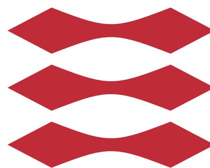
Seamlessness integration between textual informal BON and Java

Stephen Kow Sarquah
s070069



Kongens Lyngby 2013
M.Sc.-2013-83

DTU Compute
Technical University of Denmark
Matematiktorvet, Building 303B
DK-2800 Lyngby
Denmark
Phone +45 4525 3031, Fax +45 4588 1399
compute@compute.dtu.dk
www.compute.dtu.dk/ M.Sc.-2013-83

Summary (English)

BON (Business Object Notation) is a method well suited for analysis and design of object-oriented systems. BON focuses on seamlessness, reversibility, scalability, simplicity and software contracts to achieve the goal of narrowing the gap between analysis, design and implementation by using the same notation and semantics in the three phases. BON has two kinds of notations, graphical and textual BON. Textual BON is divided into formal diagrams and informal charts to describe static and dynamic modeling of object-oriented software. Textual informal BON is written in structured English. An example of a query in a class chart would be *"What is its name?"*

Textual informal BON is possible to write by hand but at this point there exists no tool which integrates textual informal BON with Java in such a way that the textual informal BON generates Java source code and the Java source code generates textual informal BON in such a way that feels seamless for the user of the tool.

This master thesis documents the design and implementation of a textual informal BON editor called iBONText, which has a code generator that features seamless integration from textual informal BON to Java. iBONText is able to reverse the process and generate textual informal BON from Java. The code generation from BON to Java creates Javadoc custom annotations and follows Java identifier naming conventions. When the textual informal BON updates the Java source code updates correspondingly. This gives traceability from the analysis to the implementation, making the software development more efficient.

Summary (Danish)

BON (Business Object Notation) er en metode velegnet til analyse og design af objekt-orienterede systemer. BON fokuserer på gnidningsfrit, reversibilitet, skalerbarhed, enkelhed og software kontrakter for at nå målet om at indsnævre kløften mellem analyse, design og implementering ved at bruge samme notation og semantik i de tre førnævnte faser. BON har to slags notationer, grafisk og tekstuel BON. Tekstuel BON er opdelt i formelle diagrammer og uformelle diagrammer til at beskrive statisk og dynamisk modellering af objekt-orienteret software. Tekstuel uformel BON er skrevet i struktureret engelsk. Et eksempel på en forespørgsel i et klasse diagram ville være *"What is its name?"*

Tekstuel uformel BON er muligt at skrive i hånden, men på dette tidspunkt eksisterer der intet værktøj, som integrerer tekstuel uformel BON med Java på en måde, at tekstuel uformel BON genererer Java kildekode og Java kildekode genererer tekstuel uformel BON på en sådan måde, der føles gnidningsfrit for brugeren af værktøjet.

Dette speciale dokumenterer designet og implementeringen af en tekstuel uformel BON editor, der hedder iBONText, som har en kodegenerator, der indeholder gnidningsfrit integration fra tekstuel uformel BON til Java. iBONText er i stand til at vende processen og generere tekstuel uformel BON fra Java. Koden generation fra BON til Java genererer Javadoc brugerdefinerede annotations og følger Java identifikator navngivning. Når tekstuel uformel BON opdaterer, opdateres Java kildekoden tilsvarende. Dette giver sporbarhed fra kravene til implementeringen, hvilket gør software udviklingen mere effektiv.

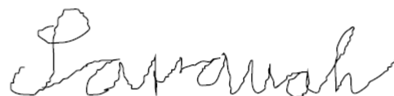
Preface

This thesis was prepared at the department of DTU Compute at the Technical University of Denmark in fulfilment of the requirements for acquiring an M.Sc. in Computer Science and Engineering. It was written with professor Joseph Kiniry as supervisor.

The thesis deals with a domain specific language (DSL), model driven design and codegeneration from abstract models. It deals with maintaining traceability from analysis to implementation.

The main focus in this thesis is on modeling textual informal Business Object Notation (BON) as a DSL, codegeneration from informal BON to Java and from Java to informal BON.

Lyngby, 05-August-2013



Stephen Kow Sarquah

Acknowledgements

I would like to thank my supervisor professor Joseph Kiniry for inspiring me to make iBONText and for his support. His previous work with BON has been a great inspiration. I would like to thank my family for supporting me during the writing process. I would like to thank my sister Maria for her suggestions, to be able to take time out of her schedule and for her support especially in the end of the writing process.

Contents

Summary (English) i

Summary (Danish) iii

Preface v

Acknowledgements vii

1 Introduction 1
- 1.1 Problem definition . . . 1
- 1.2 Why do we need such a tool? . . . 2
- 1.3 Related work . . . 3
- 1.4 Outline . . . 3

2 Background 5
- 2.1 Business Object Notation . . . 5
- 2.2 Informal BON . . . 7

3 Frameworks 13
- 3.1 Model driven design . . . 13
- 3.2 Eclipse Modeling Framework . . . 14
- 3.3 Domain Specific Languages . . . 14
- 3.4 Model to text transformation . . . 15
- 3.5 Eclipse JDT . . . 18
- 3.6 Javadoc . . . 19

4 Textual Informal BON Editor 21
- 4.1 Abstract Syntax of Informal BON . . . 21
- 4.2 Concrete Syntax of informal BON . . . 24

4.3 Test 25

5 Codegeneration from BON to Java 29
5.1 Modifications to the Ecore model 29
5.2 Informal BON to Java transformation 29
5.3 Configuration of Javadoc 32
5.4 Acceptance testing 35

6 Codegeneration from Java to BON 39
6.1 Initiation 39
6.2 Abstract Syntax Tree 40
6.3 Acceptance testing 42

7 Future work 45
7.1 Scenario charts 45
7.2 Concrete syntax 45
7.3 Ecore model 46
7.4 Java to BON 46
7.5 Graphical notation 46

8 Conclusion 47

A JMerger rules 49

B Plugin.xml 51

C User guide 53

Bibliography 61

List of Figures

2.1 Example of a system chart 7
2.2 Example of a cluster chart 8
2.3 Example of a class chart 9
2.4 Example of an event chart 10
2.5 Example of a scenario chart 10
2.6 Example of a creation chart 11

3.1 Example of a concrete syntax in EMFText 16

4.1 The Ecore model for Informal BON 22
4.2 Syntax highlighting for a comment 24
4.3 Syntax highlighting for manifest_textblock 25
4.4 JUnit test results 27

5.1 Informal_chartsImpl constructor 30
5.2 Private method named findProject 30
5.3 Private method named setParents 31
5.4 Private method named createPackages 31
5.5 Merging code with JMerger 33
5.6 Javadoc of the generated Java packages 33
5.7 Javadoc of the generated Java classes 34
5.8 The generated Java code for an Informal BON model 38

6.1 Private method named analyzePackages 40
6.2 Private method named parse 40
6.3 ExtendVisitor class 42

C.1 Import command 54

C.2 Import wizard 55
C.3 Select archive file to import 56
C.4 Select Run Configuration 56
C.5 Runtime JRE 57
C.6 Run configurations 57
C.7 Select a wizard 58
C.8 Textual informal BON editor 58
C.9 Generate BON model 59

Listings

3.1 Example of JET template 16
4.1 Snippet of the CS of informal BON 24
5.1 Javadoc configuration 33
6.1 Indexing Javadoc custom tag 41
6.2 Event Javadoc custom tag 41

List of Tables

4.1 Acceptance tests for informal BON editor 26

5.1 Acceptance tests of codegeneration from BON to Java 37

6.1 Acceptance tests of codegeneration from Java to BON 44

CHAPTER 1

Introduction

BON (Business Object Notation) is a method well suited for analysis and design of object-oriented systems. BON focuses on seamlessness, reversibility, scalability, simplicity and software contracts to achieve the goal of narrowing the gap between analysis, design and implementation by using the same notation and semantics in the three above mentioned phases[22]. BON has two kinds of notations, graphical and textual BON. Textual BON is divided into formal diagrams and informal charts to describe static and dynamic modeling of object-oriented software. Textual informal BON is written in structured English. An example of a query in a class chart would be *"What is its name?"*

Textual informal BON is possible to write by hand but at this point there exists no tool which integrates textual informal BON with Java in such a way that the textual informal BON generates Java source code and the Java source code generates textual informal BON in such a way that feels seamless for the user of the tool.

1.1 Problem definition

The main goal of this master thesis is to design and implement a textual informal BON editor, which has a code generator that features seamless integration from

textual informal BON to Java. Therefore the focus in this thesis is on textual informal BON. The optional goal for this thesis is to reverse the process and generate textual informal BON from Java. The code generation from BON to Java will create Javadoc custom annotations and must follow Java identifier naming conventions. When the textual informal BON updates the Java source code updates correspondingly. The textual informal BON editor will be created using EMFText[9] which is a plugin to the integrated development environment (IDE) called Eclipse[6]. EMFText is used to define textual domain specific languages (DSL) described by an Ecore metamodel. The grammar of the DSL is specified using concrete syntax which is derived from the EBNF[11] syntax specification language.

The textual informal BON editor must feature syntax highlighting. EMFText will be used to create the textual informal BON editor which parses and type checks the input and writes out errors, if the syntax or type is not correct. If the parsing and type checking is correct the Java source code should be generated. Eclipse JDT[7] will be used to generate textual informal BON from the Java source code.

1.2 Why do we need such a tool?

It is often the case that artifacts from the analysis are not updated during the software product life cycle. This is often due to the tools for creating the artifacts are not integrated with the tools for implementing the system. Doing so tools with rich features such as syntax highlighting and type checking will ease the burden of writing textual informal BON. If the artifacts for the analysis is not updated when the implementation is updated, the specification of the system will not be consistent with the source code. This means there will be no traceability of the analysis in the implementation. It is much harder to understand the requirements and behavior of a system by reading the source code instead of reading the specification. When the system needs to be modified, which is often the case for successful systems[3], it would be more time consuming for understanding the system because one needs to reverse engineer the system. The source code is the most accurate representation of a system since source code is the one talking to the hardware and executing system actions. Not the specification[22].

If a tool could automatically update the specification based on the implementation and vice-versa, traceability in analysis and implementation would be preserved, meaning the time used to understand the system would be less compared to reverse engineer the system. This will make the software development more

efficient. This is why iBONText has been developed.

1.3 Related work

iBONText is inspired by tools used to create BON, which is listed in the following.

