstract UI description. It specifies which data to put in the widget's content field by means of the Object Constraint Language(OCL).
- ***text*** – this field contains static text to be displayed by the widget.
- ***getTopPanel()*** – this method returns the top panel containing the widget. It is needed to relate the widget to a window.

### OutputWidget Class

As most of the widgets defined within the abstract UI meta-model can be distinguished due to their purpose, two abstract classes have been introduced, which inherit all their characteristics from the widget class presented above. All widgets displaying information are derived from this class.

### InputWidget Class

It is an abstract class as well and represents the counterpart to OutputWidget. Furthermore it serves as base class for all widgets needed to collect information from the user. It extends the widget class by introducing the property *event*.

- ***Event*** – this property holds the specification of what needs to be done with the collected data.

### Panel Class

This class serves as a container for other widgets, which are arranged according to a given layout manager. It is derived from Widget and therefore inherits all its characteristics. Panel is not an abstract class and can therefore be found in abstract UI models. Moreover, it serves as base class for other classes that contain widgets (e.g. ListWidget).

- ***Widgets*** – this property hosts the widgets contained by the panel.
- ***Layout*** – specifies the layout manager to be used to arrange the widgets.

### Frame Class

Each window of the application to be generated is represented by this widget class. An application may contain more than one window, which implies that more than one Frame widget will be found in the abstract UI model. As a Frame represents a window and a window can not be contained by another widget but choice, it can only be found one level beneath the root choice in the abstract UI description.

- ***title*** – the title of the window is set according to the value found in this field.
- ***screenResolutionX*** – this property defines the $x$ resolution of the window.
- ***screenResolutionY*** – this property defines the $y$ resolution of the window.

#### ListWidget Class

This abstract widget class is derived from PANEL as well and represents a reoccuring set of widgets.

#### Choice Class

This class offers the opportunity to choose exactly one out of its child widgets. A CHOICE is the root of all abstract UI representations, its children being the windows to be generated (i.e. FRAMES). It may, of course also appear somewhere within the tree structure of the abstract UI. CHOICE is derived from PANEL. As PANEL is derived from WIDGET, this class inherits a WIDGET'S characteristics as well.

#### TabControl Class

Being a specialisation from CHOICE, TABCONTROL offers the user to switch between different PANELS within the same screen. Concerning the inherited characteristics it is equal to CHOICE, differing only in the code that needs to be generated.

#### Label Class

This class is the abstract representation of a label widget. As it can only be used to display information, it is derived from the abstract widget class OUTPUTWIDGET.

#### Button Class

It defines the abstract representation of a button widget and is derived from INPUTWIDGET.

#### TextBox Class

This class represents a user input field and is derived from INPUTWIDGET as well.

#### Hyperlink Class

The HYPERLINK is the abstract representation of a hyperlink and inherits all its characteristics from BUTTON.

#### PictureBox Class

This class represents the abstract widget for the illustration of a picture.

- ***picture*** – this attribute contains the name of the picture to be displayed.

### ImageMap Class

This class belongs to the category InputWidget and is, therefore derived from this class.

- ***picture*** – this property field carries the name of the picture to be used.

### ComboBox Class

A Combo Box widget represents the abstract class of a widget that allows the user to choose one or more items from an offered list. It extends the class InputWidget introducing the following two properties:

- ***type*** – the value of this property defines whether only one or more items from the offered list can be selected.
- ***list*** – this property contains the items to be added to the list offered to the user.

### Style Class

A Style can be added to any widget to set its style properties. It is not derived from any other widget.

- ***id*** – this attribute contains the name of any CSS class which should be used to set the associated widget's style attributes.

### LayoutManager Class

Each Panel can optionally have a LayoutManager. This abstract class forms the base class from which all layout managers are derived.

### FlowLayout Class

This class is the abstract representation of a flow layout manager. It is derived from LayoutManager.

### XYLayout Class

This class is the abstract representation of an absolute, or *XY* layout. Just like FlowLayout it is directly derived from LayoutManager.

### GridLayout Class

This class is directly derived from LayoutManager, but as it is an abstract representation of a grid layout manager, it introduces two more attributes.

- ***rows*** – this attribute contains the grid's number of rows.
- ***cols*** – this attribute contains the grid's number of columns.

#### LayoutData Class

Depending on the layout manager additional information may be needed. This information is attached to the corresponding widget and is modelled as LayoutData in the abstract UI meta-model. LayoutData has been introduced as an abstract class from which the layout manager specific layout data is derived.

#### XYLayoutData Class

This widget provides the additional data needed to place a widget correctly in an $XY$ layout. It is directly derived from LayoutData.

- ***x*** – this values specifies the $x$ coordinate of the upper left corner of the widget.
- ***y*** – this values specifies the $y$ coordinate of the upper left corner of the widget.
- ***width*** – the width of the widget is specified by this attribute.
- ***height*** – the height of the widget is specified by this attribute.

#### GridLayoutData Class

To place a widget correctly on a grid, additional data is needed. This data is stored in an GridLayoutData object, which is attached to the widget that should be placed. GridLayoutData is directly derived from LayoutData.

- ***row*** – this attribute specifies in which row of the grid the widget should be placed.
- ***col*** – the corresponding column is specified by this attribute.
- ***rowSpan*** – any widget spans one row by default. If a widget should cover more rows, the value of this field needs to be set.
- ***colSpan*** – this attribute specifies the number of columns a widget spans.
- ***alignment*** – several predefined values are available to align a widget. Therefore, this attribute contains a value specified in the abstract UI enumeration. AlignmentType.
- ***weightx*** – grid layout managers offer to influence the distribution of extra space. This attribute sets the $x$ value of the corresponding widget.
- ***weighty*** – this attribute influences the distribution of extra space on the $y$-axis.
- ***fill*** – predefined values for this attribute are provided in the abstract UI's FillType enumeration. This attribute contains one of those values.

The abstract UI model is generated by applying mapping rules to the discourse model. Both models are instances of meta-models, which have been constructed using Eclipse's MOF-like core meta-model called Ecore. The model-to-model transformation is supported by languages like the ATLAS Transformation Language (ATL). For more details on the mapping rules see [ABF$^+$07b].

## 3.3 The Rendering Architecture

The rendering process is divided into two steps. At first an abstract user interface description is generated, using the discourse model and additional information, like a device profile and user preferences as input. This model specifies the structure of the user interface on a widget basis. It is independent of the output toolkit, used to display the screens in the end, but it is not independent of the target device. All the additional information like screen resolution, paths, etc. is included either directly in the abstract UI description or passed on to the code generator via a properties object.

The separation of the rendering process in two steps has the major advantage that the generated abstract model is still platform independent and can be translated into several different GUI-toolkit languages in a second step. Another advantage is that the design of the screens that are finally displayed can be influenced directly and much more easily by modifying the generated abstract UI model, before triggering the actual code generation. This leads to a more predictable output.

Figure 3.6 shows a conceptual view of the rendering architecture, highlighting the components needed to generate a system automatically. As illustrated below the rendering architecture consists of three main components:

- A Discourse To Structural UI Transformer
- A Concrete UI Code Generator
- And a UI Behaviour Generator.

The Discourse To Structural UI Transformer takes the Discourse- as well as the Domain-of-Discourse model as input and transforms them into a structural UI model. This translation is completed, taking device constraints into consideration. Therefore, the resulting abstract UI model is not independent of the target device but it is still not tailored to a certain target toolkit. The Discourse To Structural UI Transformer completes a model-to-model transformation based on rules. Such a mapping rule can state, for example that each "Informing" communicative act found in the discourse model has to be transformed to a panel containing a label widget for each discourse object referred to by the communicative act.

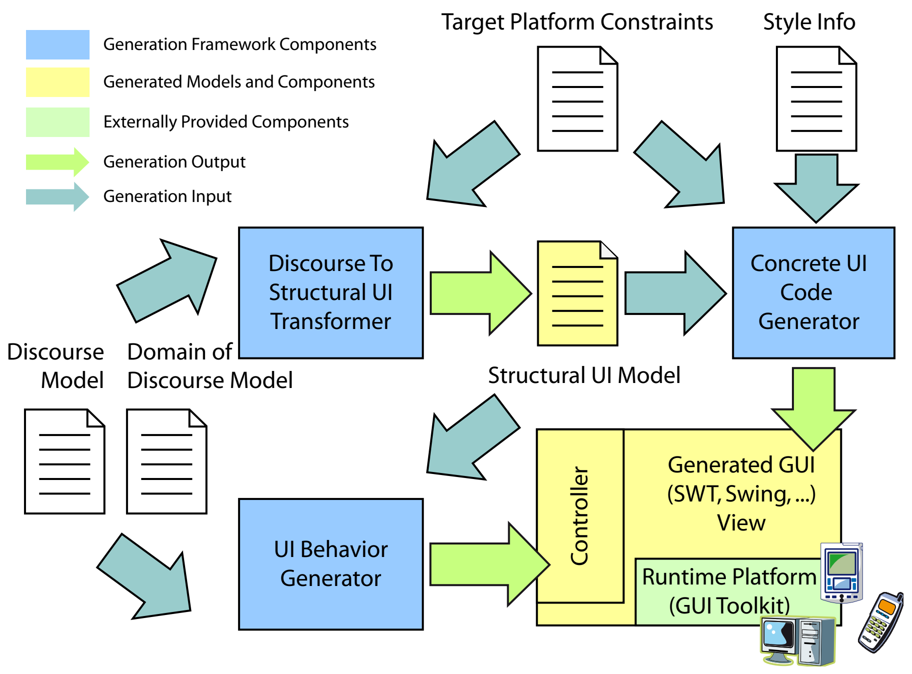

Generation Framework Components
Generated Models and Components
Externally Provided Components
Generation Output
Generation Input
Target Platform Constraints
Style Info
Discourse To Structural UI Transformer
Concrete UI Code Generator
Discourse Model
Domain of Discourse Model
Structural UI Model
Controller
Generated GUI (SWT, Swing, ...) View
UI Behavior Generator
Runtime Platform (GUI Toolkit)

**Figure 3.6:** The Rendering Architecture [FKP+09]

The concrete UI code generator takes the abstract UI description, generated by the Discourse To Structural UI Transformer as input and maps it to concrete widgets, available on the target platform's GUI toolkit. To gather the information not provided by the abstract UI description, additional constraints are passed on and style information can be included deliberately.

Apart from the source code for the actual window representation, the component needed for the communication during the runtime of the system is generated as well. These components are generated according to the Model-View-Controller (MVC) pattern. The user interface needs to be able to receive data and update the corresponding widgets. This is done by a controller that selects the corresponding screens (views) and sends the data needed to fill the widgets via communicative acts. Apart from receiving information from the controller, the user interface needs to be able to accept input from the user via its widgets.

The UI Behaviour Generator generates a finit statemachine that represents the model component of the MVC pattern. As multimodal interfaces are supported the output depends on the type of interface. For further details on this component see [PFA+09], [AFK+06] and [ABF+07a].

The input models needed for the rendering process have to be provided by the system designer. The Discourse Model specifies the communication between the system and the user and refers to objects defined in the Domain of Discourse model. This model contains the objects that are needed to model the system's domain of application. Optional style information can be provided in the form of Cascading Style Sheets (CSS), if the designer wishes to influence the design of the generated WIMP UI.

As soon as the designer selects a target device for the application, additional constraints are set. These constrains mainly represent characteristics of the selected device. They contain information like screen size, screen resolution or package names. Moreover, information like country, language or position should be contained as well as a personalization context, which determines whether the end user is a novice or an expert. This information is partly needed for the abstract UI generation as well and, therefore passed on to the code generator. Additional information needed by the code generator is style information. Within this document font styles, types, sizes and any other information concerning the layout can be defined. All these values have to be set before the start of the generation process.

## 3.4 The Online Shop

To illustrate the ideas presented so far, an online shop application is introduced as a running example. It is presented from the very first step till the running application. The communication taking place between the system and the user has been specified using the CDL presented above. To support the designer, a graphical modelling tool is available, which has been used to create this discourse model. The graphical tree representation of the whole discourse can be found in the appendix. Each step of the discourse is presented in detail, following the order of appearance in the application.

### 3.4.1 The Online Shop Discourse Model

As soon as the onlineshop application is started, it should offer the user a list of all the product categories available. Using the CDL this can be modelled as an Offer-Accept adjacency pair. As can be seen in Figure 3.7, the system designer chose to provide the user with some extra information on the product categories. Therefore, the Background relation has been used. Its satellite branch extends the nucleus and provides the communicative act that holds the extra information. All leaves contained in this part of the discourse are communicative acts. This indicates that this sub-tree can be found at the lower end of the discourse tree.

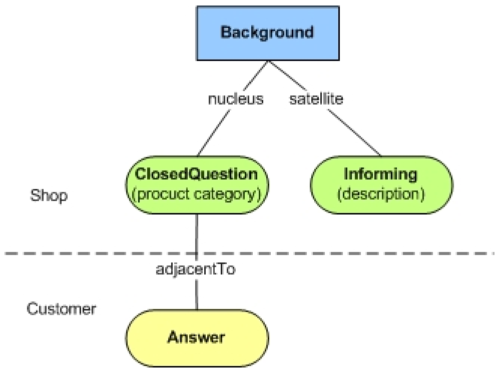


**Figure 3.7:** Background relation

The sub-tree illustrated in Figure 3.7 is related to the one in Figure 3.8 through an IfUntil relation. The second one forming the Then branch is presented as soon as the Tree branch has been executed successfully. The main purpose of this part is to get a product selected by the user. Therefore, the nucleus of this relation is another adjacency pair which offers a list of products available in the formerly selected product category.

The satellite branch links a JOINT relation which contains two informings. These communicative acts offer the additional information required by the BACKGROUND relation to the user. The JOINT only becomes necessary because there is more than one informing to be linked.

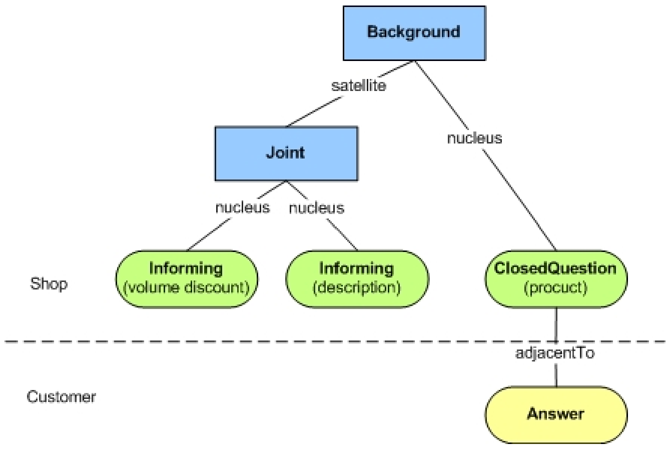


**Figure 3.8:** Then branch of IfUntil relation

As soon as the user selects a product, this branch of the IfUntil relation has been completed successfully and the next relation to be found one level further up the discourse tree is being executed. In this discourse model, this is once more an IFUNTIL relation. With the extract presented in Figure 3.8 the TREE branch of this relation has been completed and the focus lies now on the THEN branch. This branch consists of a JOINT relation that links three adjacency pairs as can be seen in Figure 3.9.

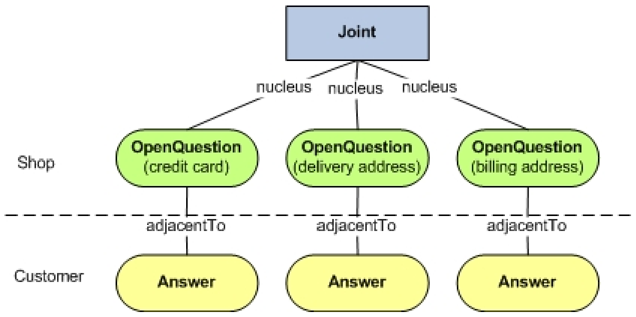

Joint
nucleus
nucleus
nucleus
Shop
OpenQuestion (credit card)
OpenQuestion (delivery address)
OpenQuestion (billing address)
adjacentTo
adjacentTo
adjacentTo
Customer
Answer
Answer
Answer

**Figure 3.9:** Joint relation

In contrast to the adjacency pairs presented above, these ones consist of OpenQuestions. Their target is to collect the data concerning the billing and the delivery address as well as the credit card details needed to finalize the transaction. This signifies that the user adds information to the system. A ClosedQuestion as presented in Figure 3.7 and 3.8 allows the user only to choose one from several available options. Whether the data inserted by the user in an OpenQuestion is valid or not needs to be verified by the application logic of the system. If the data entered by the user is to be accepted by the system, the online shop discourse has come to a successful end.

