A Rigorous Methodology for Analyzing and Designing Plug-ins

Anne E. Haxthausen
DTU Compute
Technical University of Denmark
DK-2800 Lyngby, Denmark
Email: ah@imm.dtu.dk

Joseph Kiniry
DTU Compute
Technical University of Denmark
DK-2800 Lyngby, Denmark
Email: jkin@imm.dtu.dk

Marieta V. Fasie
DTU Compute
Technical University of Denmark
DK-2800 Lyngby, Denmark
Email: marietafasie@gmail.com

***Abstract*—Today, GUI plug-ins development is typically done in a very ad-hoc way, where developers dive directly into implementation. Without any prior analysis and design, plug-ins are often flaky, unreliable, difficult to maintain and extend with new functionality, and have inconsistent user interfaces. This paper addresses these problems by describing a rigorous methodology for analyzing and designing plug-ins. The methodology is grounded in the Extended Business Object Notation (EBON) and covers informal analysis and design of features, GUI, actions, and scenarios, formal architecture design, including behavioral semantics, and validation. The methodology is illustrated via a case study whose focus is an Eclipse environment for the RAISE formal method's tool suite.**

I. Introduction

What is the paper about.

A. Background

What problems do we run into when starting building an Eclipse plug-in.

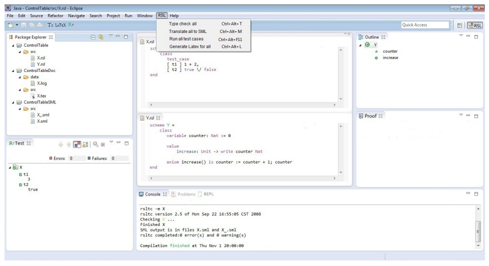
B. Related work

What solutions have other papers brought

II. Analysis and design method

This section describes the methodology used for analyzing and designing plug-ins. The methodology comprises a number of five steps, each described in a separate subsection and presented in the order in which they should be applied. These five steps are: *user interface, events, components, components communication* and *code generation*.

The methodology is illustrated via a case study whose focus is an Eclipse environment for the RAISE formal method's tool suite. In other words, the case study aims to develop an Eclipse plug-in for RAISE tool set. This case study is a real project, meant to be used in an academic environment and it will be used to illustrate how the five steps can be applied in practice. Due to space restrictions the entire project's analysis and design phases can not be presented in this paper, therefore only one scenario of the plug-in is chosen to be shown throughout the five steps. This way, the reader can see the evolution from one step to another, and can apply the method in the same way for different scenarios.


Java - ControlTable/src/X.rsl - Eclipse
File Edit Source Refactor Navigate Search Project Run Window RSL Help
Type check all Ctrl+Alt+T
Translate all to SML Ctrl+Alt+M
Run all test cases Ctrl+Alt+F11
Generate Latex for all Ctrl+Alt+L
Quick Access
Package Explorer
ControlTable
src
X.rsl
Y.rsl
ControlTableDoc
data
X.log
src
X.tex
ControlTableSML
src
X_.sml
X.sml
RтTest
Errors: 0
Failures: 0
X
t1
3
t2
true
Outline
Y
counter
increase
Proof
Console
Problems
REPL

```
  class
    test_case
    [ t1 ] 1 + 2,
    [ t2 ] true \/ false
end
```

```
scheme Y =
  class
    variable counter: Nat := 0

    value
      increase: Unit -> write counter Nat

    axiom increase() is counter := counter + 1; counter
end
```

```
rsltc -m X
rsltc version 2.5 of Mon Sep 22 16:55:05 CST 2008
Checking X ...
Finished X
SML output is in files X.sml and X_.sml
rsltc completed:0 error(s) and 0 warning(s)

Compilation finished at Thu Nov 1 20:00:00
```

Fig. 1. Eclipse user interface displaying the RSL menu item

A. User interface

The purpose of the first step is to determine the plug-in functionality from the user point of view. This means identifying all, or a subset of all the things a user can do inside the user interface. This is done by identifying the requirements and establishing the user interface in the same time. Therefore, for each user action that is relevant and important for the plug-in, a mock-up user interface is being created. Of course, if more user actions are similar, they can be grouped under a single user interface. It is up to the plug-in developer to determine what are the most important functionalities and how or if he wants to prioritize them.

The mock-up user interface can be a vague hand made sketch or a precise drawing made with an advanced graphical editing program.

For the Eclipse case study, it was decided, for example, that a user should have the possibility to type check all RSL files. Also the user should have the possibility to translate all files to SML, to run all test cases existing in all files and to generate Latex documents for them. Therefore it was decided that this four actions can be grouped under a menu item which is called *RSL* and presented in the same user interface. Figure 1 illustrates the graphical user interface for the *RSL* menu item.

While the user interface is drawn, the requirements are documented in EBON using *scenario_charts* elements. The beautiful part about using EBON from the beginning is that it allows the requirements specification to be captured using natural language. Therefore no intermediate step is required between identifying the requirements and documenting them. And to demonstrate it, this is how the *scenario_charts* for the requirements presented in Figure 1, looks like:

```
scenario_chart MENU
scenario "MENU1"
description "The
user can type check all RSL files in the workspace. Success or failure
messages will be displayed along with the list of errors in case of a
failure"
scenario "MENU2"
description "The user can translate to SML all RSL files in the
 workspace. Success or failure messages will be displayed along with
the list of errors in case of a failure"
scenario "MENU3"
description "The user can run all test cases in the workspace.
Success or failure messages will be displayed along with the list of
errors in case of a failure"
scenario "MENU4"
description "The user can generate Latex files for all files in
the workspace. Success or failure messages will be displayed along
with the list of errors in case of a failure"
```

B. Events

Incoming events representing user actions and *outgoing* events meant to inform the user.

C. Components

Major components captured in BON *static_diagram*s using *cluster_chart* and *class*.

D. Components communication

Component interfaces added to the interface diagram using *feature*, *require* and *ensure*. This will later result in plug-in extensions and extension points.

Update scenarios with events.

E. Code generation

Beetlz generates the Java code from BON specification.

III. Conclusion

In conclusion

Acknowledgment

The authors would like to thank...

References

[1] H. Kopka and P. W. Daly, *A Guide to LaTeX*, 3rd ed. Harlow, England: Addison-Wesley, 1999.