1.3.1 BONc

BONc is a parser and typechecker for BON. BONc can read one or more files and/or input from standard input in the BON textual format, parse the input and typecheck it. The parser also has a pretty-printer that formats and indents the code and displays color syntax and error highlighting.[13]

1.3.2 ESC/IBON

ESC/IBON is a tool for translating informal BON to formal BON. The tool focuses on the BON static model. ESC/IBON has an API used to access the Grammatical Framework code, which the tool is build upon.[21]

1.3.3 Beetlz

Beetlz is a consistency (refinement) checker for BON and JML-annotated Java. It takes source files and specifications as input and returns feedback on whether and where they are inconsistent. The tool is available in a command-line and in a Eclipse plugin version.[5]

1.4 Outline

The following will describe the structure of the thesis.

- **Chapter 1** gives an introduction to BON. The chapter presents the problem and explains the goal of this thesis

- **Chapter 2** describes the Business Object Notation with focus on the informal charts.
- **Chapter 3** describes the frameworks used in order to create iBONText.
- **Chapter 4** goes into detail with the implementation of the textual informal BON editor.
- **Chapter 5** describes the code generation from BON to Java and how to display informal BON using Javadoc.
- **Chapter 6** describes the code generation from Java to BON.
- **Chapter 7** gives an overview of improvements which could be added to the future development of iBONText.
- **Chapter 8** contains the conclusion of this thesis.
- **Appendix** contains the rules for JMerger, the definition of plugin.xml and a user guide for iBONText.

Chapter 2

Background

This chapter gives the reader a background information about Business Object Notation (BON) and the informal charts of BON will be described in detail.

2.1 Business Object Notation

Business Object Notation[22] (BON) was introduced in 1994 by Kim Waldén and Jean-Marc Nerson and started as an attempt to extend the concepts of the Eiffel language[20]. BON is a method for modeling analysis and design of object oriented systems. BON consists of concepts and notations which are based upon the principles: simplicity, seamlessness, reversibility, software contracting and scalability. The syntax and semantics are kept as simple as possible while containing the most important object oriented concepts.

Simplicity is one of the principles of BON which separates it from other modeling languages such as UML[19]. The BON notation tries to minimize the number of concepts. It should be easy for a user of BON to quickly master the notation. The notation summary for UML (version 1.3) is 161 pages, whereas the summary for BON is one page.[12]

Object-oriented analysis and design has the potential to turn the transition from analysis to design to implementation into a seamless process which has been the goal for software engineering for over 20 years. The same set of abstraction of the class can be used in the analysis-, design- and implementation phases. BON utilizes this by using the same set of notations in the three phases making the process seamless[22].

It is often the case that a system needs to be modified to feature new requirements. Ideally this would mean that the analysis would be modified first followed by the design and then the implementation would be modified accordingly. However, it is often the case that only the implementation gets modified to feature the new requirements. High level specifications is only a crude representation of a system and the implementation problems of a programming language are ignored. Implementation problems will have to be taken care of before a specification can be made executable. This means a new level of abstraction would be introduced. Refinements is often applied to the source code because specifications cannot address detailed decisions. If the abstract system should be consistent with the source code, changes to the source code must be reflected in the specification which is often too expensive to maintain. There is no telling if a specification is consistent with the source code because correct source code can run with an incorrect specification but not vice-versa. Using BON it is possible to have traceability from analysis to design, to implementation because the classes introduced in the analysis will also be present in the final system[22].

It is important to guarantee software correctness in a system. The approach BON uses is software contracting. The semantics of each class is defined by assertions that specifies the pre- and post conditions along with the class invariant. These semantics form a contract between the supplier (the class having operations) and all classes using its operations (the clients). If the contracts are violated, the system will make an exception.

The system examples found in textbooks on object oriented analysis and design are nearly always small. A notation in object oriented analysis and design must be able to scale up and still be useful for large systems. When the system reaches over 20-30 classes, the class concept isn't enough to describe the structure, the grouping of classes. BON uses the term clustering to describe the grouping of classes and a group of classes will be called a cluster. In implementation there is a need to gain detailed information about a cluster and an overall view of the structure. The BON notation uses nested clustering and element compression to achieve this. An example of element compression would be icons in a graphical user interface.[22]

2.2 Informal BON

Informal BON consists of the informal charts:

- System chart
- Cluster chart
- Class chart
- Event chart
- Scenario chart
- Object creation chart

BON is divided into the static and dynamic model. The static model shows the clusters of the high leveled classes and the relationship between them. It also shows the class operations, their signatures and the semantic specification. The dynamic model shows how the high leveled classes fulfill their specification by calling other operations in same or other classes.

2.2.1 The static model

The system chart contains a brief description of one or more clusters of the system. An example of a system chart can be seen in figure 2.1.

```
3  system_chart PERSONS
4  indexing
5      author: "Stephen Sarquah";
6      keywords: "prototype", "person";
7  explanation
8      "A person is a programmer using this tool"
9  part "1/1"
10 cluster ORGANIZATION
11 description
12     "This represents all persons"
13 end
14
```

Figure 2.1: Example of a system chart

A cluster chart contains a number of class charts or subclusters of the system. It can contain information called indexing which can be keywords, author, created date or whatever one specifies as relevant. The cluster chart can also contain an explanation if there needs to be added further information specific to the cluster chart. An example of a cluster chart can be seen in figure 2.2.

```
38 cluster_chart TEST
39 indexing
40     keywords: "cluster chart";
41 explanation
42     "This cluster represents the test level of the system"
43 part "1/1"
44 class PERSON
45 description
46     "A person is a normal guy"
47 cluster WORKGROUP
48 description
49     "A workgroup consists of persons"
50 cluster WORKGROUP2
51 description
52     "Another workgroup consists of persons"
53 end
```

Figure 2.2: Example of a cluster chart

The class chart can contain indexing and explanation. Class charts can contain queries which are information other classes can ask from this class. It can contain commands which are services other classes can ask from this class to provide. A class chart can contain constraints which are the rules a class must obey. A class can inherit from another class. An example of a class chart can be seen in figure 2.3.

2.2.2 The dynamic model

A system event is an event which causes the system to react to a change. System events can either be external or internal. An external event is triggered by the environment outside the system, where it has no control. An example of an incoming external event could be a user clicking on a mouse button. An internal event is triggered by the system as part of a reaction to an external event. Some internal events are outgoing, which means as part of their reaction, they send a response to the environment outside the system.

```
79 class_chart PERSON
80 indexing
81     in_cluster: "Workgroup";
82     created: "31-01-2013";
83     revised: "31-01-2013";
84     author: "Stephen Sarquah", "John Doe", "Jane Doe";
85 explanation
86     "A person is a normal guy which people derive from"
87 part "1/1"
88 query
89     "Name of person's mother",
90     "Sex of person",
91     "age of person",
92     "Skills",
93     "Color of skin"
94 command
95     "Change age"
96 constraint
97     "Person must have skills",
98     "Person has a mother"
99 end
```

Figure 2.3: Example of a class chart

The idea of the event chart is to capture a list of external events that triggers a principal type of system behavior. The reason why we list external events is because the interaction of objects, which causes the execution of the system, are external events[22]. These external events may trigger one or more internal events. If an external event has a corresponding internal event, there is no need to list both of them in the event chart. The event chart is split up to two event charts where one is the list of external events and the other the list of internal events. An event lists which classes an event involves. An example of an event chart can be seen in figure 2.4.

The starting point of making a scenario chart is to use the event charts to select a set of important scenarios which illustrates the behavior of the system. Each scenario has a name and a short description about the scenario. An example of a scenario chart can be seen in figure 2.5.

The creation of new objects may be the link between the static and dynamic models. At some point during execution of the system, objects may be created. Analyzing how objects are created may help to identify the semantics of operations of classes. The object creation chart is a list of classes which creates other objects. Only the high leveled classes are considered, not the low level objects.

```
129 event_chart PERSONS
130 incoming
131 indexing
132     author: "Stephen Sarquah";
133 explanation
134     "This event chart shows all incoming events in the system"
135 part "1/1"
136 event
137     "Make person a programmer"
138 involves
139     PERSON
140 event
141     "Turn programmer into tester"
142 involves
143     PERSON
144 end
```

Figure 2.4: Example of an event chart

```
163 scenario_chart PERSONS
164 indexing
165     author: "Stephen Sarquah";
166 explanation
167     "This scenario chart shows a possible scenario of a system"
168 part "1/1"
169 scenario
170     "Phone rings"
171 description
172     "When the phone rings a person answers"
173 scenario
174     "Removing programmer from system"
175 description
176     "A programmer is removed from the system"
177 end
```

Figure 2.5: Example of a scenario chart

It is a good idea to not list frequently reused library classes such as SET or TABLE. An example of a creation chart can be seen in figure 2.6.

```
179 creation_chart PERSONS
180 indexing
181     author: "Stephen Sarquah";
182 explanation
183     "This creation chart explains how objects are created "
184 part "1/1"
185 creator PERSON creates PROGRAMMER
186 creator PERSON creates TESTER
187 creator PROGRAMMER creates TESTER
188 end
```

Figure 2.6: Example of a creation chart

Chapter 3

Frameworks

In this chapter the frameworks used to implement iBonText will be described. It was decided to implement iBonText as a plug-in for Eclipse, which is a widely used IDE, because of the frameworks which will be described.

3.1 Model driven design

The frameworks needed to implement iBonText must feature a way to construct a metamodel for textual informal BON. They must feature a way to construct a grammar in order to write structured textual informal BON charts. The frameworks must feature a codegenerator to generate Java source code from textual informal BON charts. The frameworks must also feature a way to retrieve information for textual informal BON from Java source code.

To achieve these goals, the frameworks EMFText, Java Emitter Templates and Eclipse JDT were chosen for this purpose.

3.2 Eclipse Modeling Framework

EMF is a modeling framework and code generation facility for building tools and applications based on a data model. EMF is based on Eclipse. EMF provides tools and runtime support to produce java classes, adapter classes, which enables command-based editing of the model, from a model structured in XML metadata interchange (XMI)[23]. EMF includes a meta model for describing models and describing the runtime support for models, which is called Ecore. EMF is capable of generating code from models to build a tree editor. This is done by using the generator model. The generator model can reproduce code for the model but if the user specifies to not overwrite the custom code created, the genmodel will not overwrite the custom code. The generation facility uses Eclipse JDT[7]. In EMF notification observers are called adapters. These adapters follow the observer pattern which notifies an object, when an event occurs.[8]

3.3 Domain Specific Languages

A DSL is a type of a programming-, specification-, or modeling language used to express statements in a specific problem or domain. DSLs' targets a specific software problem in a specific domain, while a general purpose programming language like Java or C# or a general purpose modeling language like UML[19], tries to solve software problems in many domains. An example of a DSL is CSS[4].

The most important advantage of a DSL, is that it is less comprehensive than a general purpose language. The learning curve is smaller for a DSL versus a general purpose language which means a DSL will improve productivity and efficiency. In most cases DSLs are much more expressive in their domain than a general purpose language. The biggest disadvantage of a DSL is that it is harder to find examples of how to use the DSL.