### 3.4.2 The Online Shop Domain of Discourse Model

Just like the discourse model this model needs to be provided by the designer of the system. It specifies all the classes and their hierarchical structures typical for the domain the system is built for. Objects or their attributes defined in the domain of discourse model are used to fill the communicative acts' content field. Communicative acts are used to store and transport information between the application logic and the user interface. Figure 3.10 presents the domain of discourse model of the online shop application. All the classes needed for the

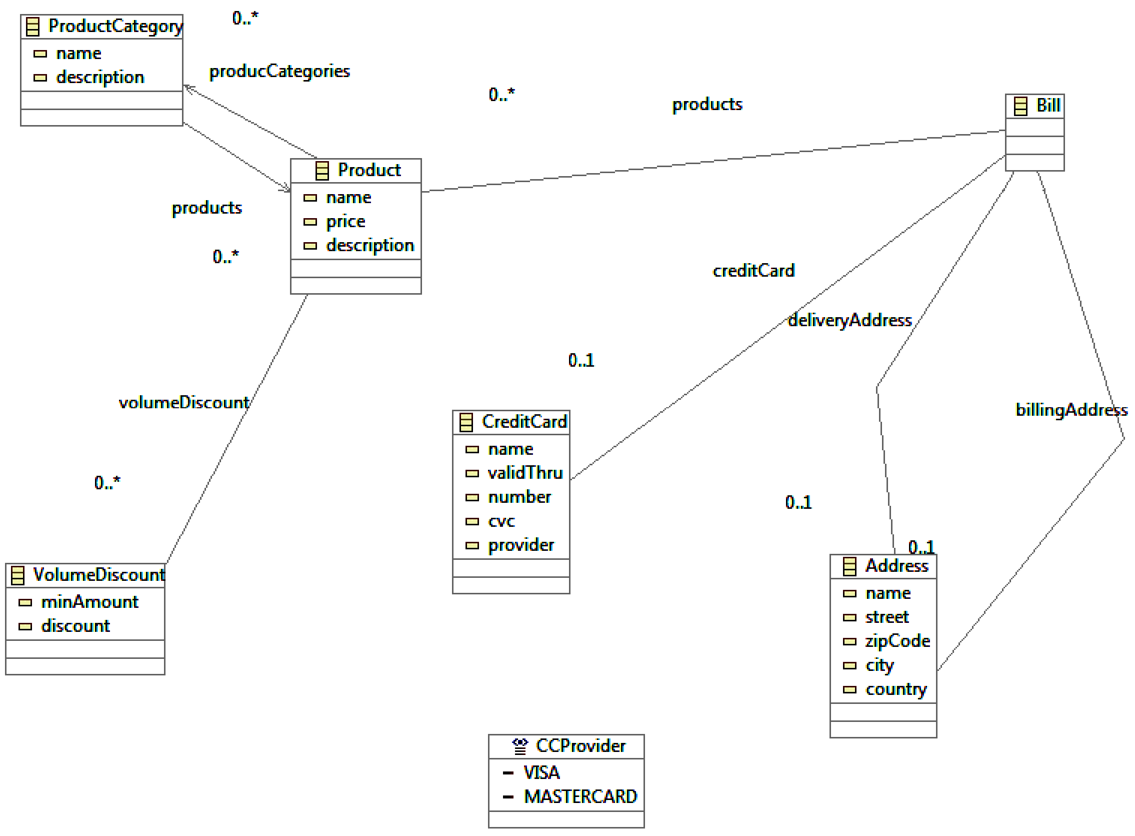


**Figure 3.10:** Online Shop Domain of Discourse Model

execution of the online shop are defined in the UML class diagram presented in Figure 3.10. This diagram has been designed as an Ecore diagram, which allows generating the Java source code automatically. To create an instance of such a class during the runtime of the system, the associated content factory has to be used.

The application logic uses these classes to provide the user information with information that should be displayed. An example would be ProductCategory, which carries the information to be displayed by the Informing communicative act in the objects description field. Further examples would be all the attributes of Product that are related to different communicative acts of the THEN branch subtree presented in Figure 3.8.

The user interface itself instantiates objects defined in the domain of discourse model as well. It uses them to store information collected from the user and subsequently sends the objects to the application logic. Examples would be billing as well as delivery address.

The class CCProvider is an enumeration, holding all the credit card providers available for payment. The user interface offers this list to the user and puts the selected value in the CreditCard class' provider attribute field.

What is of great interest concerning the model-to-code transformation is the connection between the domain of discourse model and the abstract user interface description. More details on this topic are given below.

# Chapter 4

# From Abstract User Interface to Java Swing

As presented in Chapter 3, the whole generation process can be divided in two steps. The first step takes the discourse model as well as the domain of discourse model as input and generates, apart from components needed for the communication during runtime, an abstract user interface specification.

As soon as the first step is completed the code generator translates the abstract user interface description to source code. This code provides all the windows and frames that are needed to communicate with a user. Besides a component needed to enable the communication between formerly generated components is created. This component implements the Model-View-Controller (MVC) principle, which separates the tasks to be fulfilled by every module. This chapter elaborates on the second step of the generation process – the actual source code generation.

## 4.1 The Code Generator

The code generator's task is the transformation of an abstract user interface specification to source code of a specified programming language. This structural user interface specification is provided as a model which is an instance of a meta-model. In other words, what has to be completed is a model to code transformation. As graphical user interfaces normally consists of a limited number of widgets that are combined in various ways, the resulting code can be seen as a combination of similar code fragments that differ only in their sequence of appearance.

The most appropriate generation approach is the combination of meta-model and templates, presented above. As large parts of the OntoUCP project are based on the Eclipse Modeling Framework and, therefore on Java, JET was chosen as template language. The code generator itself was implemented as a template engine, using Java Swing as target toolkit.

Figure 4.1 illustrates the basic approach of the code generator. All the structural information regarding the user interface is specified by the abstract UI specification. It consists of an ordered set of the WIDGETS presented in 3.2. Its structure forms a tree, a CHOICE making

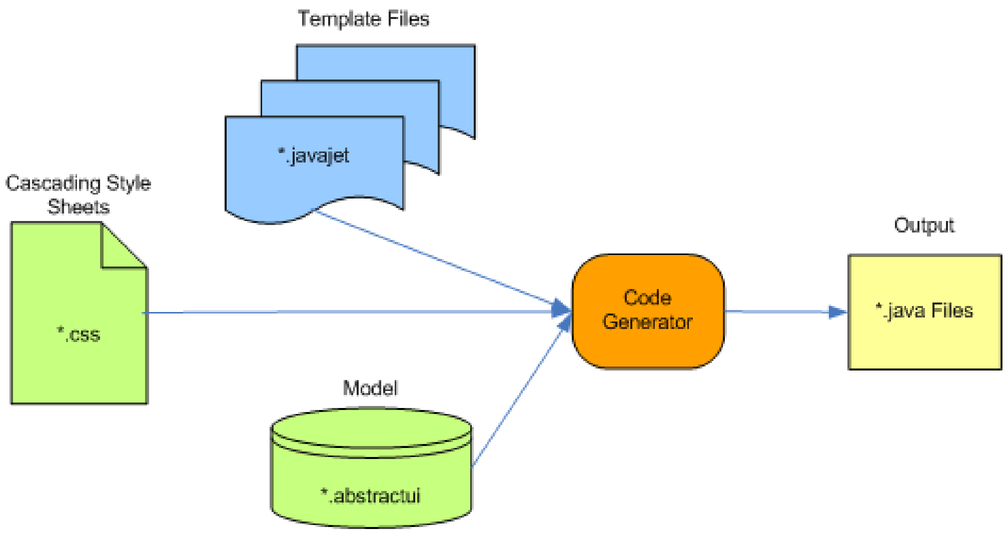


**Figure 4.1:** The Code Generator

up the root. As FRAMES are the abstract representations of windows, any WIDGET one level below the root CHOICE needs to be such a FRAME. These FRAMES contain all the WIDGETS needed to interact with the user as specified in the discourse model. If an application requested only one window there would be no need for the root being a CHOICE, but as the abstract UI is generated automatically, there's no problem to add this CHOICE to provide more flexibility. All considerations concerning a meaningful distribution of all the WIDGETS needed to interact with the user to accomplish the tasks designed in the discourse model, have to be made during the generation of the abstract UI. As this description is the result of the first generation step, it is based on the discourse model as well as on the domain of discourse model.

The discourse model defines the sequence of interaction and the domain of discourse model provides the objects needed by the system to describe actions in its domain of application. Using EMF, both models can be defined as Ecore diagrams, resulting in automatically generated code for its classes. A content factory is automatically generated, which enables the programmer to instantiate these classes.

The abstract UI specification is an instance of a meta-model. As this meta-model has been created using Ecore as well, such specifications can be created or modified without much effort by using the editor provided by EMF.

Figure 4.1 shows that the input includes Cascading Style Sheets (CSS) apart from the abstract UI model. This mechanism allows the separation of functional content and design. All the style information is bundled in these files and mapped to target platform specific values by the generator at generation time. Another advantage is that the layout of the resulting windows can be changed quite easily by modifying the style sheet and generating the source code anew. The generation of the abstract UI does not have to be repeated.

All the remaining specifications needed by the code generator, can be put in a properties file or object. This item is created automatically during the first step of the generation process and passed on. Information contained by this object ranges from runtime specific settings to values concerning the package and path of where to save the generated files or where to find images to be included.

### 4.1.1 The Code Generator Implementation Class

This class poses the core element of the code generator. Its *generate()* method is called during the generation process and performs all major steps needed to transform the input data to correct Java Swing source code. Because JET offers the full Java syntax, some functions were outsourced to the templates to keep the code well structured and easily maintainable.

The generator class takes two input objects, the abstract UI specification and the properties object. The additional style information is not obligatory. The two mandatory objects are the result of the first step of the generation process. The code generator combines this information with suitable templates that have to be provided in advance, to produce the desired output. Implementing the code generator as a template engine keeps this component target toolkit independent and allows the addition of another target toolkit by simply providing the appropriate templates. The generator implementation class extends an abstract class called AbstractGenerator. This class contains method declarations and the code formatter, that is used to give the generated output String a more readable format. In the following the methods implemented in the GeneratorImpl class and their function are presented.

#### 4.1.1.1 *generate()*

This method contains the core functionality of the module and is called to complete the generation step. The two input objects resulting from the first generation step are handed to this method. The abstract UI description is passed on as an IFile and the properties object as a map containing the key as a `String` and the corresponding object.

The first action of this method is to load the abstract UI description. As it is structured as a tree, the root CHOICE can be accessed by getting the element with the index 0.

Subsequently all WIDGET names are transformed to be unique. This task is completed by simply attaching a sequence number to every WIDGET'S name. This way the WIDGET'S name can be used as variable name in the source code later on.

The next step is to create a map named "messages", which will be filled with all the text contained by any widget of the abstract UI description. This map, of course can not contain text from widgets whose content is filled during the runtime of the system. Its values are written to a properties file in the end. During the runtime of the system all widgets that do not obtain their text from the content of a communicative act, get it from this file. The advantage of investing this little extra effort is that the language used by an application can easily be changed by replacing the messages.properties file. All that needs to be translated are the values displayed by the associated widgets.

As the separation from content and style has proven to be of great value not only in the Web but in all applications, style sheets have been introduced. The details on which CSS values can be defined and to which Java Swing values they are mapped are presented in section 4.1.3.

For now the *generate()* method tries to load a CSS file from a location given by in the launch configuration of the code generator. If any file is found it is loaded and used to set the widget's style attributes. Missing style information does not pose a serious problem, as all widgets provided by Java Swing are equipped with a default value.

Once all these preparative steps have been completed, the next step is to start the actual code generation process. All WIDGETS of the abstract UI tree are dealt with one after another, starting with the root CHOICE once more. As a CHOICE poses a WIDGET that does not produce any source code, the real start of the process is one level further down the tree. This level contains all the FRAMES that are translated into the application's windows. To generate the actual source code, a *generateFrame()* method has been implemented, which returns the code as `String`. Each window extends Java Swings `JFrame` and therefore needs to be generated as its own class. As classes need to be stored in their own *.java file, a *writeJavaFile()* method has been implemented. In case the root WIDGET of the abstract UI specification is not of the type CHOICE the generation process is stopped. Such circumstances would require a new generation of the abstract UI specification.

So far, all code needed to display the actual windows has been generated. What is still left to do is to generate a component needed to complete the implementation of the MVC pattern. The model, as well as the control component, are either generated during the first generation step or taken care of by other modules. This leaves the view component to be implemented. By invoking the *generate()* method of the appropriate template, this task can be completed easily (for details see 4.1.2.4). The interoperability of these three independently generated components is assured through the definition of interfaces.

Two more features have been included in the code generator, which require generated code as well. The first one has already been introduced and is based on the messages map. Its generation requires the creation of two more template instances that result in a generated Java class and the messages.properties file. The Java class provides the *get()* method that is needed to access the values stored in the messages.properties file. The file itself contains the actual values, loaded during the runtime of the system.

As not all content to fill the user interface is already available at generation time, the generated code can not simply be compiled and run. To actually display the windows, the communicative acts that provide the information to the window's widgets need to be provided. Therefore, an extra class named Preview is generated automatically, which allows the verification of the generated source code. This class provides all the needed communicative acts, which themselves provide the content objects needed by the widgets. By default their content is set to null, but a mechanism has been included to load content objects during the execution of the Preview class.

If all the above presented steps have been executed successfully, the structural code generation for Java Swing has been finished!

### 4.1.2 From Structural UI Widgets to Java Swing

This section is dedicated to the actual generation process i.e., it describes what happens as soon as the *generateFrame()* method is called and presents subsequently called methods as well as the templates used by this methods.

Java, as an object-oriented programming language, supports modular programming. Using JET, this principle together with encapsulation was used to transfer huge parts of the WIDGET-specific logic, as well as toolkit specific issues, from the generator to the corresponding template. This way the generator implementation class has been kept almost independent of the target toolkit. Hence it can easily be modified to generate code for a different toolkit, by providing the appropriate templates.

JET does not offer its own template language, but allows the user to include any kind of Java code by using scriptlets or expressions (see section 2.2.4.1 for details). Due to this reason the template mainly consists of Java statements that differ only in their time of execution (i.e. generation-time or runtime). All logic needed during the runtime of the system is treated as a `String` during the generation. The parts needed to translate the template correctly need to be embedded as scriptlets. The return value of any embedded expression needs to be a `String`, because this `String` is directly appended to the output. Specific JET Editors ease the distinction of which code belongs to which time by underlying the source code with different colours.

#### 4.1.2.1 Inclusions

By exploiting the possibility of including any file in a template via JET's *include* directive, repeating procedures can be modularized. This way the readability of the templates is improved and the amount of code that has to be written can be reduced.

All such files that encapsulate a certain functionality, are created as simple text files. To distinguish them from proper template files their ending has been assigned as jetinc. Various inclusions have been written for the code generator. They appear in almost any *.javajet file and are, therefore presented before the actual template files.

**inc_header**

This header is added to all the templates and only relevant at the time of generation. The argument passed on to the *generate()* method of the template is an array of objects. The first element of this array is always the WIDGET the code is generated for. The header creates a variable of the type WIDGET and casts the first element of the object array to this type. Furthermore two local `String` variables named jName and jPanelName are created. jName is filled with a unique name for any WIDGET to be generated and jPanelName contains the name of the WIDGET'S parent, if it has one. These three variables are available in any template and they allow the modularisation of code in further JET inclusions 4.1.2.1.

**inc_addToPanel**

Any Java Swing widget that is contained by another widget, needs to be added to its containing component. The only component that is never added to any container is the frame itself. This header's purpose is to add the appropriate line of source code, depending on the type of component the widget needs to be added to. Possible containers are `JTabbedPane`, `JPanel` or the content pane of the JFRAME itself.

Moreover it needs to be checked, whether the WIDGET is equipped with layout data or not. Neither FlowLayout nor XYLayout need this information, but in case of GridBagLayout the additional constraints need to be included in the *add()* method as well.

### inc_getCaContent

During the runtime of the system, the communication is based on sending and receiving communicative acts. Each of these communicative acts has a unique id and carries the information as an object, defined in the domain of discourse model of the system, in its content field.

Any OutputWidget whose content is filled dynamically during the runtime of the system extracts the information it displays from the communicative act specified in its *tracesTo* field. Frequently one Widget does not display the whole object, but only the value of one of the object's attributes. The information which attribute field should be selected is specified in the Widget's *content* field.

Any InputWidget stores the information gathered through the user interface in an object defined in the domain of discourse model as well. As soon as required, this object is sent to the application logic. The object is passed on as content of the communicative act specified in the Widget's *event* field.

The extraction of the information and its storage can be handled the same way for all Widgets and are therefore perfectly encapsulate able in an inclusion. One task completed by the code contained by the ind_getCAContent.jetinc, is to check how the information needs to be retrieved from the communicative act in case of an OutputWidget. For InputWidgets the object to store the collected information is created. As several abstract UI configurations can be found for both widget types, the inclusion has been designed to offer the appropriate lines of code for each of them.

Due to the classification hierarchy of the widgets presented in 3.2, the inclusion only needs to distinguish between Input and OutputWidgets. As InputWidgets may not only display information but primarily collect it, they need additional functionality compared to the OutputWidgets. The process of data retrieval is the same for both widget types. The communicative act to be used is specified in the Widget's *tracesTo* field, its content value specifying what should be displayed. Concerning the type of this value, three different scenarios are possible:

1. The whole object can be delivered by the communicative act. In theis case the object is an instance of EClass, no additional code is required, because the needed object is returned directly by calling the *getContent()* method on the communicative act.
2. Alternatively, the value can only be an attribute of the communicative act's content object. If the content value is an instance of EAttribute, the returned object needs to be cast to the type of the containing class. The required value can be extracted subsequently by calling the appropriate *get()* method on this object.
3. In the last case the value is just a reference to another object. EReferences just like EAttributes require to cast the received object to a certain type. In this case the reference can either point to an object or to an attribute of an object.

According to which case is found, the inclusion adds the needed lines of code to retrieve the information correctly.

One further mechanism that has been implemented in this file creates an instance of the object found in the corresponding communicative act's content field. The object is only created if it does not already exist, otherwise the existing object is used by the widget. As OutputWidgets only read from received objects, this function is only needed by InputWidgets. Frequently, the gathered information can be matched to an attribute of an object defined in the system's domain of discourse model. Therefore, the input widget checks whether the object needed to store the information already exists. If not, the object is created and the attribute value is set. In case the object already exists, the attribute is set in the existing instance. The whole object is always stored within the corresponding communicative act's content field. As soon as required, this communicative act, as specified in the Widget's *event* field, is sent to the application logic.

Of course the value to be set can not only be an attribute, but an object or a reference as well. Therefore, the inclusion distinguishes those cases and adds different lines of code according to the prevailing situation.

The OntoUCP project currently relies on Ecore models for describing the domain of discourse. Therefore, information on which package includes the factory to create instances of any object defined this way needs to be provided as well. This information is passed on in the properties file handed to the generator.

#### inc_setLayoutData

Three different types of layout are defined in the abstract UI meta-model. The only layout manager that requires additional constraints is Java Swing's `GridBagLayout`. The code contained by this header checks whether the Widget has GridLayoutData attached or not. If so, the values defined in the abstract UI are mapped to Java Swing attributes as specified in 4.1.2.3. Finally the constraints are added to the widget they belong to. The line of code needed in the template is the same for every widget as any widet's name is stored in the variable jName.

#### inc_setPreferredSize

This inclusion has no functional capacity. Its only purpose is to influence the layout. Java Swing sets the size of a component according to its "PreferredSize" value. If this value is not set, the actual size is calculated by the layout manager and differs for every widget. A label's size for example, is set according to the length of the text it displays. This way every widget's size is unique. To unify the appearance of the generated screens, the width of any widget is set in advance, with the height to be calculated during runtime.

The width is set to a third of the frame's $x$-resolution minus a tenth of the same value. This leaves a margin of one twentieth at each edge of the frame, even if three widgets are displayed next to each other in the same row. The $x$-resolution is already available in the abstract UI specification, hence this value is set at the time of generation.

The height of the widget is calculated dynamically according to the length of the text that needs to be displayed. Another parameter that influences this value is the size of the font the widget uses.

If the text is given in the abstract UI specification as well, no font size is set and it is shorter than the width of the widget, the height is set to the widget's Java Swing default value. If the text can not be displayed in one row, the height value is adjusted and the text is displayed in as many lines as required. To implement the required line break, Java's support of HTML has been exploited. Any widget's text is set as HTML text, using the HTML tag to center the text.

In case the content of the widget is only known at runtime, the height is calculated dynamically. The following formula is used to compute the factor with which the default height value is multiplied:

$$factor = \frac{widget.getPreferredSize().getWidth()}{\frac{screenResolution - screenResolution*0.1}{3}} + 1$$

1 is added to the result of the division because the type of factor is cast from `double` to `Integer` subsequently. The size of the font used by the widget is considered by checking whether its value multiplied by two is higher than the Java Swing default value. If so, the default value is doubled.

By using this formula to calculate the heigth dynamically, the widgets' dimensions and therefore the whole panels' design is unified. The preferred size differs from its default value only in case the text to be displayed does not fit in one line. The values used in the formula should be seen as default values and can, of course be adjusted if needed.

### inc_setStyleData

The style aspect is something that does not influence the logic of a system at all and can be treated separately for this reason. Cascading Style Sheets (CSS) have been designed to fulfil exactly this purpose. They encapsulate all information concerning the design of an object, making it easy to adapt a system to a new layout by simply exchanging the style sheet. As CSS have been designed for HTML pages, the attributes need to be mapped to Java Swing attributes by the generator. The CSS attributes supported so far as well as the mapping are documented in detail in 4.1.3.

According to the abstract UI meta-model, all Widgets can be given additional style information. The code needed to set the appropriate attributes, is the same for all the widgets and has therefore been put in the inc_setStyleData.jetinc. This generalisation is possible due to the header inclusion presented before. As style is a WIDGET'S attribute, it is inherited by all classes derived from WIDGET, but can be accessed as well by casting any WIDGET to its base class. This is exactly what the header inclusion does.

Using Java Swing, the attributes can be applied directly to the widget instances. The only exception is a `JFrame`, where the options need to be set for the object's content pane. Apart from this mechanism, this inclusion sets the default values for font type and size, as well as for border thickness and colour. It is included only if a style sheet is available. In any other case no attributes are set and the Java Swing default values are used.

Apart from the attribute matching, this inclusion is the fall-back mechanism known from HTML. If any widget has a valid style id attribute, the generator tries to extract all the values to be set from this CSS class. As classes in CSS frequently extend elements, the inclusion tries to find the attribute value there if nothing is specified in the CSS class. If both attempts fail, the value is set to a default value specified in the inclusion. An example is the following extraction of a style sheet:

```
.heading {
    font-weight: bold;
    font-size: 14;
}

label {
    font-size: 12;
    font-style: italic;
    font-family: Dialog;
    color: rgb(0, 0, 0);
    background-color: rgb(99, 248, 16);
}
```

The abstract UI defines a label's style id as .heading. The code generator then tries to extract all the attributes from this class. Font-weight as well as font-size can be found. As there are no values specified for the rest of the attributes, the generator starts looking at the element defined by the widget's name. This way the background as well as the foreground color and the font-family are set. This saves the designer the work of specifying the values twice. Instead, they are inherited and only the new values or the ones to be changed need to be introduced or overwritten.

The actual access to the style sheet values is provided by a style sheet parser, that has been developed as a free software project named CSS4J [Ame08]. This parser implements the Document Object Model (DOM) Level 2 CSS interface specification published by the World Wide Web Consortium (W3C) [WHA00]. Apart from parsing the style sheet, the interface provides objects for easy handling of frequently needed attributes (e.g. an RGBColor object).

#### 4.1.2.2 Template Classes

So far the general generation aspects have been presented. As these functions are stored in *.jetinc files they are not translated by the JET engine. This would not be possible anyway as they do not contain the obligatory JET directive. The following subsection is dedicated to fully fledged templates that are translated to common Java classes by the JET engine. These classes are instantiated by the corresponding *generateWidget()* method of the generator. They contain the inclusions presented in the previous section and return the code to be generated as a `String`.

Before the actual template class is instantiated, the code generator needs to verify the type of the WIDGET. This task is completed by the *generateWidget()* method. It consists of a row of if-statements that check which class the WIDGET is an instance of and invoke the widget-specific *generate()* method. The only issue that needs to be considered concerning this part of the generation process is the sequence of the if-statements. Due to the hereditary hierarchy, defined in the abstract UI meta-model, all WIDGET'S are derived either from INPUT or OUTPUTWIDGET. This is no problem as these classes are abstract and no code is generated for them. What matters is the sequence of WIDGETS that produce output code. The check whether a WIDGET is of type CHOICE needs to be made before the check whether it's of type PANEL. A glance at the meta-model reveals that CHOICE is a specialization of

PANEL. If the check is made in the reverse order, any CHOICE results in the invocation of the *generatePanel()* method.

The remainder of this section is used to present the widget's templates. The order is the same as in the *generateWidget()* method. Primarily, all WIDGETS derived from PANEL are presented, hence PANEL is the last one of this group. TABCONTROL is checked first as it is derived from CHOICE, which itself inherits all characteristics from PANEL. The second group is HYPERLINK preceding BUTTON for the reasons given above. From then on the order does not matter anymore, as the entire rest of WIDGET'S is derived directly either from the abstract widget type input or output.

So code generator supports code generation for the abstract WIDGETS presented below. The abstract UI meta-model defines their attributes as well as the relationships among them. This information is of relevance only to the code generator and to the templates. It is of absolutely no concern to the generated code itself.

The following paragraphs explain exactly how those abstract WIDGETS are mapped to Java Swing objects. For some of them, Java Swing already offers corresponding Java data types, others had to be constructed using Java Swing types to fulfil a requested task. An example is TABCONTROL versus HYPERLINK. The first construct can be mapped directly to Java Swing's `JTabbedPane` whereas the second construct needs to be simulated by using a `JTextfield`, implementing the functionality known from HTML in a MouseListener.

### TabControl

A TABCONTROL widget is gives the user the possibility to switch between different panels on the same screen. Such functionality is offered by the Java Swing class `JTabbedPane`. A TABCONTROL provides just a container for other PANELS. Each of those PANELS is generated as Java class. This is why all that has to be done by the `JTabbedPane` is to create an instance of the contained class and add it to itself. A TABCONTROL is always containded by another PANEL or FRAME and therefore usually requires additional layout data. Header, panel addition, style and layout attributes are set by using the appropriate inclusion files.

### Choice

A CHOICE is always followed by at least one FRAME or PANEL. This WIDGET contains no information relevant for the user interface and, therefore does not result in any generated code. Due to this fact the *generateChoice()* method does not invoke a template class but only calls the *generateWidget()* method anew.

As presented in 3.2, this WIDGET always forms the root of the abstract UI tree. This root CHOICE can only be followed by FRAMES. This is due to the fact that all windows needed by the application are defined on this level. If the CHOICE appears somewhere within the abstract UI tree, the ensuing WIDGETS need to be PANELS.

A CHOICE implies that not any of its two successors can be displayed at the same time within the containing element. This is why it does not directly influence the generation of code relevant for the user interface, but the chain of events and therefore the state machine.

### ListWidget

Just like Choice this Widget is derived from Panel and, therefore acts as a container for other Widgets. Each entry of the list consists of the Widgets placed right beneath the ListWidget in the abstract UI tree. The number of entries is variable because the content of a ListWidget is only filled during runtime. This separation of structure and content adds a good amount of flexibility to the application. All entries can be added, deleted or exchanged without the need to generate the system anew.

In Java Swing the ListWidget has been realized as `JPanel`. The layout manager is set to `GridBagLayout` with fixed GridBagConstraints. The widgets that make up one list entry are put in another panel. This means that a ListWidget does not directly contain the widgets but only the panels that make up an entry each. These panels should be placed one beneath the other, so the GridBagConstraints of the list-panel are set to one column and the number of rows is adjusted to the number of entries during runtime. Each panel representing one entry should display the components contained one after another. Therefore these panels' layout manager is set to `FlowLayout`.

The attributes positioning the list-panel correctly in the containing panel are set by including the inc_setLayoutData.jetinc file. Apart from this, only the header and addToPanel inclusion are employed.

### Panel

A Panel is generated as extension of Java's `JPanel`. As every Panel is generated as a Java class, it results in a *.java file, which contains the code for Widgets contained by the panel as well. This implies that the code generation has to be done recursively, generating the code for the Widgets first, handing it to the panel template in the end. In case the Panel itself contains another Panel, it is not necessary to add the code for the contained Panel, because the sub-panel will be stored as a Java class in its own file. Only the lines of code to create an instance of the sub panel which is added to the containing one need to be appended.

As said above a panel's function is to work as a container for other widgets. Frequently these widgets need to put or extract information from communicative acts. Due to this reason the panel's constructor needs to be passed on an `EventHandler`. This object is stored as a global variable within the panel class. To make the EventHandler easily accessible to all widgets the panel provides a *get()* and a *set()* method.

The `String` returned by the panel template's *generate()* method is written directly to a *.java file with the panel's name. During runtime, these panels are instantiated and added as required to the displayed windows.

Header, style and layout attributes are set by using the appropriate inclusion files.

### Frame

This component acts as container for the panels to be displayed and represents the window seen by the user. Looking at the abstract UI tree such Widgets can only be positioned right beneath the top level Choice because windows can not be contained by other windows. Any instance of this WIDGET is based on Java's `JFrame` and implements the interface `UIWindow`. This interface ensures that the *showScreen()* as well as the *getPanel()* method is implemented.

The method *showScreen()* receives an `EventHandler` and the name of the screen to be displayed as arguments. A dynamic classloader is used to create an instance of the panel. Subsequently the window checks whether it already contains an instance of the panel or not. This way the new panel either substitutes the old one or is simply added to the window.

The second method returns a panel which simulates the window's content pane. All widgets contained by the window are added to this so-called "base panel" instead of directly adding it to the window's content pane. This trick is necessary to be able to display the window's content in another window. The CommRob project[1] requires this functionality, because the application consists of more than one discourse executed concurrently. So far these discourses are rendered separately and therefore the WIDGETS found beneath the root CHOICE are FRAMES. Using the *getPanel()* method the content of these frames can be displayed in parallel in one big window.

Just like panel instances, the window instances are created dynamically using a classloader. To create such an instance, the widget's name is taken as a STRING. This way it is impossible to know whether the object to be instantiated is of the `UIWindow` or `JPanel`. To solve this problem, constructors with the same arguments are used in both cases. The type of the created object can then be checked and the object is either directly added or, in case it is a `UIWindow`, its "base panel".

The look and feel of each `UIWindow` is adapted to the platform it is displayed on and its resolution is set according to the values found in the abstract UI specification. Header, style and layout attributes are set by using the appropriate inclusion files.

### Hyperlink

This WIDGET simulates a hyperlink as known from HTML (i.e. a given text should be sensitive to mouse clicks and represents a link to some other part of a document, screen,...). As soon as the mouse pointer enters the area, the cursor changes from standard to hand cursor, to signal that the underlying text is a link. As soon as the mouse button is clicked over the text, a communicative act containing the link's text, is sent to the application logic.

As Java Swing does not offer such a construct, the hyperlink has been implemented as a `JTextField` where the attribute editable is set false. The text to be displayed is either taken directly from the *text* attribute field of the abstract UI WIDGET or from a specified field in a given communicative act. As with any other objects derived from WIDGET this communicative act is specified in the *tracesTo* field.