3.3.1 EMFText

EMFText is a plugin to Eclipse for defining textual syntax for Ecore-based meta models. A developer can define his/her own domain specific language (DSL) using this tool. EMFText can generate tool support for the DSL and it generates an Eclipse editor with syntax highlighting and customizable colors

which can be used to load and save instances of a model of the DSL. EMFText will be used to define textual informal BON as a DSL.

EMFText comes with the parser ANTLR[1], for loading instances of the model and a printer to save instances of the model. EMFText generates code for the DSL which is fully customizable. The editor which is generated from EMFText provides many features that are known from e.g. the Eclipse Java Editor such as code completion, syntax highlighting and instant error reporting when you mistype a character in the editor. EMFText has support for ANT[2].

3.3.1.1 Concrete Syntax

When a metamodel is defined as an Ecore model the textual representation of the metamodel concepts must be defined consequently. This is called the concrete syntax (CS). EMFText provides a syntax generator that can automatically create a CS specification from the metamodel. The CS conforms to Human Usable Textual Notation (HUTN)[16]. To see an example of a CS[10] see figure 3.1. The CS for the textual informal BON editor will follow the BON textual grammar provided by Kim Waldén and Jean-Marc Nerson[22].

3.4 Model to text transformation

The main idea of model to text transformation (M2T)[15] is to use a template to generate textual artifacts. Textual artifacts can be code, documentation, reports etc. The template has fields which are data extracted from the model that makes the codegeneration of textual artifacts dynamic. The code generation from model to text can be done manually but this might introduce new errors, can be very complex and take a long time. This process can be automated using a code generator which often results in better quality of the code. The transformations often uses best practices for code and ensures consistency to a project. An implementation of a code generator which iBONText will use is Java Emitter Templates.

3.4.1 Java Emitter Templates

Java Emitter Templates (JET) is a code generation framework used by EMF. JET can generate code or documentation from an EMF model using a template.

```
SYNTAXDEF forms
FOR <http://www.emftext.org/language/forms>
START Form

OPTIONS {
        overrideBuilder = "false";
        additionalDependencies = "org.emftext.language.forms.generator";
}

TOKENS {
        DEFINE MULTIPLE $'multiple'|'MULTIPLE'$;
}

TOKENSTYLES {
        "TEXT" COLOR #da0000;
        "FORM" COLOR #000000, BOLD;
        "ITEM" COLOR #000000, BOLD;
        "CHOICE" COLOR #000000, BOLD;
        "ONLY" COLOR #da0000, BOLD;
        "IF" COLOR #da0000, BOLD;
        "DATE" COLOR #000000, BOLD;
        "FREETEXT" COLOR #000000, BOLD;
        "NUMBER" COLOR #000000, BOLD;
        "DECISION" COLOR #000000, BOLD;
        "GROUP" COLOR #000000, BOLD;
}

RULES {
        Form ::= "FORM" caption['"','"'] !1 groups*;
        Group ::= !0 "GROUP" name['"','"'] !0 items*;
        Item ::= "ITEM" text['"','"'] ( explanation['"','"'] )?
                                ("ONLY" "IF" dependentOf[])? ":" itemType !0;
        Choice ::= "CHOICE" (multiple[MULTIPLE])?
                                "(" options ("," options)* ")";
        Option ::= ( id[] ":")? text['"','"'];
        Date ::= "DATE";
        FreeText ::= "FREETEXT";
```

Figure 3.1: Example of a concrete syntax in EMFText

The syntax of JET is a subset of Java Server Pages for retrieving data from the model. Using JET syntax one can iterate through a model and retrieve data. An example of a JET template can be seen in listing 3.1.

```
1 <%@ jet package="hello" class="GreetingTemplate" %>
2 Hello, <%=argument%>!
```

Listing 3.1: Example of JET template

If we give the string "world" as argument the generated text will be Hello, world.

3.4.2 JMerge

JMerge[18] is an open source tool in EMF which allows code generators to merge generated code with user modified code. The user modified code will be merged with the generated code based on rules specified in XML. EMF has JMerge rules which merges Java elements containing a Javadoc tag named @generated. If a user modifies the generated code which has a @generated tag and regenerates code the user modified code will be overwritten. If a user changes the @generated tag, modifies the generated code and regenerate the code, the changes will not be overwritten. The process of merging user modified code with generated code is the following:

1. JMerge loads the rules specified in XML, the existing code and the generated code into an Abstract Syntax Tree(AST)

2. The JMerge rules must contain one or more dictionary patterns which specifies regular expressions that will be executed against elements in the AST. Dictionary patterns must include a capture which is an expression enclosed in parentheses. The regular expression for EMF's @generated tag is @(gen)erated. When a dictionary pattern matches a Java element JMerge records the capture and the Java element.

3. JMerge walks through the pull elements from the JMerge rules. Each pull element specifies the AST object to operate on and the AST attribute which should be copied from existing to generated code.

4. JMerge walks through the AST of the generated code to identify elements which are not in the AST of the existing code. When an element is found in the generated code and not in the existing code, the generated code is copied to the existing code.

5. JMerge walks through the sweep element from the JMerge rules. Each sweep element specifies the AST object to operate on and a markup. If the existing code matches a sweep element but the generated code does not, the Java element is removed from the existing code.

6. The updated existing code is written to a Java class file.

JMerge can merge rules which target:

- CompilationUnit
- Field
- Import
- Initializer
- Member
- Method
- Package
- Type

3.5 Eclipse JDT

JDT stands for Java Development Tools. It consists of a set of plug-ins to give the rich featured environment Java IDE to Eclipse. The plug-ins provides APIs to be accessed by external tools. The plug-ins are categorized into the following groups:

- JDT APT
- JDT Core
- JDT Debug
- JDT Text
- JDT UI

JDT APT adds support for annotations to Java 5 projects in Eclipse. JDT Core adds support for a Java model which provides an API for navigating the Java element tree. A Java element tree contains package fragments, compilation units, binary classes, types, methods and fields. JDT Core also adds support for an indexed based search infrastructure which is used for searching, type hierarchy and refactoring. JDT Debug adds support for debugging. JDT Text adds support for syntax and keywords highlighting. It adds support to show Javadoc for a specific Java element in a pop-up window. It adds support to automatically create and organize import declarations. JDT Text also adds support for code formatting. JDT UI adds support for the Java views of the Eclipse workbench. It features the package explorer, the type hierarchy view, Java outline view and a wizard for creating Java elements.[7]

3.6 Javadoc

Javadoc is a tool for parsing declarations and documentation comments in Java source code. It produces a set of HTML pages describing classes, interfaces, constructors, methods and fields. Using Javadoc it is possible to get a structured overview of a Java project and the comments in the Java source code.[17]

Chapter 4

Textual Informal BON Editor

In this chapter, the implementation of textual informal BON as a DSL will be described. The Ecore model for textual informal BON will be described followed by a description of a concrete syntax of the textual informal BON editor. Tests will be demonstrated to prove the system works according to the requirements.

4.1 Abstract Syntax of Informal BON

The Ecore model metamodels informal BON as the abstract syntax. It can be seen in figure 4.1. The following passage will describe the figure.

Informal BON consists of informal charts which is named Informal_charts. The class Informal_charts has a contained association with the multiplicity of zero or more, to Class_Chart, Cluster_chart, Event_chart, Scenario_chart and Creation_chart. Informal_charts has a contained association to System_chart with the multiplicity of one. System_chart inherits from the abstract class called NamedElement. NamedElement has an attribute called name which is of type EString. Since many classes have a name, they all inherit from NamedElement. System_chart is a chart and therefore it inherits from the abstract class

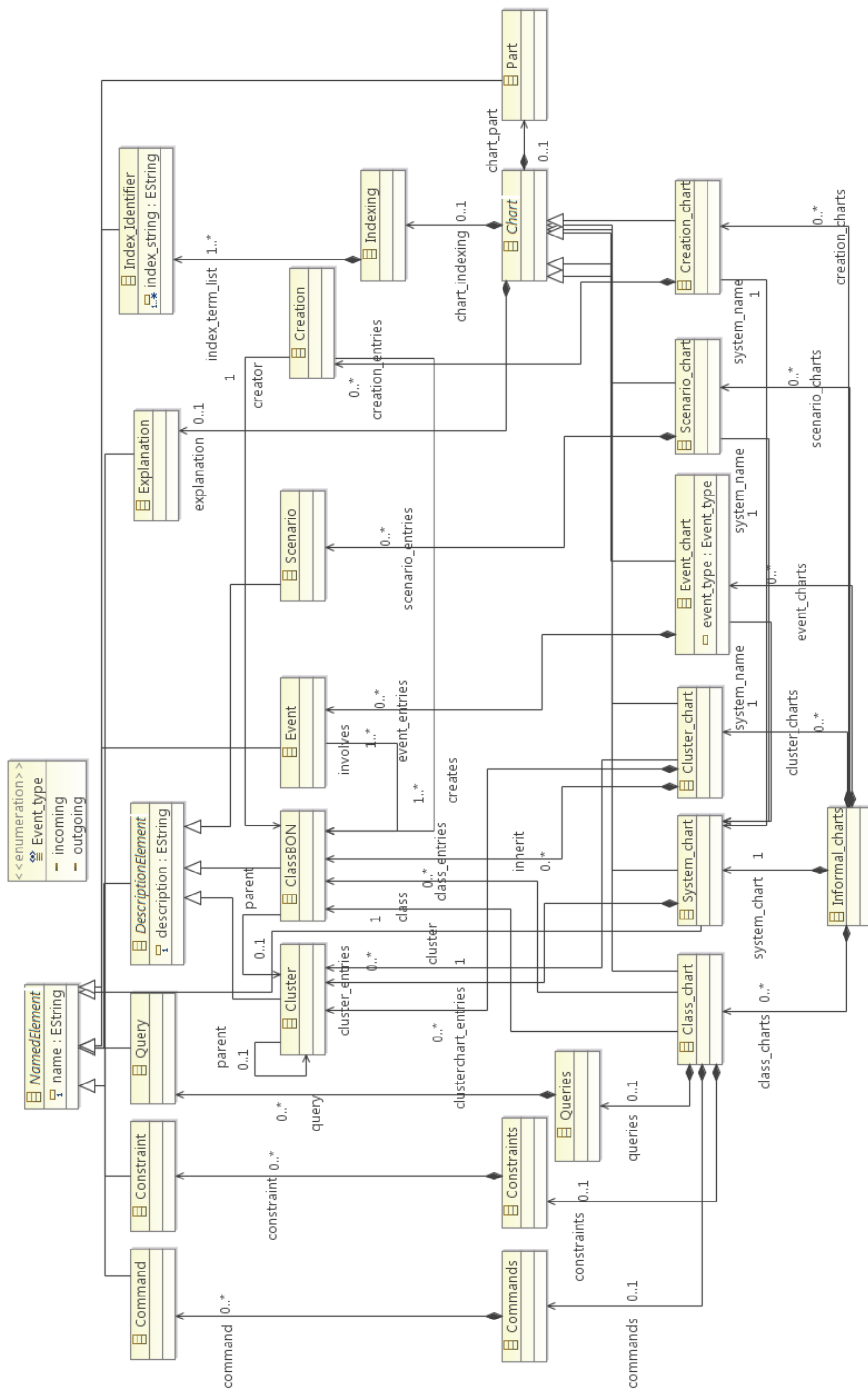


Figure 4.1: The Ecore model for Informal BON

Chart. A Chart has a contained association to Part called chart_part, with the multiplicity zero or one, which inherits from NamedElement. Part describes if and how the chart is divided into files; e.g. a System chart could be written in two files to simplify the system overview. Chart has a contained association to Explanation named chart_indexing and has the multiplicty zero or one. It inherits from NamedElement and it has a contained association to the class Indexing, which has a multiplicity zero or one named chart_indexing. Indexing has a contained association named index_term_list to the class Index_Identifier which inherits from NamedElement and has the attribute index_string, which is of the type EString. Index_string has the multiplicity of one or many because it should be possible to write several index_strings. An example could be an Index_Identifier which has the name keywords. Keywords consists of one or more index_strings. System_chart has a contained association to Cluster named cluster_entries, which has the multiplicity of one to many. This is because in a system chart we define the clusters. Cluster inherits from DescriptionElement which has the attribute called description of the type EString. The name states that it is a description of the cluster. DescriptionElement inherits from NamedElement so we can give Cluster a name.