Header, layout, style and the panel addition inclusion are used by this template.

### Button

This WIDGET is directly mapped to Java Swing's `JButton`. It's text and the action to be performed on clicking the button are set according to the corresponding attribute values set in the abstract UI description. Any text that is specified in the BUTTON'S *text* field is written

[1]http://commrob.eu

to the messages.properties file and retrieved as soon as needed during runtime. This way the label of any button can be changed easily by modifying the properties file. In case the text is set dynamically by retrieving information from an object received by a communicative act, the label is set to <default> in the properties file.

As soon as pressed each button should send the communicative act specified in its *event* field. This communicative act carries the content object associated with the pressed button. To be able to set these values in the action listener belonging to this button, they are set as client properties to the widget.

In case the BUTTON'S *tracesTo* field is found blank the object to be sent is collected from the event communicative act's adjacency pair. This configuration is found if the button is used to send data collected by text fields, for example. If there is more than one communicative act specified in the *event* field the button sends them all.

Finally a `CommActListener` is added to each button. This mechanism allows to update the button's content independently from the rest of the containing frame. As soon as the associated communicative act is received anew by the `EventHandler`, it updates all the widgets registered for this communicative act. This way any button's information can be kept up-to-date even if the screen stays the same.

All style and layout attributes are set by using the appropriate inclusion files. As BUTTONS are frequently contained by LISTWIDGETS its preferred size is set by using the inc_setPreferredSize.jetinc inclusion. This is set alongside with the preferred size for labels and results in a well structured and easily well arranged layout. Moreover the header and container addition inclusion are used.

### Label

A LABEL'S task is to display given information and it is, therefore derived from OUTPUTWIDGET. It is directly mapped to Java Swing's `JLabel`. In case the abstract UI label's attribute *text* is set, this text is added to the messages.properties file and retrieved during runtime, just like for buttons. If no text is available the properties value is set to <default> and the actual value is retrieved from a communicative act during runtime.

Once again a `CommActListener` has been added. This is especially important for labels as these components are frequently used to display status information. The principle is the same as for button.

Header, container addition, style, layout and preferred size attributes are set by using the appropriate inclusion files once again.

### TextBox

The TEXTBOX widget has been mapped to Java Swing's `JTextField`. A common use for text boxes is to collect information, which is stored in an object. Frequently these objects have several attributes, which means that several textboxes are needed to collect the information for one object. This implies a different *content* attribute for all the abstract UI widgets, which makes it impossible to include them in a LISTWIDGET. Therefore the preferred size is not calculated using the inclusion but defined in the `JTextFields` constructor.

Apart from collecting information, a TextBox can display text as well. This text is editable and taken once more from the messages file. If no text is specified in the abstract UI, <default> is used if the text box type is `String` and 0 for `Integer` and `double` values.

Frequently more than one text box is used to store information that belongs to one object. To keep this object consistent and up-to-date, every text box works on the same instance. As soon as the text box loses the focus, its text is stored in the corresponding object's attribute field. The object itself is always retrieved from the communicative act specified in the TextBox's *tracesTo* field. The object is created as soon as the first entry is made. From then on it is only retrieved and updated. All these tasks are completed by inserting the inc_getCAContent.jetinc inclusion.

All input retrieved from a `JTextField` is gathered as a `String`, but sometimes the attribute type of the object may be `Integer` or `double`. This makes explicit parsing necessary to avoid a type mismatch. Header, the correct lines for adding the widget to its container, style and layout attributes are set by using the appropriate inclusion files.

### ComboBox

This Widget's objective is to provide a list of certain values for selection. If the focus lies elsewhere, only the selected item is displayed instead of the whole list. The corresponding Java Swing widget is `JComboBox`.

The values that fill the combo box need to be specified in the domain of discourse model of the system. This enumeration can then be referenced as an attribute by a content object. A positive side effect of this mechanism is that the values are not put in the source code and can, therefore be altered without any problem during runtime. A `focusListener` is used to update the object attribute with the latest selection.

Header, layout data and the right adding to the containing object are accomplished by using the appropriate inclusions. Besides the inc_getCAContent.jetinc is used to retrieve and set the content of the associated communicative act correctly.

### ImageMap

The idea of an ImageMap is to make specified regions of an underlying image sensitive to mouse clicks. An example is a map of a zoo with the area of an animal's cage being sensitive to clicks. As soon as the user selects an animal he receives background information about the chosen species.

As Java does not offer any corrisponding construct, the widget has been created by using a `JLabel` combined with a `mouseListener`. The picture needs to be provided by the creator of the discourse. Moreover, he needs to define the interactive areas in absolute coordinates. The left, upper edge of the picture is used as point of reference (i.e. (0, 0)). The information of where to position an object on the map is usually defined as attribute of this object. This way the association between object and sensitive area can be set easily in the `mouseListener`. As soon as a mouse click occurs this listener checks whether it has been within any sensitive area or not. In case any of these conditions becomes true, a communicative act with the corresponding object as content, is sent to the application logic.

Header, layout data and adding the widget correctly to the containing object are accomplished by using the appropriate inclusions.

#### PictureBox

A PictureBox is intended to display a simple image without any interactive task. Just like Label it is derived directly from OutputWidget. Using Java Swing it is generated as `JLabel` without text, displaying just an icon. The pictures to be displayed must be provided by the discourse model designer. They need to be put in a folder, whose path needs to be specified in the *.properties object passed on to the generator implementation class as soon as the code generation starts.

The attribute *picture* of the abstract UI widget contains the name of the image to be displayed on the screen. Header, layout data and adding the widget correctly to the containing object are accomplished by using the appropriate inclusions.

The source code for any of the Widgets presented so far is generated by invoking the *generate()* method of the corrsponding template. Encapsulating a great deal of the logic needed to select the right lines of code to be generated in the inclusions, leaves almost no logic to be contained by the widget-specific *generateWidget()* method. The task of this method is in most cases restricted to creating an instance of the template class and handing the right arguments to its *generate()* method. These arguments are passed on as an array of objects. They include the Widget itself in any case, the `properties` object and the `styleProvider` object if needed. Moreover the `messages` object is needed by those Widgets that retrieve their content from the equally named file later on. In case the Widget is a container the code generated for its children is handed over as well.

#### 4.1.2.3 Layout Types

Code for all Widgets presented so far, is generated by invoking the corresponding template. Therefore, they correspond directly to one widget on a screen. The layout types are containded as Widgets in the abstract UI description as well. In contrast to direct correspondence they only influence the way the widgets are arranged and therefore their attributes.

The abstract UI meta-model contains three different layout types: FlowLayout, GridLayout and XYLayout. Depending on the layout type, appropriate layout data has to be added to the Widgets, thus defining where each widget is placed within the containing panel. Panels apart from having a layout type might have layout data as well. This defines their position in the containing component.

#### FlowLayout

FlowLayout is mapped to Java Swing's `FlowLayout`. Every component contained in a FlowLayout follows the component added right before. If the end of the window is reached, the widget moves to a new row. Resizing the window might change the position of the widgets. If through resizing the window enough space is gained for one widget to move one row up, all the following widgets are arranged anew as well. If not specified differently, the preferred size value is used for each widget.

FlowLayout is the simplest layout type. It offers the use of constraints but it does not require them. Due to its simplicity and its ease of use it has been selected as the default layout type for `JPanels`. If no layout type is specified explicitly, the code generator uses the `FlowLayout` manager as default as well.

### GridLayout

This layout type is mapped to Java's `GridBagLayout`. Unlike the FlowLayout manager, this type of layout requires constraints, which are represented in the abstract UI description as GridLayoutData. Every widget being part of a `GridBagLayout` has to contain such layout data, otherwise it cannot be placed correctly.

Compared to other layout managers, Java's `GridBagLayout` is very powerful and flexible. Basically all components added are placed in a grid of rows and columns. Flexibility is added as not all of these rows and columns need to have the same width or height. To calculate these values, a component's preferred size is taken into consideration. Placement, resizing behaviour and other things needed to determine a component's display characteristics can be defined by using the above mentioned GridBagConstraints as well. Each of these values has a matching counterpart in the abstract UI description's GridLayoutData.

**GridLayoutData** A GridLayoutData object consists of various attribute fields, which are defined in the meta-model. As the meta-model's GridLayout is mapped to Java's `GridBagLayout`, these attributes need to be mapped to Java's `GridBagConstraints` somehow. The following paragraph states which abstract UI attribute is matched to which Java Swing constraint and gives a short explanation of what their effect is. The first attribute name represents the abstract UI value and the one following the "∼" is the one used by Java Swing.

- *Alignment* ∼ *Anchor*
  This attribute is used to determine where the component should be displayed if the area provided for this component is bigger than its actual size. The attribute's value is set by constants. The abstract UI constants are matched to the corresponding `GridBagConstraints` constants during generation time, the default value in both representations is CENTER.
- *Col* ∼ *gridx*
  This value is a number which defines the column, or the $x$ value, in which to place the component. The leftmost $x$ value is zero.
- *ColSpan* ∼ *gridwith*
  This value defines the number of columns the component is going to span. Using `GridBagConstraints.REMAINDER` guarantees that the component is the last one in its column.
- *Fill* ∼ *Fill*
  Apart from Alignment this value is used if the display area is larger as the component's size as well. It determines whether the remaining space should be filled or not and if so, in which way. Just like the anchor value its value is defined by constants. Possible selections are HORIZONTAL, VERTICAL and BOTH.

- *Row* ∼ *gridy*
  The value of this attribute defines the row or the $y$ value of where to place the component. 0 is the uppermost value.

- *RowSpan* ∼ *gridheight*
  The number of rows the widget spans is defined by this value. Just like gridwidth, `GridBagConstraints.REMAINDER` makes the widget being displayed as the last one in its row.

- *WeightX* ∼ *weighxt, WeightY* ∼ *weighty*
  These values are used to determine how extra space is distributed among components. *Weightx* refers to the $x$ axis and *weighty* provides the values for the $y$ axis. Its default value is 0.0, which means that all extra space is put between the window's grid of cells and its edges. This results in all the components clumping together at the centre of the window. Usually numbers between 0.0 and 1.0 are used to define how much extra space should be given to the specified row or column.

  If more than one component is contained by one row or column, then the highest number is taken into consideration at the end. Normally extra space tends to be given to the rightmost column and to the bottommost row.

  Apart from the windows appearance when displayed for the first time, these two values have also great influence on its resizing behaviour.

**XYLayout**

XYLayout makes it necessary to define the absolute position of each widget within one panel. It requires XYLayoutData which determines a components height and width apart from the left upper edge of its position. In this case resizing the window does not change any widget's position. Using this layout requires some extra work, but is sometimes inevitable. The deployment of any other layout, apart from absolute layout, would change the link positions on an ImageMap, for example. Using Java Swing, a `JPanel` can be set to `XYLayout` by setting its layout manager value to null.

**XYLayoutData** As this layout data does not allow any flexibility, it is quite simple. The coordinates of the upper left edge of a rectangular area are defined by an $x$ and a $y$ value. The widget's dimensions are given by the properties height and width. This way actually no layout manager is needed but the objects are placed directly on the canvas.

#### 4.1.2.4 View Implementation

This component is generated by invoking the corresponding template by the generator main class. It does not produce code needed to display information but completes the Model-View-Controller configuration needed for communication during runtime. Implementing the MVC pattern, three components need to be generated. A finit-state machine is generated automatically from the abstract UI descpription as behavioural model for the whole system . Within this component the system's states are specified as well as the communicative acts needed to trigger a transition from one state to another.

These communicative acts are handled in the control component, called EventHandler. The EventHandler registers and stores all incoming communicative acts. In case a communicative act's content is needed during runtime to get some data to be displayed in a window, the communicative act can be retrieved from the EventHandler. According to the communicative acts it receives from the application logic, the EventHandler triggers state transitions in the state machine.

The third component to complete this triple is the view component. Just like the creation of the state machine, its generation is based on information provided by the abstract user interface model. Its task is to instance the screen set by the state machine. As an application may consist of more than one window, it needs to know which panel belongs to which frame. Its inner structure roughly consists of two hash maps. The first one is used to store which panel belongs to which window. As some of those windows may have been already instantiated, the second hash map provides information on which of them already exist.

At generation time the template is handed over the data of which window contains which panels. This information is used to fill the first hash map. The second one is only filled during runtime. A new entry is made for every newly instantiated window. The constructor of this class requires to be given the EventHandler of the system. This is due to the fact that any window, whose *showScreen()* method is called needs to know where to retrieve the required communicative acts from.

The ViewImpl class only contains one method, which is used to display the available screens. To complete this task, the *setScreen()* method receives a panel's name as an argument. As a first step it checks whether this panel is contained by an already existing window or if a new window needs to be instantiated. In case the corresponding window is already available, its *showScreen()* method is called with the EventHandler and the screen's name as arguments. In the other case an instance of the window is created by using Java's dynamic class loader. Subsequently the newly created window's *showScreen()* method is used to display the requested panel, just like before.

#### 4.1.2.5 The Preview Component

This component is not relevant for any aspect of the running system. Its only use is to display the generated screens without the need for the model and the control component, or the application logic. The preview component becomes necessary due to the dynamic character of widgets like the list. The trial to display any such panel by simply creating an instance leads to null pointer exceptions, because the corresponding communicative acts can not be retrieved from the EventHandler.

The preview component is generated by invoking the corresponding template. Apart from the properties object it receives a list of all screens as well as a list of all communicative acts deployed by the system. The list of screens is displayed to the user and offers him all panels that have been generated. As soon as one item of this list is selected, the corresponding screen is instantiated and shown to the user within the preview window. The panel selected next, replaces the panel shown before and so on. Due to the insertion of the base panel it is possible to display any frame's content as well.

The communicative acts needed to avoid null pointer exceptions as soon as the panel is instantiated, are registered to an EventHandler, that is created by the preview class itself.

During the generation of the component, the list of communicative acts is put into the content of a method that registers those communicative acts to the EventHandler as soon as the preview class is instantiated. The content objects for each communicative act are created and initialised within this method as well. If no values are available, the objects are set to null in the beginning. This makes the widgets get their default value from the messages.properties file. Through the insertion of a *.xmi file this content objects can be set to meaningful values during runtime of the preview component. Clicking a button offers the user a dialogue in which he can select the file graphically. The objects that are found in this file are set as the communicative acts' content subsequently.

If the system's domain of discourse model has been constructed using Ecore, this file can be created easily. EMF includes the possibility to create dynamic instances from any class specified within such a model. This instances' attribute values can be set graphically due to the automatically generated editor, provided by EMF as well.

Finally the String generated by the preview template is written to a file named Preview.java. As this class contains a *main()* method it can be executed as Java application.

Figure 4.2 shows the preview screen generated for the online shop application. The screen selected from the list shown on the left side of the frame is displayed in the right panel. To test whether the widgets have been bound to the right communicative acts, test data has been entered. This test data is stored in a *.xmi file and can be selected via the Browse button. By taking a look at the labels carrying the product description, the correct function of the preferred size calculation can be tested. As soon as the text is too long to be displayed in one line, a second row is opened.

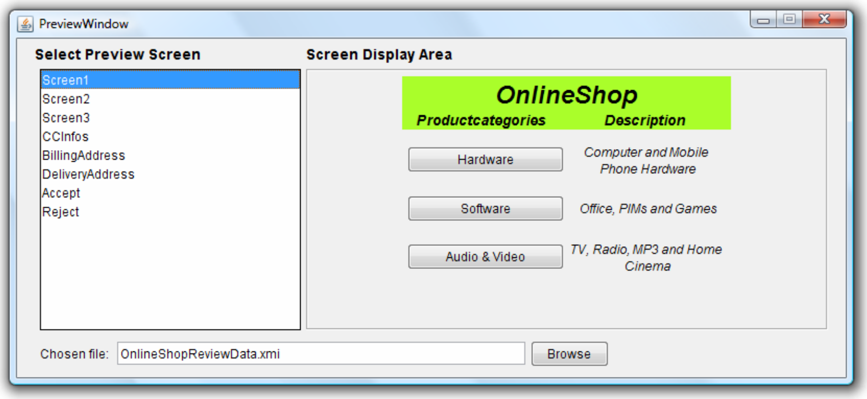


**Figure 4.2:** Preview Window for the Online Shop

### 4.1.3 Cascading Style Sheets

One of the main reasons for not directly generating the source code out of the discourse model was that a user interface must not only be functional but also fulfil certain requirements concerning usability and design. The introduction of the abstract UI as an intermediate step offers the possibility of influencing the layout of the frames more directly. The automatic translation is done according to rules specified in [ABF$^+$06c] and [ABF$^+$07b].