As seen in figure 4.1, Cluster_chart inherits from Chart, has an association to Cluster and has a contained association to ClassBON named class_entries with the multiplicity of zero or many. The reason for the multiplicity of zero or many is because it is possible in BON to have a cluster in a cluster making it possible of nesting subsystems in a large system. ClassBON inherits from DescriptionElement.

Class_chart has one list for each of the classes Commands, Constraints and Queries. This is made to make several instances of Command, Constraint and Query inside one class which gives a better overview. The multiplicity of the associations from Commands to Command, Constraints to Constraint and Queries to Query are zero or many. Command, Constraint and Query inherits from NamedElement. A Class_chart has an association to ClassBON called class which has the multiplicity of one. This means there has to be instantiated a Class in order for a Class_chart to have an association to it. A Class_chart has an association, with the multiplicity of zero or more, to ClassBON which is called inherit. This means a Class_chart can inherit from one or more ClassBON. Class_chart inherits from Chart.

Event_chart has an attribute called event_type of the type Event_type.
Event_type is an enumeration specifying if the event chart, lists incoming or outgoing events. Event_chart has an association to System_chart called system_name because in informal BON one specifies the system name for the event chart. Event_chart inherits from Chart and has a contained association to Event named event_entries, which has the multiplicity of zero or many. This

lists all events for the event chart. Event has an association to ClassBON called involves, which has the multiplicity of one to many. As the name states an event can involve one or many classes. Event inherits from NamedElement.

Scenario_chart has an association to System_chart named system_name. It inherits from Chart and has a contained association to Scenario, which is named scenario_entries and has the multiplicity of zero or many. Scenario inherits from DescriptionElement. A scenario chart can have multiple scenarios which each has a name and a description.

Creation_chart inherits from Chart, has an association to System_chart named system_name and has a contained association to Creation named creation_entries with the multiplicity of zero or many. Creation has two associations to ClassBON. One named creator whichh the multiplicity of one, the other named creates, which has the multiplicity of one or many. In a creation chart a class, which is the creator, creates instances of one or more classes.

4.2 Concrete Syntax of informal BON

The concrete syntax (CS) defines what is allowed to be written in the informal BON editor. In the CS a comment is defined as seen in listing 4.1.

```
1  TOKENS {
2    DEFINE COMMENT $'--'(~('\n'|'\r'|'\uffff'))*$;
3    DEFINE MANIFEST_TEXTBLOCK STRING_BEGIN_END+$($+SIMPLE_STRING+$|'
        \\' $+LINEBREAK+WHITESPACE+$* '\\'($+WHITESPACE+$)*$
4    +SIMPLE_STRING+$ | $+WHITESPACE+$)+$+STRING_BEGIN_END;
5    DEFINE STRING_BEGIN_END $'"'$;
6  }
7  TOKENSTYLES {
8    "COMMENT" COLOR #00bb00, ITALIC;
9    "MANIFEST_TEXTBLOCK" COLOR #2a00ff;
10 }
```

Listing 4.1: Snippet of the CS of informal BON

This means whenever a line begins with the characters "–", the rest of the line will be viewed as a comment and have syntax highlighting. See figure 4.2.

```
1  -- This system chart is only for testing purposes
2  -- Informal charts
```

Figure 4.2: Syntax highlighting for a comment

A manifest_textblock is defined as string_begin_end, which is quote begin, simple_string, which is all characters without whitespace then string_begin_end, which is quote end. The manifest_textblock is highlighted in a blue color as seen in figure 4.3.

```
4 indexing
5     author: "Stephen Sarquah"
6     keywords: "prototype", "person"
```

Figure 4.3: Syntax highlighting for manifest_textblock

Keywords such as system chart, indexing, creation chart are highlighted in bold red. The full list of keywords can be found in the BON textual grammar[22]. If there is a reference from one ClassBON to another, that reference must be of that class type. E.g. in a cluster chart there is a reference to a cluster. This cluster must be created before it can be referenced. In this case you cannot reference to a scenario, it must be a cluster. The CS follows the Ecore model by including all elements and having the same multiplicity as in the model. An identifier must be written in uppercase. An identifier could be the name of the system chart. In BON an identifier does not have to be in uppercase but we have chosen to do so to keep a strict way of modeling.

4.3 Test

To check if the system works according to the requirements one needs to test the system. The informal BON editor will be tested with acceptance and unit tests.

4.3.1 Acceptance testing

One or more features would be complicated to test using unit tests which is why we use acceptance tests. These features are:

- Keywords highlighting
- Syntax highlighting
- Type checking

The actual results of the tests can be seen in figure 4.2 and figure 4.3.

Test case ID	Test case	Test description	Expected result	Actual result	Passed / Failed
T01	Keywords highlighting	Test that keywords such as "system chart" or "indexing" are highlighted	Keywords are highlighted	The keywords are highlighted in bold red	Passed
T02	Comment highlighting	Test that comments are written in italic and is colored green	Comments are highlighted	The comments are highlighted in green and italic	Passed
T03	String highlighting	Test that strings are colored blue	Strings are highlighted	The strings are highlighted in a blue color	Passed
T04	Type checking	Test that references to objects are correct. In an event chart it must be possible to reference to the system chart and not the cluster chart	References are correct types	The references are correct types	Passed

Table 4.1: Acceptance tests for informal BON editor

4.3.2 Unit testing

The model of the informal BON editor will be tested using JUnit. It should be possible to create one system chart. It should be possible to create zero or more class charts, cluster charts, event charts, scenario charts and creation charts. There are 8 tests. In each test we test if it's possible to create the following: informal charts, system chart, cluster chart, event chart, scenario chart and creation chart. The last two tests are for two class charts. The results of the tests can be seen on figure 4.4. The tests finished after 0,227 seconds and all 8 tests passed.

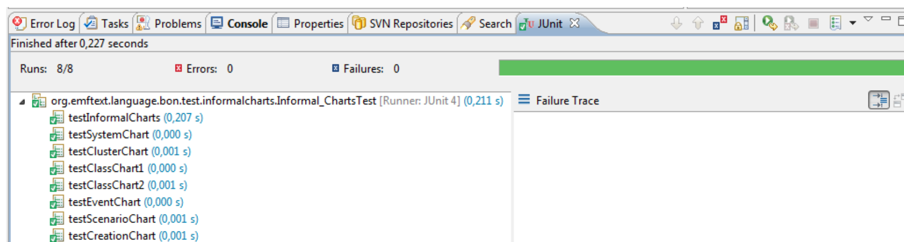


Figure 4.4: JUnit test results

Chapter 5

Codegeneration from BON to Java

In this chapter we describe the generation of Java code from informal charts. The chapter describes the modifications made to the Ecore model, the model to text transformation, solving the issue of multiple inheritance, the merging of user modified code and generated code, the configuration of Javadoc and the test results of the code generation.

5.1 Modifications to the Ecore model

The Ecore model shown in figure 4.1 has been modified to help the generation of Java code. In order to know which cluster a class or a cluster belongs to, a class and a cluster has a reference to a cluster named parent.

5.2 Informal BON to Java transformation

The methods for creating Java classes, packages and Javadoc top-level description are in the Java class GenerateJavaCode. These methods are invoked from

the constructor of Informal_chartsImpl. An adapter is added to the object of the class which initializes when it receives a notification of the event type *REMOVING_ADAPTER*. The reason for this is, when the user saves the Informal BON model, the event type *REMOVING_ADAPTER* is received. When the event type is received, the BON model is validated using the basic validity checker provided by EMF. If the severity equals *OK*, the generation of Java code can begin. The Java code for this is shown in figure 5.1.

```
112⊖    /**
113      * <!-- begin-user-doc --> <!-- end-user-doc -->
114      *
115      * @generated NOT
116      * @author Stephen Sarquah
117      */
118⊖    protected Informal_chartsImpl() {
119        super();
120        final GenerateJavaCode generator = GenerateJavaCode
121                .getGenerateJavaCode();
122        generator.setInformalCharts(this);
123⊖       Adapter adapter = new AdapterImpl() {
124⊖           public void notifyChanged(Notification notification) {
125                if (notification.getEventType() == Notification.REMOVING_ADAPTER && !generator.isGeneratingJava2Bon()) {
126                    if (Diagnostician.INSTANCE.validate(
127                            generator.getInformalCharts()).getSeverity() == Diagnostic.OK) {
128                        try {
129                            generator.generateCode();
130                        } catch (CoreException e) {
131                            e.printStackTrace();
132                        }
133                    }
134                }
135            }
136        };
137        this.eAdapters().add(adapter);
138    }
```

Figure 5.1: Informal_chartsImpl constructor

Using Eclipse JDT we can search the workspace for a project with the same name as the system name of the system chart. If it isn't found, a Java project with that name is created. The Java code for finding the project is shown in figure 5.2.

```
185⊖    private boolean findProject(IProject[] projects) {
186         projectname = informalCharts.getSystem_chart().getName().substring(0,1).toUpperCase()+
187                 informalCharts.getSystem_chart().getName().substring(1,informalCharts.
188                         getSystem_chart().getName().length()).toLowerCase();
189         for (IProject runtimeproject : projects) {
190             if (runtimeproject.getName().equals(projectname)) {
191                 project=runtimeproject;
192                 return true;
193             }
194         }
195         return false;
196     }
```

Figure 5.2: Private method named findProject

If a bin folder for the project doesn't exist, it is created and added to the project. If a source folder for the project with the name src doesn't exists it is created and the default Java Runtime Environment (JRE) is added to the project. After this the parent of all clusters and classes are set. The Java code for setting the parent of all clusters and classes can be seen in figure 5.3.

```
378     private void setParents() {
379         setParentsOnClusters();
380         setParentsOnClasses();
381     }
382
383     private void setParentsOnClasses() {
384         for (int i = 0; i < informalCharts.getClass_charts().size(); i++) {
385             for (int j = 0; j < informalCharts.getCluster_charts().size(); j++) {
386                 for (int k = 0; k < informalCharts.getCluster_charts().get(j).getClass_entries().size(); k++) {
387                     if (informalCharts.getCluster_charts().get(j).getClass_entries().get(k).
388                             equals(informalCharts.getClass_charts().get(i).getClass_())) {
389                         informalCharts.getClass_charts().get(i).getClass_().
390                         setParent(informalCharts.getCluster_charts().get(j).getCluster());
391                     }
392                 }
393             }
394         }
395     }
396
397     private void setParentsOnClusters() {
398         for (int i = 0; i < informalCharts.getCluster_charts().size(); i++) {
399             for (int j = 0; j < informalCharts.getCluster_charts().get(i).getClusterchart_entries().size(); j++) {
400                 informalCharts.getCluster_charts().get(i).getClusterchart_entries().get(j).
401                 setParent(informalCharts.getCluster_charts().get(i).getCluster());
402             }
403         }
404     }
```

Figure 5.3: Private method named setParents

The Java packages corresponding to the clusters are created in the correct hierarchy. The description, indexing and explanation of a cluster chart is written to a Java file in the package using JET. The Java file is named package-info.java. The JET template is named clusterBON.javajet. The Java code for creating the Java packages can be seen in figure 5.4.

```
197     /**
198      * This method creates the Java packages
199      */
200     private void createPackages(IJavaProject javaProject, IProgressMonitor monitor, EList<Cluster> clusters, IFolder src) {
201         for (Cluster cluster : clusters) {
202             String name=cluster.getName().toLowerCase();
203             try {
204                 javaProject.getPackageFragmentRoot(src).createPackageFragment(name, false, monitor);
205             } catch (JavaModelException e) {
206                 e.printStackTrace();
207             }
208             for (Cluster_chart clusterChart : informalCharts.getCluster_charts()) {
209                 if (clusterChart.getCluster().equals(cluster)) {
210                     createPackages(javaProject, monitor, clusterChart.getClusterchart_entries(), src.getFolder(name));
211                     break;
212                 }
213             }
214         }
215     }
```

Figure 5.4: Private method named createPackages

This ensures that information about the cluster is preserved as Javadoc in the implementation. After the packages are created, the Java classes are created containing information from the class charts, event charts and creation charts as Javadoc. The information are description, indexing and explanation from the class_chart, events from the events chart if it involves the class and creations from the creation chart if the class is the creator. The JET template to create the Java classes is named informalBON.javajet. When the Java classes are created, the top-level description of the system is created in a file called overview.html. The file contains the explanation of the system chart embedded into HTML.

This serves as a top-level description of the system for Javadoc. The template to create this is named systemBON.htmljet.

5.2.1 Multiple inheritance

When modeling in BON it is allowed to have classes derive from multiple classes allowing multiple inheritance by inheriting fields and methods from those classes. However, in Java this is not allowed. Java supports single inheritance. A class can only derive from one class except the Object class which has no superclass. To solve the issue with multiple inheritance in BON and single inheritance in Java, the generated Java class will only inherit from one class which is the first in the list of classes to inherit from in the informal BON model. The textual informal BON editor supports multiple inheritance but the code generation doesn't so the user of the tool must be aware of this when designing the system. The user must keep in mind that this tool is designed to be used for integration between informal BON and Java. Java allows multiple implementations of interfaces however informal BON doesn't make use of interfaces.

5.2.2 Merging Java code

Under certain circumstances, one doesn't want the codegenerator to overwrite the changes made to Java source code. This could happen if additional Javadoc comments are added to the Java class, which are specific to Java and not BON. To solve this problem, JMerger is used to merge user modified code with generated code. The setup and merging of JMerger can be seen in figure 5.5.

Based on the rules we made for JMerger, it searches the body of a class and the Javadoc comments of a class for the annotation **@generated**. If the tag is modified, the code generator will not generate any code for the Javadoc comments or body of a class. The rules can be seen in appendix A.

5.3 Configuration of Javadoc

When the Java project, containing the Java packages and Java classes, is created Javadoc documents can be generated. Javadoc gives a more structured overview of the Java project as a webpage in HTML. The Javadoc documentation can contain a title, which should be the same as the Java project. The overview must

```
//Setup the JMerger
JMerger merger = getJMerger();
merger.setSourceCompilationUnit(merger.createCompilationUnitForContents(result));
try {
    merger.setTargetCompilationUnit(
            merger.createCompilationUnitForInputStream(
                    new FileInputStream(file.getLocation().toFile())));
} catch (FileNotFoundException e1) {
    e1.printStackTrace();
}
merger.merge();
InputStream mergedContents = new ByteArrayInputStream(
        merger.getTargetCompilationUnit().getContents().getBytes());
//Merge content
file.setContents(mergedContents, true, false, monitor);
try {
    newContents.close();
} catch (IOException e) {
    e.printStackTrace();
}
```

Figure 5.5: Merging code with JMerger

be set to the file called overview.html, which is in the folder called doc inside the Java project. Options must be set in order for the Javadoc to recognize and display the custom Javadoc annotations. This can be seen in listing 5.1.

```
1 -tag bon.indexing:a:"Indexing:"
2 -tag bon.query:a:"Queries:"
3 -tag bon.command:a:"Commands:"
4 -tag bon.constraint:a:"Constraints:"
5 -tag bon.creator:a:"Creation chart:"
6 -tag bon.event:a:"Event chart:"
7 -tag bon.explanation:a:"Explanation:"
```

Listing 5.1: Javadoc configuration

This will make it easier to get an overview of the Java project and packages as shown in figure 5.6 and the classes as shown in 5.7.

Persons

A person is a programmer using this tool

See: Description

Packages

Package	Description
organization	This represents all persons
organization.test2	Test2 is inside an organization
organization.test2.test	Test is inside test2
organization.test2.test.workgroup	A workgroup consists of persons
organization.test2.test.workgroup2	Another workgroup consists of persons

A person is a programmer using this tool

Figure 5.6: Javadoc of the generated Java packages

organization.test2.test

Class Person

java.lang.Object
  organization.test2.test.Person

Direct Known Subclasses:

Programmer

```
public class Person
extends java.lang.Object
```

A person is a normal guy

Author:

Stephen Sarquah John Doe Jane Doe

Indexing:

in_cluster: Workgroup
created: 31-01-2013
revised: 31-01-2013

Queries:

Name of person's mother
Sex of person
age of person
Skills
Color of skin

Commands:

Change age

Constraints:

Person must have skills
Person has a mother

Creation chart:

Person creates Programmer
Person creates Tester

Event chart:

incoming: Make person a programmer
incoming: Turn programmer into tester
outgoing: Buy pizza

Explanation:

A person is a normal guy which people derive from

Figure 5.7: Javadoc of the generated Java classes

5.4 Acceptance testing

The features of the code generation will be tested using acceptance tests. The tests can be seen in the following table:

Test case ID	Test case	Test description	Expected result	Actual result	Passed / Failed
T05	Generate Java project	Test that a Java project is created based on the name of the system chart	Java project is created with the system chart name as name of the project	Java project is created with the system chart name as name of the project using Java naming convention	Passed
T06	Generate Java packages	Test that Java packages are created according to the cluster charts	Java packages are created according to the cluster charts	Java packages are created according to the cluster charts in the correct structure and the names are all in lowercase	Passed
T07	Generate Java classes	Test that Java classes are generated in the correct packages	Java classes are generated in the correct packages	Java classes are generated in the correct packages and name of classes is compliant with Java naming convention	Passed
T08	Generate bin folder	Test a bin folder is generated for the Java project	A bin folder is generated for the Java project	A bin folder is generated for the Java project as an output for class files	Passed
T09	Set JRE on Java project	Test that a JRE is set on the Java project	JRE is set on the Java project	The default JRE is set on the Java project	Passed

T10	Apply information to Java class based on class chart	Test that indexing, explanation, queries, commands and constraints are generated as Javadoc custom annotations on the class	Class chart information is generated as Javadoc custom annotations	Class chart information is generated as Javadoc custom annotations	Passed
T11	Apply information to Java class based on creation charts	Test that if a class is a creator in a creation chart, that creation information should be generated as a Javadoc custom annotation in that class	Creation chart information is generated as Javadoc custom annotations	Creation chart information is generated as Javadoc custom annotations	Passed
T12	Apply information to Java class based on event charts	Test that if an event involves a class, that event should be generated as a Javadoc custom annotation in that class	Event chart information is generated as Javadoc custom annotations	Event chart information is generated as Javadoc custom annotations	Passed
T13	Apply information to Java class based on the system chart	Test that information in the system chart is generated as text	System chart information is generated as text	System chart information is generated as text in a file called overview.html	Passed
T14	Apply information to Java class based on scenario charts	Test that information from the scenario charts are generated as text	Scenario charts are generated as text	Information from scenario charts are not found in the generated files	Failed

T15	User modified code	Test that if a user has modified the generated Java code and modified the generated custom tag in the Javadoc then changes will not be overwritten	Changes are not overwritten if the generated tag is modified	Changes are not overwritten	Passed
T16	Generate classes from model in same project	Test that if a general project with the same name as the system chart, contains the informal charts then that project will be converted to a Java project containing packages and classes	Project is converted to a Java project	Project is converted to a Java project	Passed
T17	Create class chart with special characters as identifier	Test if the class chart can contain special characters such as "$" and ":", as identifiers	Java class is generated according to the information in the class chart	iBONText does not accept "$" and ":" as identifiers	Failed
T18	Create class chart which has a class that inherits from itself	Test if a class in a class chart can inherit from itself	Class in a class chart should not be able to inherit from itself	Class in class chart can inherit from itself	Failed

Table 5.1: Acceptance tests of codegeneration from BON to Java

An example of the generated Java project, packages and classes can be seen in figure 5.8

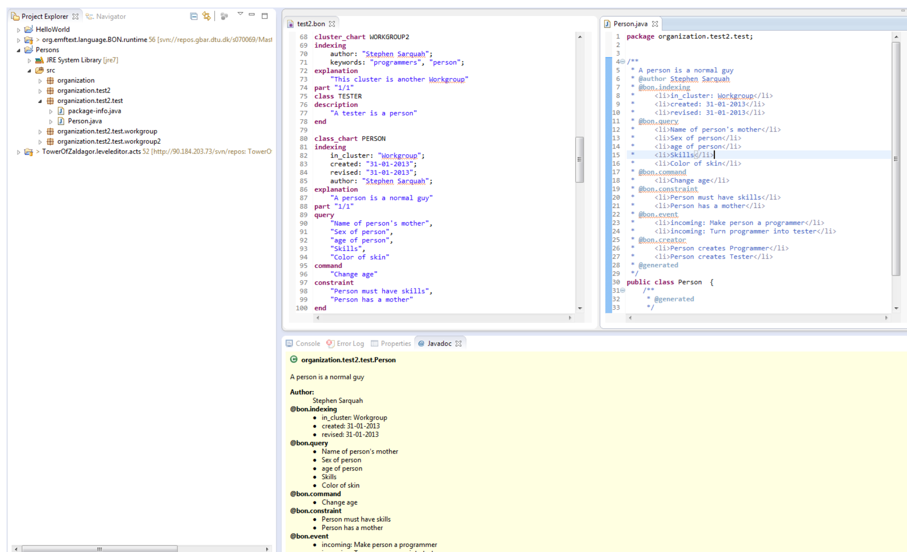


Figure 5.8: The generated Java code for an Informal BON model

CHAPTER 6

Codegeneration from Java to BON

This chapter describes how informal charts will be generated from Java source code to a bon file. The creation of an abstract syntax tree, the issue of inheritance and the test results will be described.