Additional refinement can be accomplished by the system designer after the generation as well. The abstract UI takes usability into account, but hardly contains anything concerning design. All this data can be encapsulated in a style sheet, whose classes and elements are then referenced by the abstract UI.

The code generator parses this style sheet and extracts the needed information at generation time. The style sheet itself needs to be provided by the system designer. To change the style of a system only the second generation step (i.e. the abstract UI to Java Swing translation) needs to be repeated instead of the whole process. The whole mapping logic as well as the fall-back mechanism known from HTML is contained by the inc_setStyleData.jetinc (see 4.1.2.1).

Below the CSS attributes supported so far are presented. Moreover is shown to which Java Swing property they have been mapped. As some values are not directly mapped to a Java Swing attribute but passed on in an object's constructor, default values for some attributes have been introduced.

- **color**
  The value of this attribute specifies the color of the text. It can either be defined as an RGB value or using the hexadecimal notation marked with a #. To set the foreground value in Java Swing the RGB values need to be extracted for each color. This task can be completed with the help of the CSS API that provides a method which returns the RGB values independently of the notation found in the style sheet.

- **background-color**
  The CSS API's RGBColor object is used once more to set the RGB value of this Java Swing property. Additionally the background needs to be set opaque, otherwise no effect can be seen.

- **border-width**
  This is one of several short-cuts provided by CSS. By using this attribute, actually four attributes, one for each side of the border, are set. As all of them are set to the same value it is sufficient to query one of them. In fact Java Swing borders do not allow more than one value to specify its width. As only top, bottom, left and right value can be queried using the CSS API, the border-top-width has been chosen. Hence the Java Swing border thickness can either be set by using CSS' border-width or border-top-width.

  To create a new border object in Java Swing, this value needs to be handed over to the object by the constructor. Therefore a default value has been introduced to avoid NullPointerExceptions. This value is used if no value has been specified in the style sheet. Two pixels have been chosen as a default with the regular CSS attributes matching to pixels as follows:

  - **thin** $\sim$ 1 pixel
  - **medium** $\sim$ 2 pixels
  - **thick** $\sim$ 5 pixels

  If the system designer wants to change these values, he can do so by setting them anew in the inc_setStyleData.jetinc file.

- **border-color**
  Just like border-width, this tag is a short-cut to set the four sides of a border to the

same color. Java Swing does not support this mechanism as it allows only the definition of one color for each border. As border-color can not be queried directly, anyway, the border-top-color attribute has been chosen. Therefore, border-color can either be set by using this attribute or border-top-color. The color definition can be set using either the decimal or the hexadecimal notation of the RGB color schema. As it is only possible to set a border's color passing it on to the object's constructor, the default value black (i.e. rgb(0, 0, 0)) is used. If the border-color has been defined in the CSS these values are overwritten.

- **border-style**
  This attribute is yet another short-cut and, therefore results in querying border-top-style. The border types defined in CSS are mapped to the types offered by Java Swing as follows:
  - **solid** $\sim$ `LineBorder(color, thickness)`;
  - **groove** $\sim$ `EtchedBorder()`; with LOWERED edges.
  - **ridge** $\sim$ `EtchedBorder()`; with RAISED edges.
  - **inset** $\sim$ `BevelBorder()`; with LOWERED edges.
  - **outset** $\sim$ `BevelBorder()`; with RAISED edges.
- **font-family**
  Similar to the generation of a border, the constructor for a new `java.awt.Font` object requires certain parameters to be set. To avoid `NullPointerExceptions`, default values have been introduced once again. These values fill attributes that are required and have been left blank by the designer. If no style sheet is given at all, the Java Swing default values are used. For this attribute, *Dialog* has been chosen as a default value, because it is used by Java Swing as well.
- **font-weight**
  This attribute is used to set the weight of the font. The CSS attributes **normal** and **bold** are supported. Java Swing sets the weight of the font including whether it's italics or not using only one `Integer`. Therefore the selection of the font-weight sets a local variable that is combined with the one set by font-style as soon as the font object is instantiated. Its default value is *plain*.
- **font-style**
  This attribtue is used to set the font-style either to **normal** or **italic**. During the generation of the source code a second `Integer` variable is set. This way a combination of bold and italics can be realized. Its default value is *plain*.
- **font-size**
  The value of this attribute has to be the font size in pixels. Its default value has been set to *12*, as this is once more the value Java Swing uses as a default. Therefore it is unrecognizable for the end user which default value has actually been used and a consistent design has been preserved.

These CSS options presented so far can easily be extended by adding the required code to the inc_setStyleData.jetinc file. Changes are then automatically applied to all widgets due to the modularisation of this fuctionality.

## 4.2 The Online Shop continued

This section resumes the online shop example presented in 3.4. So far the online shop's discourse description and its domain of discourse model have been introduced. It remains to show the abstract UI specification as well as the resulting source code.

A lot of literature can be found regarding the test procedures for new technologies, especially on software. The most demonstrative way to prove the feasibility of any concept is to create a sample application. In our case the online shop did not only show that the system works correctly, but helped to discover and clarify some of the aspects that had not been considered profoundly enough before.

The basis to work on was the meta-model for the abstract user interface. Top down engineering was used as well as the bottom up method, because the automatic generation of the abstract UI description was not completely implemented yet. Due to this reason the first abstract UI descriptions to be turned into source code have been constructed manually. They were created to represent the ideal version of the screens and therefore constituted what the ideal output of the discourse to abstract UI conversion should look like. Another advantage of the combination of these two techniques was that usability criteria had already been taken into consideration. These particular characteristics could then be incorporated in the transformation rules as well.

By implementing concrete screens it sometimes turned out that some more widgets would be needed or that it would be better to combine a few input widgets to one to fulfil a certain task. These changes in the actual screens resulted in changes in the abstract UI of course, sometimes even implying changes in the discourse model. Therefore it became necessary to introduce new transition rules for the discourse model to abstract UI translation.

Apart from these modifications, some aspects concerning the attributes of the abstract UI widgets could be refined during the code generator's implementation. The information about the screen resolution was directly attached to the FRAME instead of being passed on in the properties object, for example. Therefore, the attributes *screenResolutionX* and *screenResolutionY* had to be added.

Some classes as well as attributes could be deleted, because all information concerning the style of the application was outsourced to CSS. The STYLE class remaining in the abstract UI meta-model carries the id of the style to be applied to the widget only. This separation simplified the meta-model and improved the independence between logic and style.

Furthermore some redundant relations were eliminated. Any of this changes needed to be followed by a new generation of the editor as well as the factory. As these steps are automated using EMF, neither of them was too much effort.

One of the main goals in the OntoUCP project is to separate the discourse creation as much as possible from the actual screen representation. It should not be necessary to follow any rules while creating the discourse model, because the discourse model is target independent. If the discourse model is completely self consistent, the output code should be almost as good as hand tailored code. As general rules are applied to the discourse model some overhead is unavoidable. This overhead mainly results in additional choices that are of no importance for the code generation because no code is generated for such a widget. Furthermore some extra panels, which are of no vital importance to the application, appear in the generated abstract UI. This is due to the generalized character of the transformation rules.

In case of critical applications further optimisation can either be done by improving the abstract UI description before the code generation, or the generated code itself. This still saves a lot of programming work compared to complete manual creation.

### 4.2.1 The Application

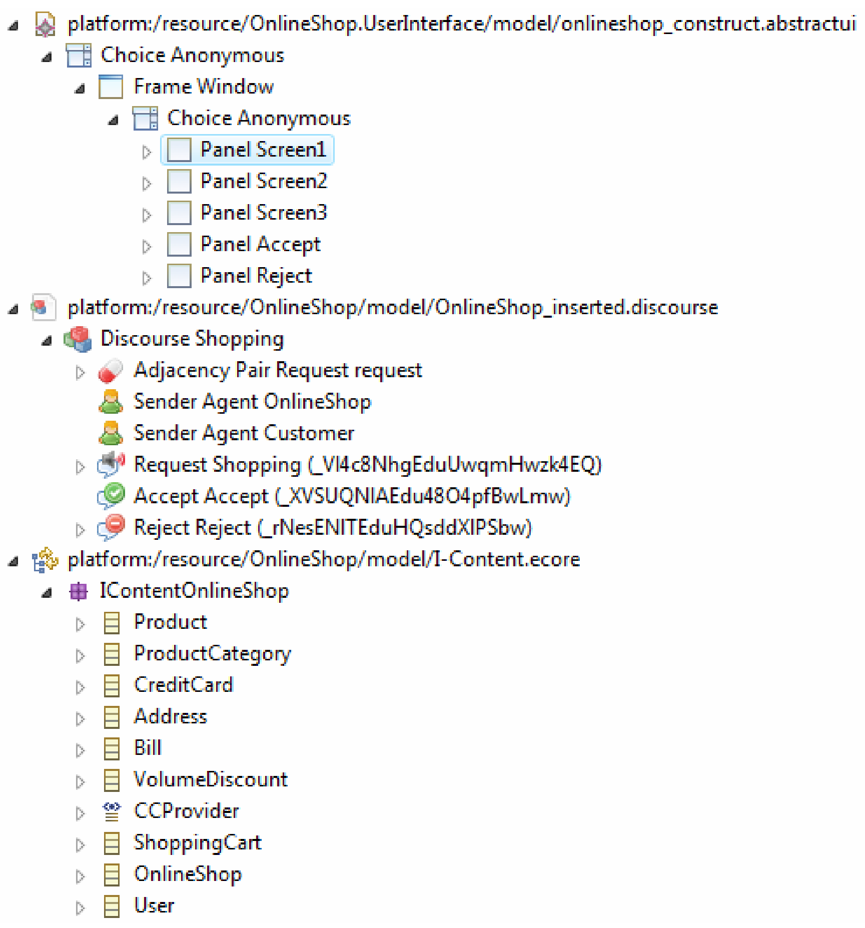
The implementation of an online shop is a good example to start with, because everybody knows in advance what the result should be like. The functionality was stripped down to the very basics, nevertheless separating the structure from the content. That is, the structure of the screens is fixed by the abstract UI specification but the content is filled during runtime, with content delivered by the received communicative acts. A good example is the list of product categories. This way a category can be added or deleted easily, even during runtime, without having any effect on the generated code.

The initial version of the online shop consisted of various screens, which were partially combined to TabControls or recombined, introducing the possibility of a Choice being situated somewhere else than at the root of the abstract UI tree.

The chain of events of the online shop is quite easy and has already been presented in the form of a discourse in 3.4.1. This chain of events is transformed into several screens that contain everything that has been defined in the discourse model, combining it with the information from the domain of discourse model.

The initial screen offers the customer a list of product categories to chose one out of many possibilities. According to his choice a list of products is displayed in Screen2. As soon as the customer selects a product, Screen3 is displayed. This screen collects the delivery and payment details by offering the user an appropriate form. Depending on whether he inserted the data correctly or not, a screen appears telling him whether his transaction was completed successfully or not.

The abstract UI description constructed manually according to the discourse description and to the images we had in mind, is illustrated by Figure 4.3.



```
platform:/resource/OnlineShop.UserInterface/model/onlineshop_construct.abstractui
  Choice Anonymous
    Frame Window
      Choice Anonymous
        Panel Screen1
        Panel Screen2
        Panel Screen3
        Panel Accept
        Panel Reject
platform:/resource/OnlineShop/model/OnlineShop_inserted.discourse
  Discourse Shopping
    Adjacency Pair Request request
    Sender Agent OnlineShop
    Sender Agent Customer
    Request Shopping (_Vl4c8NhgEduUwqmHwzk4EQ)
    Accept Accept (_XVSUQNIAEdu48O4pfBwLmw)
    Reject Reject (_rNesENITEduHQsddXIPSbw)
platform:/resource/OnlineShop/model/I-Content.ecore
  IContentOnlineShop
    Product
    ProductCategory
    CreditCard
    Address
    Bill
    VolumeDiscount
    CCProvider
    ShoppingCart
    OnlineShop
    User
```

**Figure 4.3:** Abstract UI Prototype

The root CHOICE is followed by one FRAME only, which signifies that the application displays all information within the same window. Therefore the root CHOICE could be eliminated without the loss of information. The reason to generate it is the resulting unification of the generation process. Besides the application is kept easily extendable this way.

This FRAME is followed by another CHOICE which implies that only one PANEL at a time, from all PANELS situated subsequently, will be displayed. As there are no other objects than one panel at a time contained by the window, there is no need for a layout manager. Depending on the desired layout, the panels themselves contain appropriate layout managers, of course.

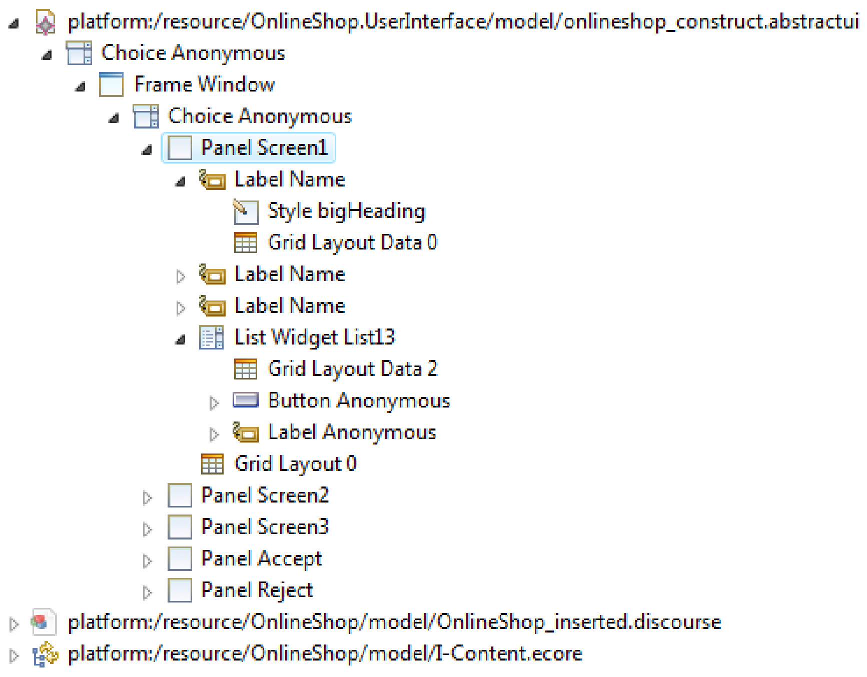
Apart from showing the screen structure of the Online Shop, Figure 4.3 illustrates quite well which information is processed by the code generator. The abstract UI relies on information found in the discourse as well as in the content model. Therefore these two resources are needed as source of information by the generator as well.

A reference to the online shop's discourse model is shown right beneath the abstract UI description. Within this model the communicative acts used by the application are represented. Right underneath the domain of discourse model with all the classes defined for the online shop is shown. These objects are used to fill the *content* field of the communicative acts spedified in the discourse model.

The following paragraphs describe the function of each screen and present the readily rendered screens as well as the corresponding part of the abstract UI description.