6.1 Initiation

In order to generate informal charts from Java source code and save it to a bon file, a menu must be created for starting the codegeneration. The menu is an extension which needs to be added to a plugin and is called org.eclipse.ui.menus. A command must also be created so when the user presses **Generate BON model** the codegeneration begins. The command is the extension called org.eclipse.ui.commands. The plugin.xml used for iBONText can be seen in appendix B. The codegeneration is handled by the class BONHandler which will be described in the next section.

6.2 Abstract Syntax Tree

In order to create the informal charts model an AST must be created containing the abstract syntax of the Java source code. The AST will be created based on the Javadoc of Java packages and classes in the class called BONHandler.

First we must iterate through the Java project and find all Java packages. This is shown in figure 6.1. Then we iterate through all ICompliationUnits (Java source files) of the package and create an AST for retrieving information from the Java source code, which can be Javadoc. This is shown in figure 6.2.

```
214    private void analysePackages(IProject project) throws JavaModelException {
215        IPackageFragment[] packages = JavaCore.create(project)
216                .getPackageFragments();
217        for (IPackageFragment javapackage : packages) {
218            if (javapackage.getKind() == IPackageFragmentRoot.K_SOURCE) {
219                createAST(javapackage);
220            }
221
222        }
223    }
224
```

Figure 6.1: Private method named analyzePackages

```
590    //Reads an ICompilationUnit and creates the AST for manipulating the Java source file
591    private CompilationUnit parse(ICompilationUnit unit) {
592        ASTParser parser = ASTParser.newParser(AST.JLS4);
593        parser.setKind(ASTParser.K_COMPILATION_UNIT);
594        parser.setSource(unit);
595        parser.setResolveBindings(true);
596        return (CompilationUnit) parser.createAST(null);
597    }
```

Figure 6.2: Private method named parse

Information for BON clusters are stored in the Javadoc of Java files called package-info.java. Whenever package-info.java is encountered then a cluster chart can be added to the informal charts model. The hierarchy of the clusters can be determined by the structure of the Java packages. Package-info.java contains the description of the cluster, which starts from the first line of Javadoc without any tags to the first Javadoc tag. Package-info.java can contain the explanation and indexing of a cluster chart. The explanation has a Javadoc custom tag which is **@bon.explanation**. The indexing has the Javadoc custom tag **@bon.indexing** where each index identifier is contained within the HTML tag <**li**>. Within each index identifier there can one to many index strings where each index string is separated by a comma. An example of this can be seen in listing 6.1. Each of these elements are added to the cluster chart or the cluster of the cluster chart and then added to the informal charts model.

```
1 /**
2  * @bon.indexing
3  *      <li>keywords: cluster chart, Organization </li>
4  */
```

Listing 6.1: Indexing Javadoc custom tag

Java classes has the same name as BON classes and the BON classes are stored within a cluster with the same name as the Java package. Information for BON classes are stored in the Javadoc of Java classes. The description, indexing and explanation for BON classes and class charts are stored in the same way as for BON clusters and cluster charts. The Java classes may contain queries, commands, constraints, events for event charts and creations for creation charts. Queries has the Javadoc custom tag **@bon.query** where each query is contained within the HTML tag <**li**>. Commands has the Javadoc custom tag **@bon.command** where each command is contained within the HTML tag <**li**>. Constraints has the Javadoc custom tag **@bon.constraint** where each constraint is contained within the HTML tag <**li**>. A BON event is only present within a Java class if the BON event involves the BON class. An event chart can have the event type incoming or outgoing meaning all the events in the event chart is either incoming or outgoing. Events has the Javadoc custom tag **@bon.event** where each event is contained within the HTML tag <**li**> along with the event type. An example of this can be seen in listing 6.2.

```
1 /**
2  * @bon.event
3  *      <li>incoming: Make person a programmer</li>
4  */
```

Listing 6.2: Event Javadoc custom tag

Creations are stored within a Java class if the creator of the creation is equal to the BON class. Creations has the Javadoc custom tag **@bon.creator** where each creation is contained within the HTML tag <**li**>. Indexing, explanation, queries, commands and constraints are added to the class chart, while the description is added to the class of the class chart. Incoming events are added to the event chart containing incoming events and outgoing events are added to the event chart containing outgoing events. Creations are added to one creation chart and hereafter the informal charts are then added to the informal charts model.

When the informal charts model is created then a textual informal BON file can be created. The textual informal BON file is created using JET and the template is named Java2BON.bonjet. The informal charts model is iterated through and

the system chart, cluster charts, class charts, event charts and creation charts are written to the textual informal BON file.

6.2.1 Inheritance in class charts

In order to add inheritance to the class chart, we need to retrieve the class which the class of the class chart derives from. This is done by adding the class ExtendVisitor which is derived from ASTVisitor, which can be seen in figure 6.3. ExtendVisitor checks if an AST node is derived from another class, if it has a super class. If it has a super class then the field type is set and we're able to retrieve the super class.

```
 7 public class ExtendVisitor extends ASTVisitor {
 8
 9     private Type type;
10
11     @Override
12     public boolean visit(TypeDeclaration node) {
13         if (node.getSuperclassType()!=null) {
14             type=node.getSuperclassType();
15         }
16         return super.visit(node);
17     }
18
19     public Type getType() {
20         return type;
21     }
22 }
```

Figure 6.3: ExtendVisitor class

6.3 Acceptance testing

The features of the code generation will be tested by using acceptance tests. The tests can be seen in the following table:

Test case ID	Test case	Test description	Expected result	Actual result	Passed / Failed

T19	Generate bon file	Test that a bon file is created with the same name as the Java project selected	A bon file is created with the same name as the Java project	A bon file is created with the same name as the project inside a folder called bon	Passed
T20	Generate system chart	Test that a system chart is created with explanation and clusters	A system chart is created with name as the Java project. The system chart has clusters and an explanation	A system chart is created with the same name as the selected Java project and the system chart contains clusters with a description. The system chart also has an explanation	Passed
T21	Generate cluster charts	Test that cluster charts are generated and the cluster charts contains indexing, explanation and a combination of clusters and/or classes	Cluster charts are created containnig indexing, explanation and a combination of clusters and/or classes	Cluster charts are created containnig indexing, explanation and a combination of clusters and/or classes	Passed
T22	Generate class charts	Test that class charts are generated and the class charts contains indexing, explanation, queries, commands and constraints	Class charts are created containing indexing, explanation, queries, commands and constraints	Class charts are created containing indexing, explanation, queries, commands and constraints	Passed
T23	Generate event charts	Test that event charts are generated and the event charts contains the type of events, events and which class the events involves	Event charts are generated and the event charts contains the eventtype, events and which class the events involves	Event charts are generated and the event charts contains the eventtype, events and which class the events involves	Passed

T24	Generate creation charts	Test that creation charts are generated and the creation charts contains the creator class and which class it creates	Creation charts are generated and the creation charts contains the creator class and which class it creates	Creation charts are generated and the creation charts contains the creator class and which class it creates	Passed

Table 6.1: Acceptance tests of codegeneration from Java to BON

Chapter 7

Future work

This chapter describes which features could be added to future development of iBONText based on the test results from table 4.1 5.1 and 6.1.

7.1 Scenario charts

As it is made now, scenario charts from the informal charts aren't generated as Javadoc. This can be done by generating the information of the scenario charts to the file overview.html since the scenario charts illustrate important aspects of the overall behavior of the system.

7.2 Concrete syntax

When creating informal charts it is not possible to use special characters inside a simple_string. The characters **"$"** and **":"** cannot be used. It is possible that the use of other special characters would result in an error. To solve this problem, the concrete syntax must be corrected to include special characters in simple_strings.

7.3 Ecore model

When creating a class chart it is possible for a class to inherit from itself. This should not be possible. To add this feature, a constraint in the Ecore model must be made. The inherit must not be equal to the class of the class chart.

7.4 Java to BON

When generating the textual informal BON models from Java, in some cases there's spacing in the simple_strings. A text might look like the following:
" This cluster represents the base level of the system". The spacing must be deleted so the informal charts and the Javadoc is ideally identical.

7.4.1 Seamless codegeneration

The generation of the informal charts from Java isn't a seamless process. One must select a Java project in the Java view then press Java to BON in the menu, then press Generate BON model. The generation of the informal charts from Java source code should start whenever the user presses save.

7.5 Graphical notation

When making informal charts, it is much easier to get an overview of a system if the informal charts use a graphical notation. Diagrams are much easier to read than a textual notation. The textual notation should provide a foundation for the graphical notation. A way to implement this could be using Graphical Modeling Framework (GMF) where you can develop graphical editors based on EMF and GEF[14].

Chapter 8

Conclusion

This thesis has demonstrated how to design and implement a textual informal BON editor called iBONText which features seamless integration between textual informal BON and Java. The editor follows the grammar for textual informal BON provided by Kim Waldén and Jean-Marc Nerson in 1994. The editor parses and type checks the input. If the input is correct, Java source code and Javadoc with custom annotations will be generated from the textual informal BON. Textual informal BON can be generated from Java source code. iBONText has been tested using JUnit tests and acceptance tests. The Java source code maintains its traceability from the textual informal BON. This means the time to understand the requirements and the system's behavior is much less compared to reverse engineering the system from the Java source code, making the software development more efficient.
The textual informal BON editor is missing improvements which remains to be done. Scenario charts aren't generated as Javadoc meaning the behavior of the system isn't found in the Java source code. When writing a simple_string in the textual informal BON editor, it is not possible to use special characters. The class of the class chart can inherit from itself. When generating informal charts from Java source code, simple_strings contains spaces. The process of generating informal charts from Java source code isn't seamless. In this report there has been suggested solutions to these improvements. iBONText encourages the use of informal BON in software engineering and maintains traceability from the analysis to the implementation.

Appendix A

JMerger rules

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <merge:options
3    xmlns:merge="http://www.eclipse.org/org/eclipse/emf/codegen/
       jmerge/Options">
4
5    <merge:dictionaryPattern name="generatedMember"
6      select="Member/getComment" match="\s*@\s*(generated)\s*\n" />
7
8    <merge:pull targetMarkup="^generated$" sourceGet="Method/getBody"
9      targetPut="Method/setBody" />
10
11   <merge:pull targetMarkup="^generated$" sourceGet="Member/
       getComment"
12     targetPut="Member/setComment" />
13
14   <merge:pull targetMarkup="^generated$" sourceGet="Type/
       getSuperclass"
15     targetPut="Type/setSuperclass" />
16
17 </merge:options>
```

Appendix B

Plugin.xml

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <?eclipse version="3.0"?>
3  <!--
4  -->
5  <plugin>
6     <extension point="org.eclipse.emf.ecore.generated_package">
7        <package
8              uri="http://www.emftext.org/language/BON"
9              class="org.emftext.language.BON.BONPackage"
10             genModel="metamodel/BON.genmodel"/>
11    </extension>
12    <extension
13          point="org.eclipse.ui.commands">
14       <category
15             id="org.emftext.language.BON.commands.category"
16             name="Sample Category">
17       </category>
18       <command
19             categoryId="org.emftext.language.BON.commands.category"
20             id="org.emftext.language.BON.commands.BONHandler"
21             name="Generate BON model">
22       </command>
23    </extension>
24    <extension
25          point="org.eclipse.ui.handlers">
26       <handler
27             class="org.emftext.language.BON.javatobon.BONHandler"
28             commandId="org.emftext.language.BON.commands.BONHandler
                  ">
```

```
29          </handler>
30      </extension>
31      <extension
32              point="org.eclipse.ui.bindings">
33          <key
34                  commandId="org.emftext.language.BON.commands.BONHandler
                       "
35                  contextId="org.eclipse.ui.contexts.window"
36                  schemeId="org.eclipse.ui.
                       defaultAcceleratorConfiguration"
37                  sequence="M1+6">
38          </key>
39      </extension>
40      <extension
41              point="org.eclipse.ui.menus">
42          <menuContribution
43                  locationURI="menu:org.eclipse.ui.main.menu?after=
                       additions">
44              <menu
45                      id="org.emftext.language.BON.menus.BONHandler"
46                      label="Java to BON"
47                      mnemonic="M">
48                  <command
49                          commandId="org.emftext.language.BON.commands.
                               BONHandler"
50                          id="org.emftext.language.BON.menus.BONHandler"
51                          mnemonic="S">
52                  </command>
53              </menu>
54          </menuContribution>
55          <menuContribution
56                  locationURI="toolbar:org.eclipse.ui.main.toolbar?after=
                       additions">
57              <toolbar
58                      id="org.imm.dtu.s070069.java.toolbars.sampleToolbar"
                           >
59                  <command
60                          commandId="org.emftext.language.BON.commands.
                               BONHandler"
61                          id="org.emftext.language.BON.toolbars.BONHandler"
62                          tooltip="Say hello world">
63                  </command>
64              </toolbar>
65          </menuContribution>
66      </extension>
67  </plugin>
```

APPENDIX C

User guide

This document describes how to run iBONText.

iBONText is a plugin to Eclipse, therefore to run iBONText one needs to download and install Eclipse from http://www.eclipse.org/downloads/. Information about installing Eclipse can be found at http://wiki.eclipse.org/Eclipse/Installation. iBONText has been developed using Eclipse 4.2. Other versions haven't been tested.

Download the source code for iBONText at https://github.com/sarquah/iBONText.

Open Eclipse. From the menu press **File** then **Import**, as shown on figure C.1.

Expand **General** and select **Existing Projects into Workspace** as shown on figure C.2.

Select **Select root directory:** and browse to the projects downloaded as shown on figure C.3. Select all projects then press **Finish**.

The projects should now be imported and iBONText is almost ready to run. Right-click on the project **org.emftext.language.BON**, go to **Run as** and select **Run Configurations...** as shown on figure C.4.

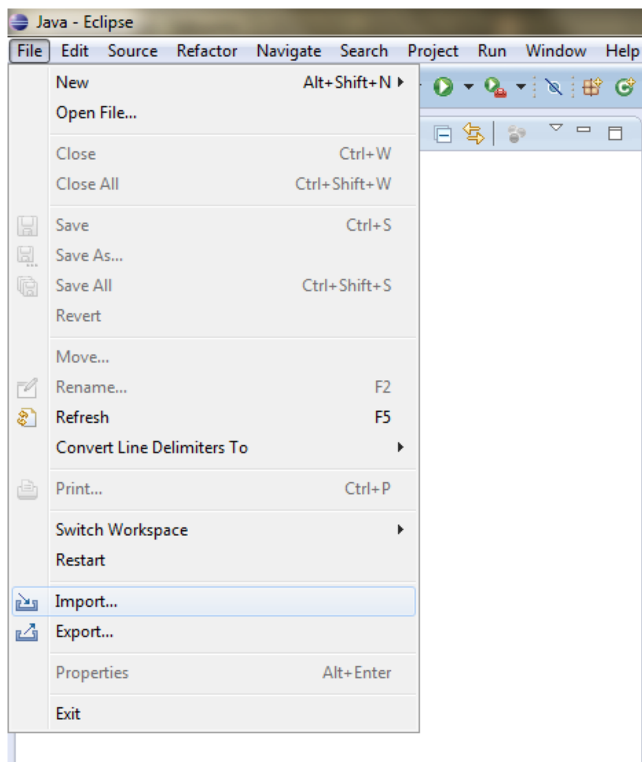


Figure C.1: Import command

Select **Eclipse Application** and press **New launch configuration**. Name your launch configuration and make sure the **Runtime JRE** is the latest version as shown in figure C.5.

Next select the tab **Arguments**. Remove all text in **VM arguments** and insert **-XX:MaxPermSize=128m** to make sure there's enough memory at runtime. Press **Apply** then **Run**. This is shown in figure C.6.

Select **File** –> **New** –> **Project** and navigate to **EMFText Project**. Select **EMFText bon project** and press next. This is shown in figure C.7.

Give your project a name then press **Finish**. You should now have a project with a bon file containing errors as shown in figure C.8.

To generate Java source code and Javadoc custom annotations edit your informal charts and press save. If the model is verified, Java source code and Javadoc

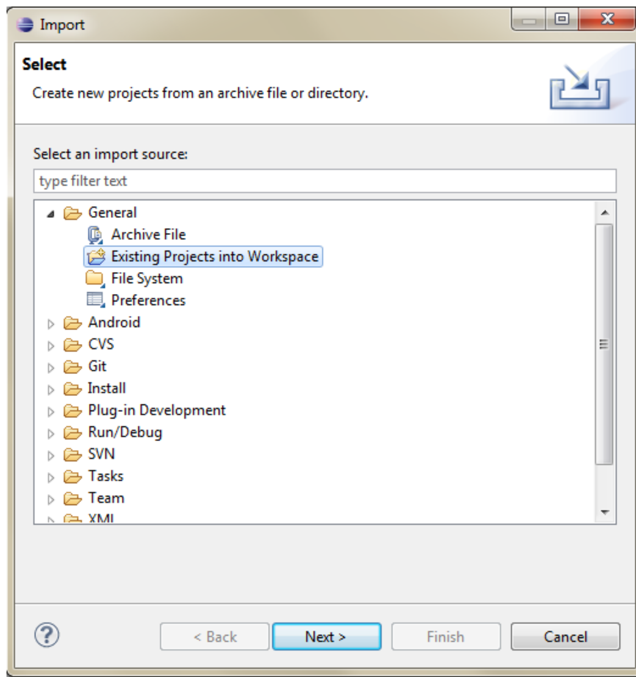


Figure C.2: Import wizard

custom annotations will be generated as shown in figure 5.8.

To generate textual informal BON from Java source code select your Java project in the view called **Java**. In the menu press **Java to BON** –> **Generate BON model** as shown in figure C.9.

A bon file will be generated in the folder **bon** of the Java project.