#### 4.2.1.1 Screen1

This panel is the first one to be shown to the user. Its purpose is to offer all the available product categories and send the user's selection to the EventHandler as soon as the choice has been made. As can be seen in Figure 4.4 the PANEL'S layout manager has been set to GRIDLAYOUT. The appearance of this WIDGET as child of the panel Screen1 implies the inclusion of GRIDLAYOUTDATA widgets for each WIDGET contained by this component. The three Name LABELS are used to display a given, static text. This text is taken from



```
◢ platform:/resource/OnlineShop.UserInterface/model/onlineshop_construct.abstractui
  ◢ Choice Anonymous
    ◢ Frame Window
      ◢ Choice Anonymous
        ◢ Panel Screen1
          ◢ Label Name
              Style bigHeading
              Grid Layout Data 0
          ▷ Label Name
          ▷ Label Name
          ◢ List Widget List13
              Grid Layout Data 2
            ▷ Button Anonymous
            ▷ Label Anonymous
            Grid Layout 0
        ▷ Panel Screen2
        ▷ Panel Screen3
        ▷ Panel Accept
        ▷ Panel Reject
▷ platform:/resource/OnlineShop/model/OnlineShop_inserted.discourse
▷ platform:/resource/OnlineShop/model/I-Content.ecore
```

**Figure 4.4:** Abstract UI Description for Screen1

the LABEL'S *text* field and stored in the messages.properties file during the generation of the system.

Due to the use of local variables no problems arise through the identical denomination of the three LABELS. As name clashes are almost inevitable for larger systems, the code generator attaches a simple number to every generated widget name to make them unique.

Each WIDGET directly contained by a panel with GRIDLAYOUT, needs to be followed by at least one child. The attribute values of the obligatory `GridBagConstraints` are defined in the corresponding abstract UI widget and transferred to source code by the generator. The STYLE widget that can be seen as an attachement of the first LABEL, carries optional information. In case no style information is found, no style information is set and Java Swing uses its default values.

Figure 4.5 shows the generated screen during runtime, thus filled with information. The three Name LABELS can be seen in the upper part of this screen. The list of product categories seen below corresponds to the LISTWIDGET List13 in Figure fig:Screen1.

The ListWidget is intended to display the list of product categories. Each entry in this list consists of a Button, which can be clicked to chose the product category and a Label that carries additional information. The third child of the ListWidget is the obligatory layout information. Widgets contained by a ListWidget do not require any layout data, because they are contained by a panel whose layout manager is set to `FlowLayout`. The only way to influences the arrangement of the widgets is their order in the abstract UI description. The widgets are added onte after another to the panel of the list entry. This is why the sequence found in the abstract UI specification implies the order in the actual screen.

As the content of the ListWidget is only filled during runtime, the number of entries is adjusted to the number of data sets received by the communicative act specified in each Widget's *tracesTo* field.
The last widget contained by Screen1 is the layout widget that specifies the GridLayout.

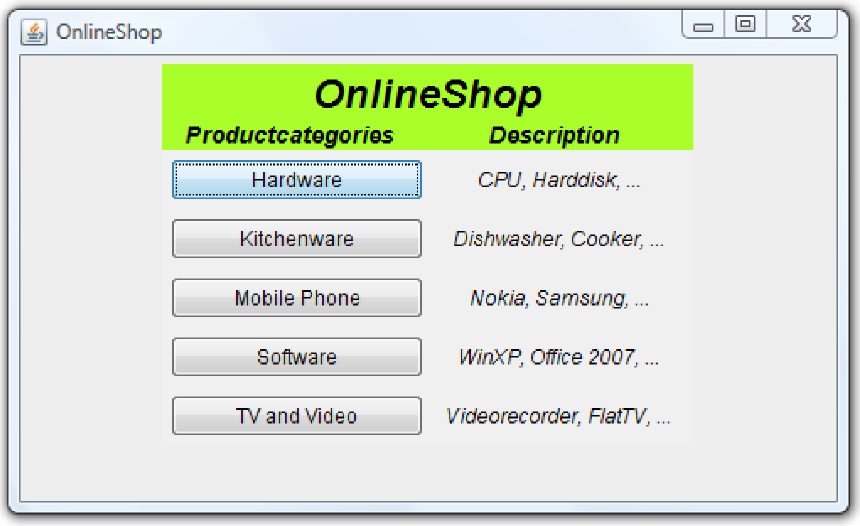
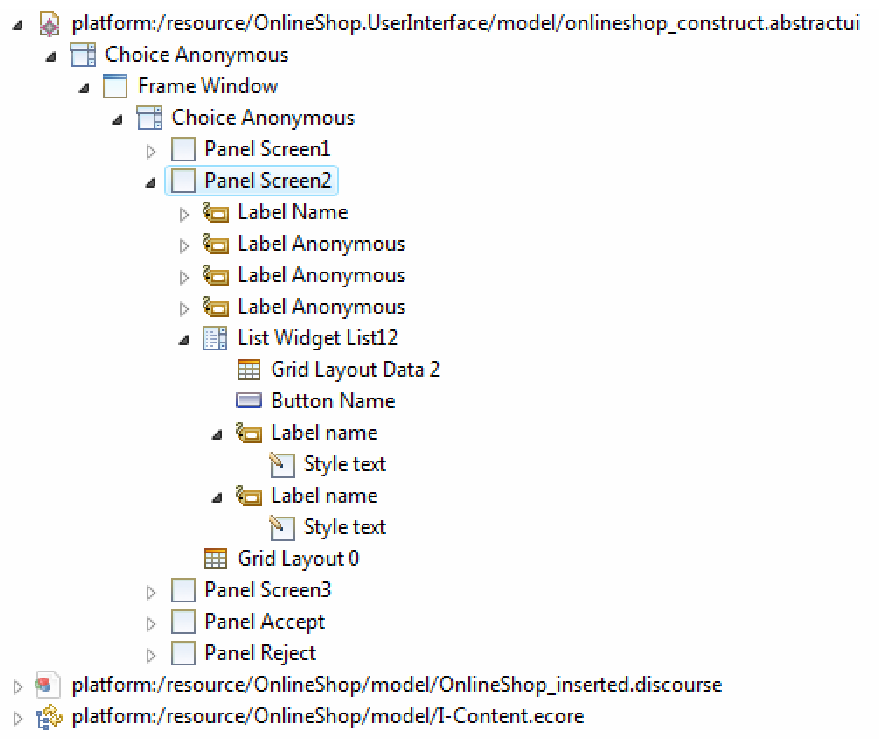


**Figure 4.5:** Online Shop Screen1

### 4.2.1.2 Screen2

As soon as the user selects a product category by clicking the corresponding button, a communicative act is sent to the application logic. The communicative act received in return triggers a state change in the state machine, which results in Screen2.

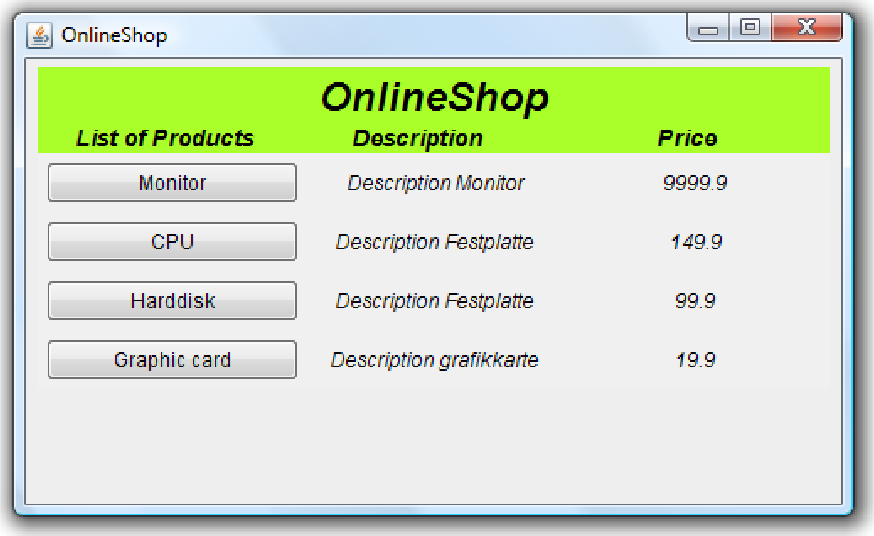
Screen2 is another panel that offers the user a list to select from. Therefore its structure is very similar to Screen1 as can be seen in Figure 4.6. The ListWidget offers the user the items belonging to the selected product category. The Button selects the product and the Labels carry additional information, in this case a description of the product and its price. As the content of the list is filled during runtime and consists of entities of the sample entry found in the abstract UI specification, the Style widgets attached to the two Labels are applied to each entry.



```
platform:/resource/OnlineShop.UserInterface/model/onlineshop_construct.abstractui
  Choice Anonymous
    Frame Window
      Choice Anonymous
        Panel Screen1
        Panel Screen2
          Label Name
          Label Anonymous
          Label Anonymous
          Label Anonymous
          List Widget List12
            Grid Layout Data 2
            Button Name
            Label name
              Style text
            Label name
              Style text
          Grid Layout 0
        Panel Screen3
        Panel Accept
        Panel Reject
platform:/resource/OnlineShop/model/OnlineShop_inserted.discourse
platform:/resource/OnlineShop/model/I-Content.ecore
```

**Figure 4.6:** Abstract UI Description for Screen2

Figure 4.7 shows the readily rendered and already filled Screen2 that is displayed if Hardware has been selected as the product category. The three LABELS named Anonymous represent the header of each list entry. Due to the calculated preferred size the layout resembles the form of a table.

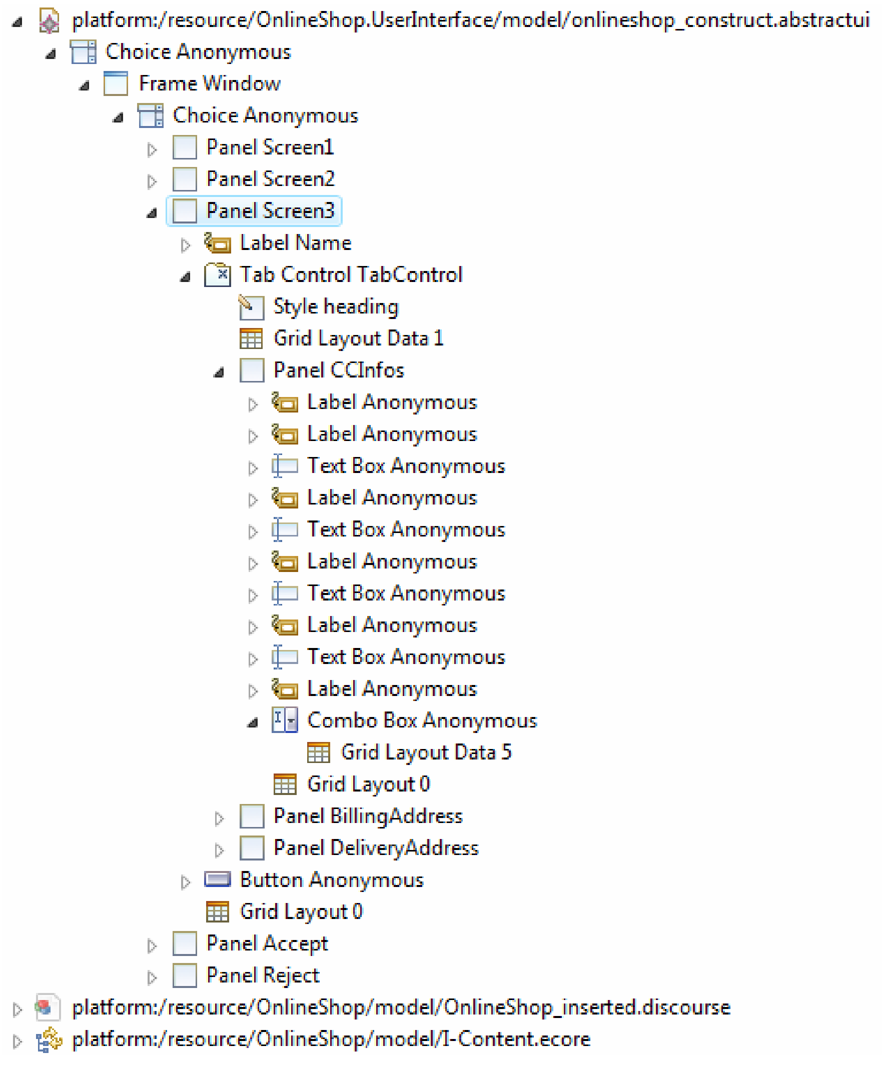


| List of Products | Description | Price |
|---|---|---|
| Monitor | Description Monitor | 9999.9 |
| CPU | Description Festplatte | 149.9 |
| Harddisk | Description Festplatte | 99.9 |
| Graphic card | Description grafikkarte | 19.9 |

**Figure 4.7:** Online Shop Screen2

#### 4.2.1.3 Screen3

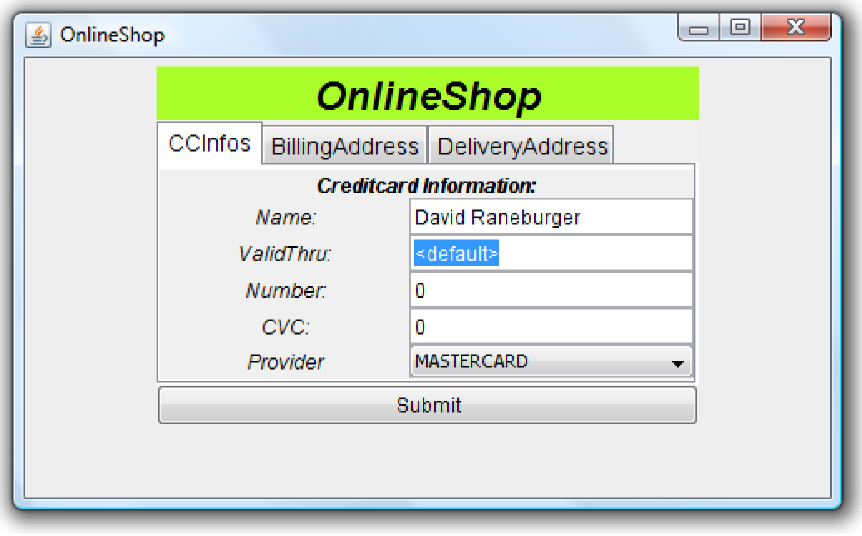
To select an item, the user only needs to click the corresponding button. Just like in Screen1, a communicative act with the content set according to which button has been clicked, is sent to the application logic. Subsequently the state machine triggers another transition as the following communicative act is received by the EventHandler. As a result, Screen3 is displayed. The difference between this screen and its two predecessors is that data is collected instead of only displayed. As a consequence, the screen's abstract UI description, shown in Figure 4.8, is quite different compared to Screen1 and Screen2.



```
platform:/resource/OnlineShop.UserInterface/model/onlineshop_construct.abstractui
  Choice Anonymous
    Frame Window
      Choice Anonymous
        Panel Screen1
        Panel Screen2
        Panel Screen3
          Label Name
          Tab Control TabControl
            Style heading
            Grid Layout Data 1
            Panel CCInfos
              Label Anonymous
              Label Anonymous
              Text Box Anonymous
              Label Anonymous
              Text Box Anonymous
              Label Anonymous
              Text Box Anonymous
              Label Anonymous
              Text Box Anonymous
              Label Anonymous
              Combo Box Anonymous
                Grid Layout Data 5
              Grid Layout 0
            Panel BillingAddress
            Panel DeliveryAddress
          Button Anonymous
          Grid Layout 0
        Panel Accept
        Panel Reject
platform:/resource/OnlineShop/model/OnlineShop_inserted.discourse
platform:/resource/OnlineShop/model/I-Content.ecore
```

**Figure 4.8:** Abstract UI Description for Screen3

Only four Widgets are directly contained by this screen: a Label, a TabControl, a Button and the GridLayout. The Label can easily be identified as the header seen in Figure 4.9. The task of collecting the user data is fulfilled by the TabControl. The abstract UI shows that this widget contains three Panels apart from the obligatory layout information. Just like in the other Panels the additional style information is not mandatory.



Creditcard Information:

Name: David Raneburger

ValidThru: <default>

Number: 0

CVC: 0

Provider MASTERCARD

**Figure 4.9:** Online Shop Screen3

The first Panel is used to collect the user's payment data. Figurefig:Screen3 shows that each TextBox is preceded by a Label. These Labels are found to the left of each text box in Figure fig:OnlineShopScreen3 and inform the user about which data to insert. Each panel corresponds to a communicative act, which implies that each text box sets the attribute of the communicative act's content object.

The CCInfos Panel collects the information needed for payment. The corresponding object is defined in the application's domain of discourse model and is named CreditCard (see fig. 4.3). Whenever the text box loses the focus, its data is transferred to the attribute field of this object. The combo box is filled with the values specified as the enumeration CCProvider in the domain of discourse model as well. Unlike the ListWidget's layout, the layout of these Panels needs to be specified explicitly to receive a consistent design.

The panels BillingAdress and DeliveryAddress contain only labels and text fields as their information is specific for each user. The corresponding content objects are specified in the domain of discourse model and named Address.

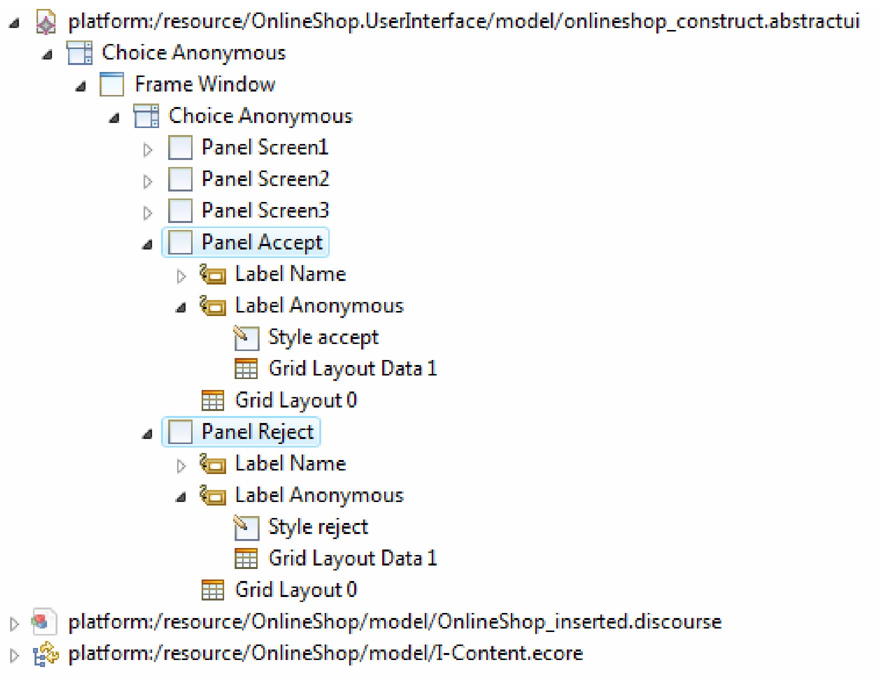
The TabControl allows the user to switch between these panels within the same screen. The information is not sent when he changes the tab, but only when he clicks the submit Button situated on the same level as the TabControl. Unlike the Buttons presented so far not only one, but three different communicative acts are sent. Each of these communicative acts contains one of the objects filled by the panels (i.e. credit card details, payment and shipment address).

This Button already represents some kind of optimization, because compared to one Button per Panel and communicative act it saves the user two mouse clicks and more importantly two rounds of communication with the application logic.

#### 4.2.1.4 Accept & Reject

One of these two panels concludes every online shop discourse. Both structures are similar and very simple as can be seen in Figure 4.10. Their only task is to inform the user whether the transaction has been finished successfully or not. Which screen is displayed depends on the data sent to the application logic by Screen3. Both of them are, as they are children of the FRAME Window, displayed in the application window and not pop-ups as the smaller size of the screens in Figure fig:OnlineShopAcceptReject may suggest.

If every text field has been filled out with the expected type of data the Accept panel confirms the successful end of the transaction. In case any data has been filled out incorrectly or left blank, the application logic returns a communicative act to inform the user about the failure of the transaction. The content of this communicative act is displayed to the user.

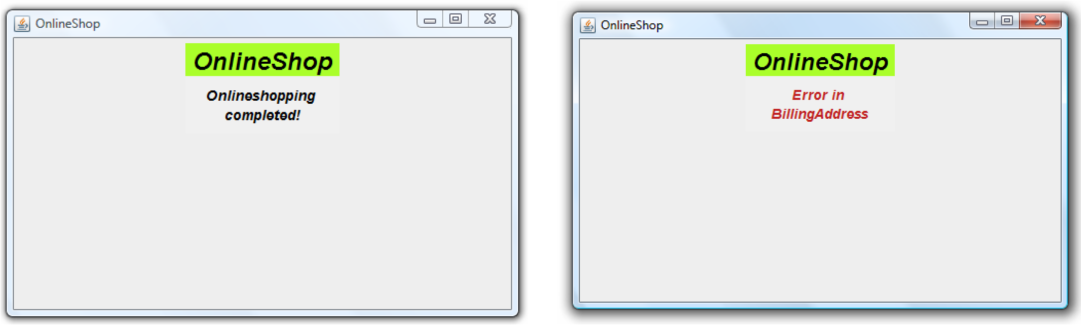


```
platform:/resource/OnlineShop.UserInterface/model/onlineshop_construct.abstractui
  Choice Anonymous
    Frame Window
      Choice Anonymous
        Panel Screen1
        Panel Screen2
        Panel Screen3
        Panel Accept
          Label Name
          Label Anonymous
            Style accept
            Grid Layout Data 1
          Grid Layout 0
        Panel Reject
          Label Name
          Label Anonymous
            Style reject
            Grid Layout Data 1
          Grid Layout 0
platform:/resource/OnlineShop/model/OnlineShop_inserted.discourse
platform:/resource/OnlineShop/model/I-Content.ecore
```

**Figure 4.10:** Abstract UI Description for Screen Accept & Reject

Style definitions are given for both LABELS in the online shop's style sheet. The corresponding style class is displayed next to the STYLE widget in the abstract UI.

Figure 4.10 shows two almost identical screens. Therefore it seems quite suggestive and easy to combine them and to set the text on the label during runtime.



OnlineShop

OnlineShop

Onlineshopping completed!

OnlineShop

OnlineShop

Error in BillingAddress

**Figure 4.11:** Online Shop Screen Accept & Reject

The problem is that this change induces the introduction of a new communicative act and, therefore a change in the discourse model. Due to the general character of these rules the resulting abstract UI is not 100 percent optimized for a certain application. There is no remarkable difference for the user, so this small overhead is not too much of a price to pay in comparison to the work saved by generating the system automatically.

### 4.2.2 Manually-Created vs. Automatically-Generated Abstract UI Description

General approaches always lead to some overhead as compared to hand-tailored solutions. In case of the abstract UI specification this overhead results in the generation of some widgets that are not necessarily required to fulfil the essential task of the system. This results in an abstract UI structure that may be harder to comprehend for humans but is equivalent to the optimized version, as far as the code generator is concerned.

A great part of optimization regarding the usability of the system needs to be done on the abstract UI model. Such things would be the integration of several panels into one tab control or the combination of different buttons to one. This is only possible if the data of all those buttons is needed to fullfil one goal (e.g. the online shop's payment and shipment details). As the statemachine is generated, based on the abstract UI specification as well, a functioning system is obtained in both cases.

The transformation from the discourse model to the abstract UI description is completed using rules. To optimize the usability of the screens, the number of rules will have to be augmented. The more rules, the more typical discourse sequences can be covered directly, hence the easier will be the generation of optimized code for several discourse constellations. This signifies that the more work is invested by the system developers, the easier will be the use of this technology for the system designer.

As mentioned above, this optimization does not have any direct implications for the code generator. Fine-tuning, like specifying the distance between neigbouring widgets, can either be done through the layout manager defined in the abstract UI specification, or via "default" values defined in the code generator.

# Chapter 5

# Conclusion

A major objective of the OntoUCP project was the fully automatic generation of a user interface, from a specification of the communication between user and system. The transformation process consists of two steps with an abstract UI model as an intermediate result. Device constraints are considered during the generation of the abstract UI description as well and need not be considered during the execution of the second step.

The code generator translates this abstract specification into toolkit-specific source code. Using EMF, the abstract user interface model is based on an Ecore meta-model. Moreover, the domain of discourse model, which is needed as input for the generator as well, is provided as an Ecore description, too. As the input is provided in the form of models, the generator has been implemented as a template engine. Due to the extensive use of Ecore models, and the fact that the whole system is based on Eclipse, JET was chosen as template language.

Java Swing was selected as the target toolkit, because of its platform independence and the wide deployment of Java, on all kinds of devices. The design can be adjusted to the operating system the application is running on, by setting the appropriate look and feel.

Another target toolkit option would have been Eclipse's Standard Widget Toolkit (SWT). The advantage of the SWT, compared to Java Swing, lies in the direct use of the operating system's graphic elements, thus a better performance can be achieved. The drawback is that many features are taken for granted, which are frequently not provided by non-Windows platforms. In this case those widgets need to be emulated, and this invalidates the performance benefit.

As a third option, HTML output has been considered to provide better support to browser-based applications. This option would have required to focus much on browser compatibility and runtime communication problems instead of the code generation process. Therefore, this option has not been implemented so far.

The design of the windows used by the application, has successfully been separated from all functional units by introducing support for CSS. This gives the system designer the option to influence the appearance of the windows. Furthermore, the style can be changed without a new generation of the abstract UI description. All that has to be adapted is the style sheet, which is handed to the generator. No problems arise if no, or incomplete information is added, because the generator, along with Java Swing, defines default values for any widget.

Another feature introduced by the code generator, supports the use of multiple languages. Any text displayed to the user is stored in a properties file. To change this text, all that needs to be done is to modify or replace the properties file. Additionally to the files needed to display the windows of an application, a runtime component is generated. This component represents the view module and completes the MVC implementation.

Apart from the components created for the application, the Preview class is generated. This class can be executed as a Java application and enables the user to display the created windows independently from the other components of the application.

Concurrently the technology developed in the OntoUCP project, is being adapted and extended to satisfy the requirements of the CommRob project. Throughout this project, a prototype robot is being developed, which will have the form of a robot trolley. This trolley knows several scenarious of interaction with its user. These scenarious have been specified, using the CDL presented in 3.1. What still needs to be found is a way to link these discourse descriptions to be able to render them to a consistent user interface. Using the OntoUCP components, all discourses can be rendered without problems. As they are processed separately, the combining component still needs to be created manually. One first step towards full automation will be the creation of this component by using abstract UI widgets. The abstract UI descriptions from every discourse can be generated automatically and, therefore only needs to be added to the file. Once a consistent abstract UI representation is available the code generator can be started. The second possibility is to take the automatically generated source code from each discourse and include it in the manually written main frame.

The code generator has been implemented and tested successfully as part of the OntoUCP framework. It is applicable to any kind of abstract UI specification generated automatically from a discourse model. Due to its modular character, the support for further abstract UI widgets or target-toolkits can be included without much effort.