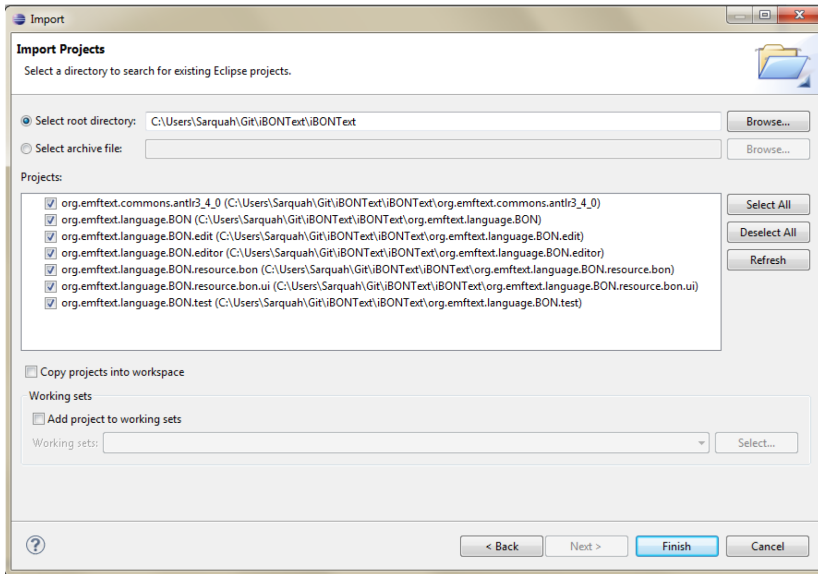


Figure C.3: Select archive file to import

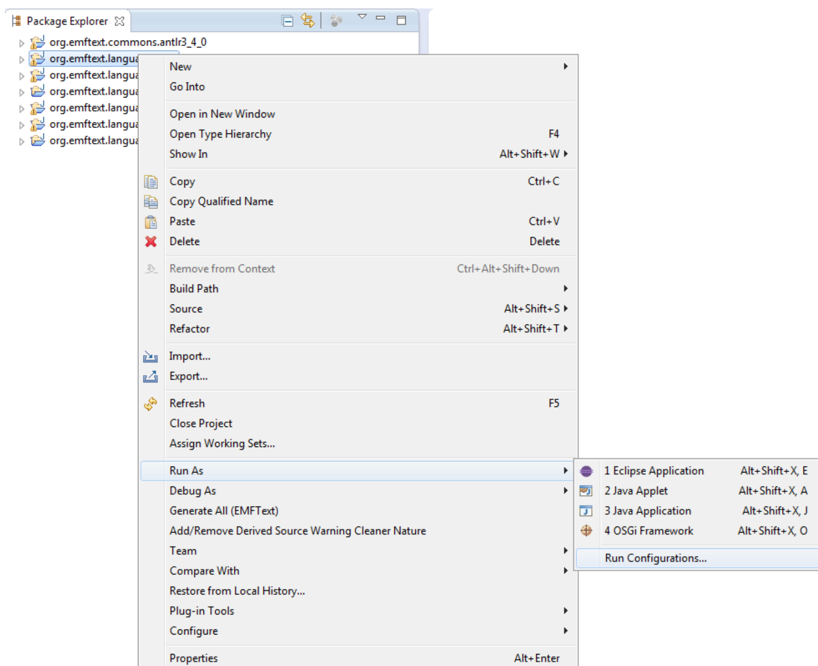


Figure C.4: Select Run Configuration

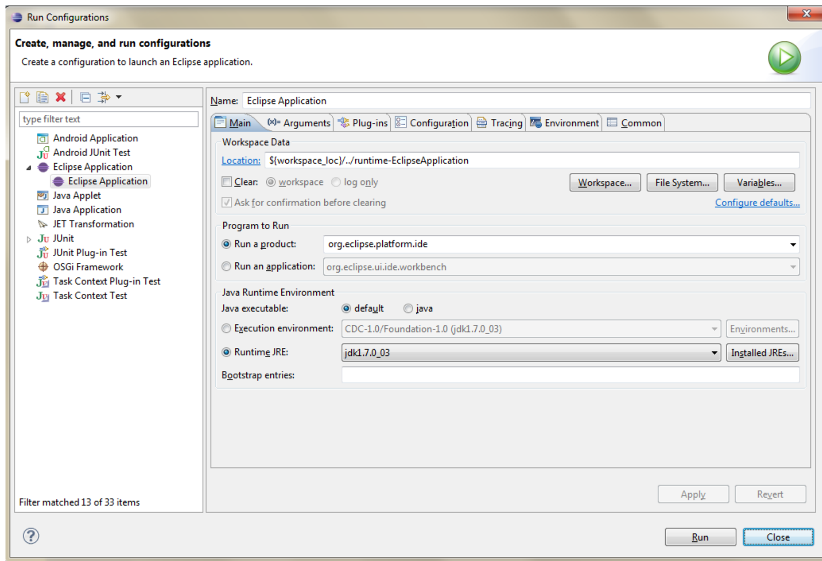


Figure C.5: Runtime JRE

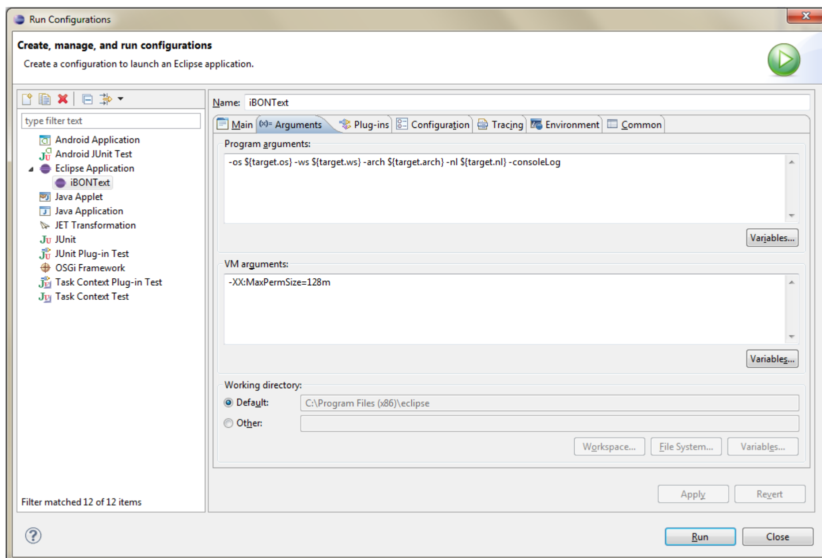


Figure C.6: Run configurations

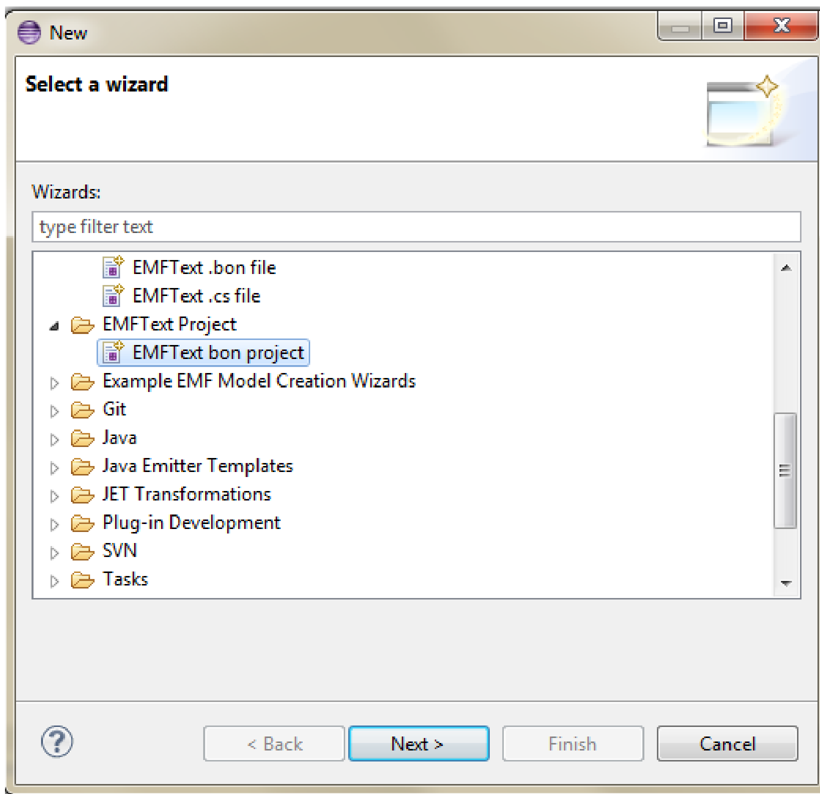


Figure C.7: Select a wizard

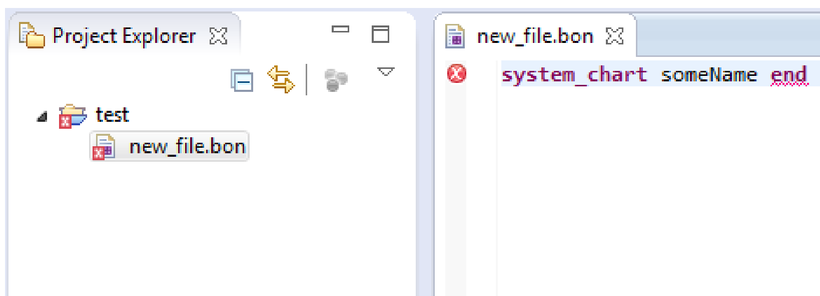


Figure C.8: Textual informal BON editor

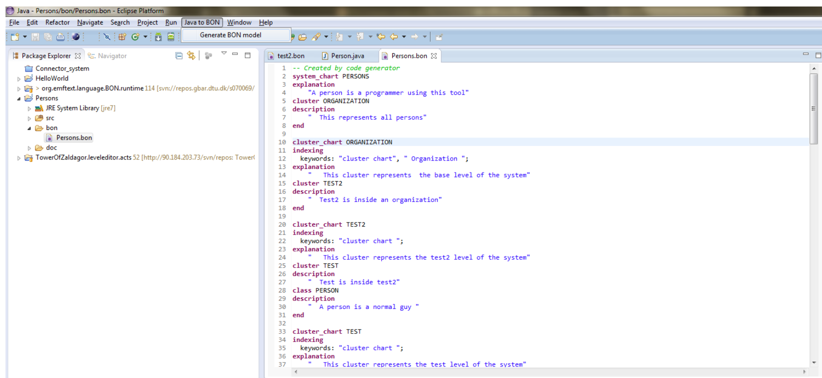


Figure C.9: Generate BON model

Bibliography

[1] ANTLR. http://www.antlr.org/.

[2] Apache Ant. http://ant.apache.org.

[3] Frederick Brooks. No silver bullet - essence and accident in software engineering. page 4, 1986.

[4] Cascading Style Sheets. http://www.w3.org/standards/webdesign/htmlcss.

[5] Eva Darulova and Fintan Fairmichael. http://kindsoftware.com/products/opensource/Beetlz/.

[6] Eclipse. http://www.eclipse.org.

[7] Eclipse Java Development Tools. http://www.eclipse.org/jdt/overview.php.

[8] Eclipse Modeling Framework. http://www.eclipse.org/modeling/emf/?project=emf#emf.

[9] EMFText. http://www.emftext.org.

[10] EMFText guide. http://www.emftext.org/index.php/EMFText_Documentation.

[11] Extended Backus–Naur Form. http://www.cl.cam.ac.uk/~mgk25/iso-14977.pdf.

[12] Richard F. Paige and Jonathan S. Ostroff. A comparison of the business object notation and the unifed modeling language. Technical report, Department of Science, York University.

[13] Fintan Fairmichael. *http://kindsoftware.com/products/opensource/BONc/*. PhD thesis.

[14] Graphical Modeling Framework. http://www.eclipse.org/modeling/gmp/.

[15] Object Management Group. *MOF Model to Text Transformation Language (MOFM2T), 1.0*. http://www.omg.org/spec/MOFM2T/1.0/.

[16] Human-Usable Textual Notation. http://www.omg.org/spec/HUTN/.

[17] Javadoc. http://docs.oracle.com/javase/7/docs/technotes/tools/windows/javadoc.html.

[18] JMerge. http://wiki.eclipse.org/JET_FAQ_What_is_JMerge%3F.

[19] Craig Larman. *Applying UML and patterns: an introduction to object-oriented analysis and design and iterative development*. Prentice Hall, 2004.

[20] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, 1991.

[21] Aidan Morrissey. http://kindsoftware.com/documents/reports/Morrissey10.pdf. Master's thesis.

[22] Kim Waldén and Jean-Marc Nerson. *Seamless Object-Oriented Software Architecture*. Prentice Hall, 1994.

[23] XML Metadata Interchange. http://www.omg.org/spec/XMI/.