# Appendix A

# Diagrams

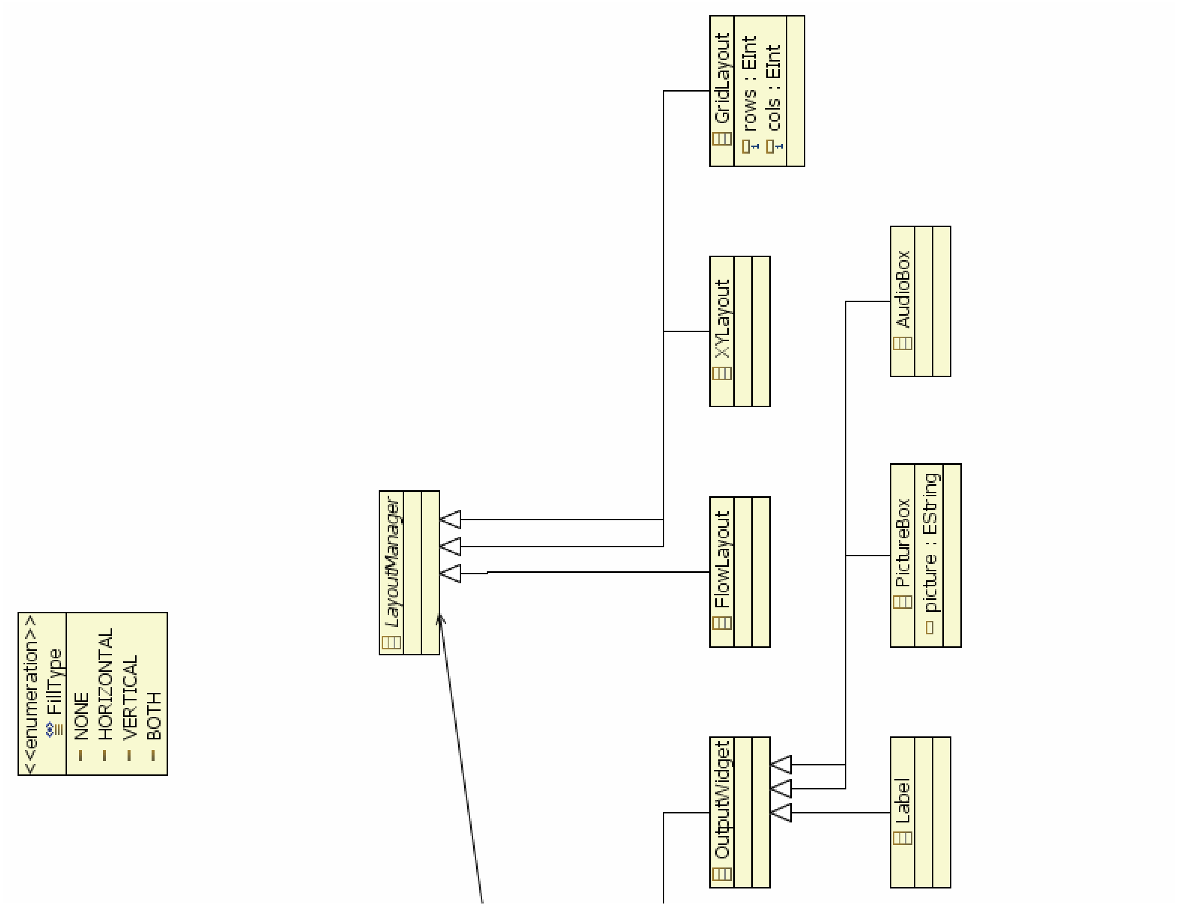


**Figure A.1:** The Abstract UI Meta Model Part I

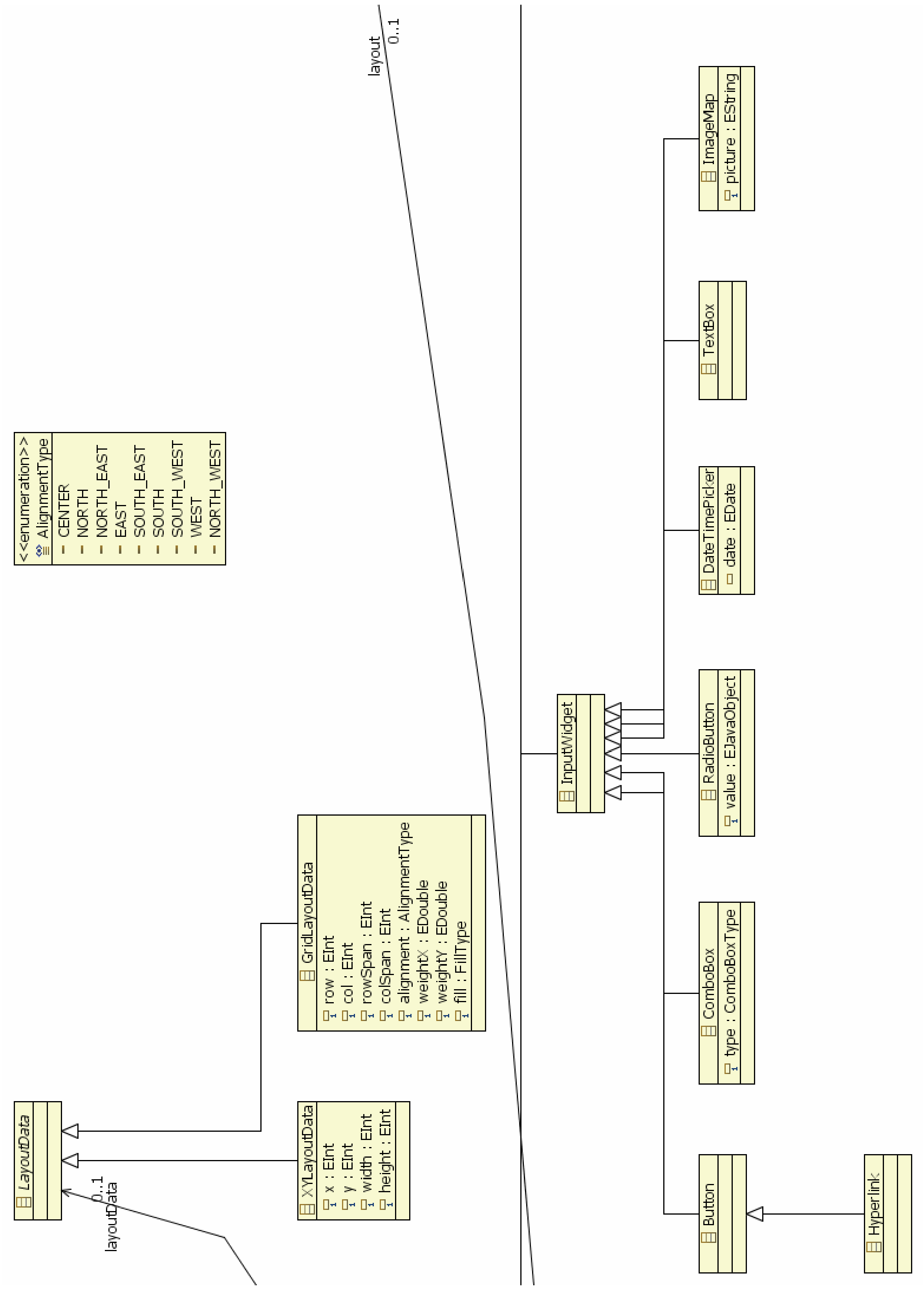


**Figure A.2:** The Abstract UI Meta Model Part II

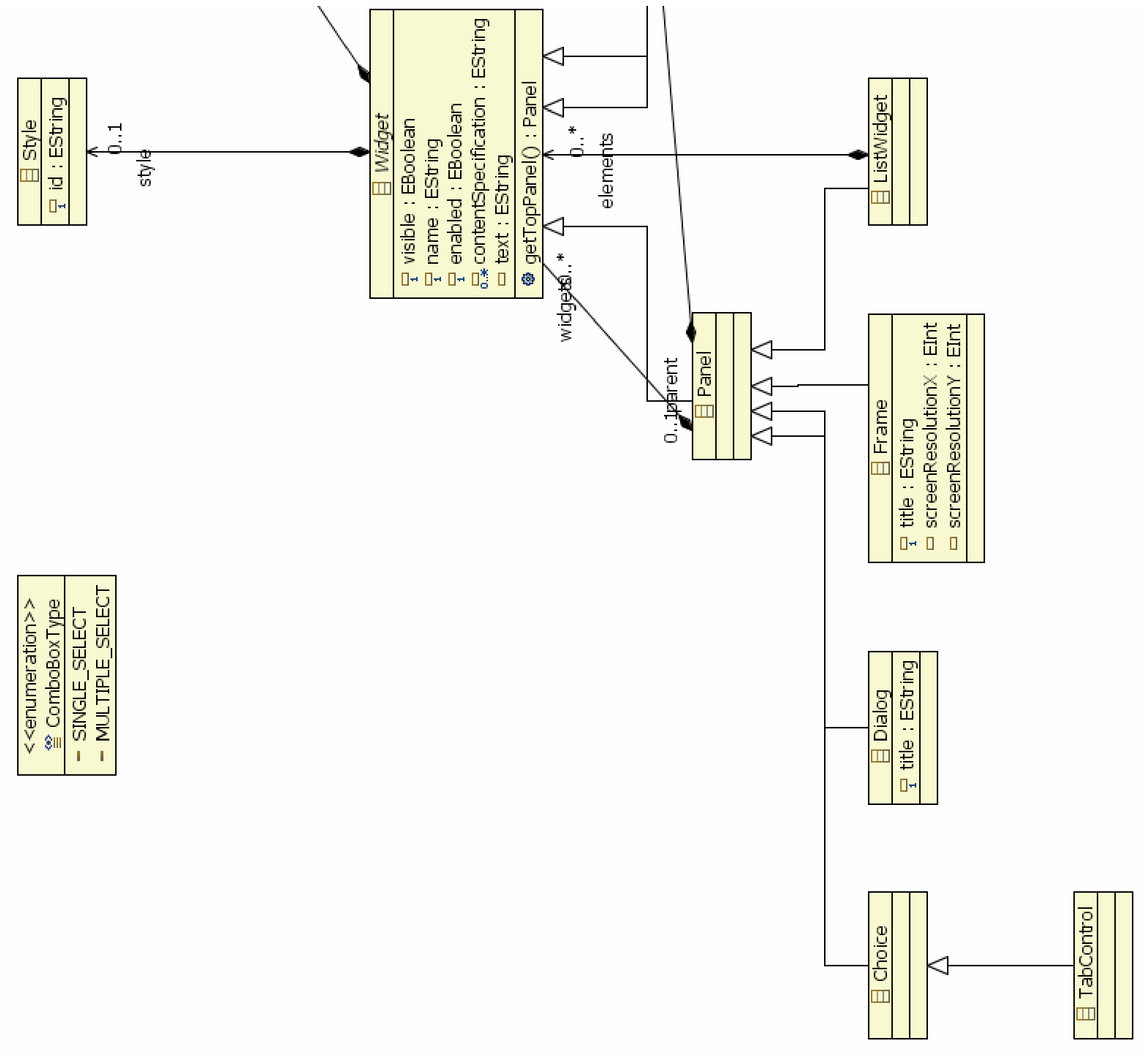


**Figure A.3:** The Abstract UI Meta Model Part III

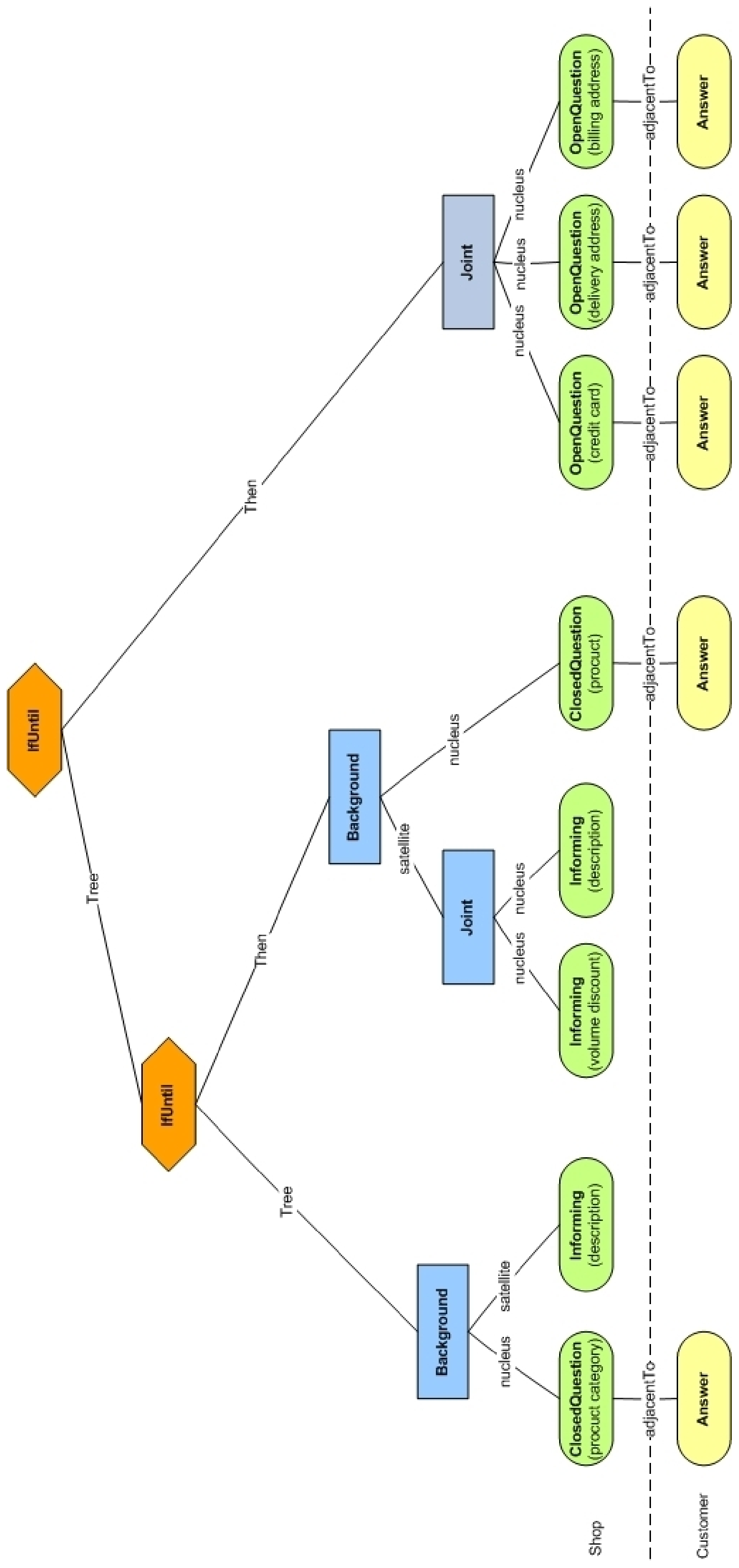


**Figure A.4:** The Online Shop Discourse Model

# List of Figures

# Bibliography

[ABF+06a] Arnautovic, Edin ; Bogdan, Cristian ; Falb, Juergen ; Horacek, Helmut ; Kaindl, Hermann ; Kavaldjian, Sevan ; Popp, Roman ; Roeck, Thomas: Complete Specification of the Communication Description Language (CDL). (2006) 26

[ABF+06b] Arnautovic, Edin ; Bogdan, Cristian ; Falb, Juergen ; Horacek, Helmut ; Kaindl, Hermann ; Kavaldjian, Sevan ; Popp, Roman ; Roeck, Thomas: Language Specification of Episodes. (2006) 21, 23

[ABF+06c] Arnautovic, Edin ; Bogdan, Cristian ; Falb, Juergen ; Kaindl, Hermann ; Kavaldjian, Sevan ; Popp, Roman ; Szep, Alexander: User Interface Rendering Specification (Structural Part). (2006) 56

[ABF+07a] Arnautovic, Edin ; Bogdan, Cristian ; Falb, Juergen ; Kaindl, Hermann ; Kavaldjian, Sevan ; Popp, Roman ; Szep, Alexander: Information System Application Design Document. (2007) 33

[ABF+07b] Arnautovic, Edin ; Bogdan, Cristian ; Falb, Juergen ; Kaindl, Hermann ; Kavaldjian, Sevan ; Popp, Roman ; Szep, Alexander: User Interface Rendering Specification (IO Rendering). (2007) 27, 31, 56

[AFK+06] Arnautovic, Edin ; Falb, Juergen ; Kaindl, Hermann ; Kavaldjian, Sevan ; Popp, Roman ; Szep, Alexander: Communication Process Specification. (2006) 33

[Ame08] Amengual, Carlos. *CSS4J: A CSS API implementation for the Java platform.* http://informatica.info/. 2008 46

[BDRS97] Browne, Thomas ; Dávila, David ; Rugaber, Spencer ; Stirewalt, Kurt: Using declarative descriptions to model user interfaces with MASTERMIND. (1997) 22

[BFK+08] Bogdan, Cristian ; Falb, Jürgen ; Kaindl, Hermann ; Kavaldjian, Sevan ; Popp, Roman ; Horacek, Helmut ; Arnautovic, Edin ; Szep, Alexander: Generating an Abstract User Interface from a Discourse Model Inspired by Human Communication. In: *Proceedings of the 41th Annual Hawaii International Conference on System Sciences (HICSS-41).* Piscataway, NJ, USA : IEEE Computer Society Press, January 2008 21, 75

[CE00] Czarnecki, Krysztof ; Eisenecker, Ulrich: *Generative Programming.* Addison-Wesley, 2000 7

[FFMM94] FININ, Tim ; FRITZSON, Richard ; MCKAY, Don ; MCENTIRE, Robin: KQML as an Agent Communication Language. In: *Proc. of the third international conference on information and knowledge management* (1994) 21

[FKP$^+$09] FALB, Jürgen ; KAVALDJIAN, Sevan ; POPP, Roman ; RANEBURGER, David ; ARNAUTOVIC, Edin ; KAINDL, Hermann: Fully automatic User Interface Generation from Discourse Models. In: *Proceedings of the 2009 ACM International Conference on Intelligent User Interfaces (IUI 2009).* Sanibel Island, Florida, USA, to appear Feb 2009 33, 75

[Fou08a] FOUNDATION, The Apache S. *The Apache Velocity Project.* http://velocity.apache.org/. 2008 10

[Fou08b] FOUNDATION, The E. *AspectJ - crosscutting objects for better modularity.* http://www.eclipse.org/aspectj/. 2008 7

[Fou08c] FOUNDATION, The E. *The Eclipse Modeling Framework.* http://www.eclipse.org/modeling/emf/. 2008 17

[KFAP06] KAINDL, Hermann ; FALB, Juergen ; ARNAUTOVIC, Edin ; POPP, Roman: High-level Communication in Systems-of-Systems. In: *Proceedings of the Fourth Annual Conference on Systems Engineering Research (CSER-06), Los Angeles, California* (2006) 19, 75

[KM97] KIMBROUGH, Steven O. ; MOORE, Scott A.: On automated message processing in electronic commerce and work support systems. In: *ACM Transactions on Information Systems* (1997) 21

[LGF90] LUFF, Paul (Hrsg.) ; GILBERT, Nigel (Hrsg.) ; FROHLICH, David (Hrsg.): *Computers and Conversation.* Academic Press, London, UK, January 1990 1

[MT88] MANN, W.C. ; THOMPSON, S.A.: Rhetorical Structure Theory: Toward a functional theory of text organization. In: *Text* (1988) 1, 22

[PFA$^+$09] POPP, Roman ; FALB, Jüurgen ; ARNAUTOVIC, Edin ; KAINDL, Hermann ; KAVALDJIAN, Sevan ; ERTL, Dominik ; HORACEK, Helmut ; BOGDAN, Cristian: Automatic Generation of the Behavior of a User Interface from a High-level Discourse Model. In: *Proceedings of the 42nd Annual Hawaii International Conference on System Sciences (HICSS-42).* Piscataway, NJ, USA : IEEE Computer Society Press, to appear 2009 26, 27, 33, 75

[Pop07a] POPMA, Remko. *JET Tutorial Part 1.* http://www.eclipse.org/articles/Article-JET/jet_tutorial1.html. 2007 15

[Pop07b] POPMA, Remko. *JET Tutorial Part 2.* http://www.eclipse.org/articles/Article-JET2/jet_tutorial2.html. 2007 16

[Pro08] PROJECT, The F. *FreeMarker: Java Template Engine Library.* http://freemarker.org/. 2008 12

[Sea69] SEARLE, J. R.: *Speech Acts: An Essay in the Philosophy of Language.* Cambridge, England : Cambridge University Press, 1969 1, 20

[SV05] Stahl, Thomas ; Voelter, Markus: *Modellgetriebene Softwareentwicklung.* dpunkt.verlag, 2005 4

[Tea05] Team, XDoclet. *XDoclet - Attribute-Oriented Programming.* http://xdoclet.sourceforge.net. 2000 - 2005 7

[Tea08] Team, The AndroMDA C. *AndroMDA.* http://www.andromda.org/. 2008 13, 75

[WHA00] Wilson, Chris ; Hégaret, Philippe L. ; Apparao, Vidur. *Document Object Model CSS.* http://www.w3.org/TR/2000/REC-DOM-Level-2-Style-20001113/css.html. 2000 46