# A Rigorous Methodology for Analyzing and Designing Plug-ins

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

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

***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.

About the case study: RAISE method [1], RAISE Specification Language [2]

### B. Related work

What solutions have other papers brought

## II. Analysis and design method

This section describes a methodology used to analyze and design plug-ins. The methodology has six stages, each described in a separate subsection and presented in the order in which they are applied. These six steps are: *domain modeling*, *user interface*, *events*, *components*, *components communication* and *code generation*.

The methodology is illustrated on a case study taken from a real project in which an Eclipse environment for the RAISE formal method's tool suite is being developed. Due to space restrictions, the entire project's analysis and design phases are not presented here; instead, only one scenario of the plug-in is shown from beginning to end.

### A. Domain Modeling

The first step when analyzing and designing a system is to establish its domain model. This is done in order to create a common vocabulary between those involved in the project and to identify the concepts used in the product development process. This means that the most important entities and terminology related to the system domain must be identified, explained documented from the very beginning so they can be unanimous understood and used throughout the entire product life cycle.

-Should we write that is "at a very high level" and "also meant for non technical people"? Because in our case we are a little technical -Also where do we describe the case study? -mvf

Looking at the case study name and description, the domain model is constructed by analyzing areas like Eclipse, RAISE and graphical user interface (GUI). The result is a list of terminologies along with their explanation, essentially describing entities and elements from a high level point of view. Some examples from the list are notions like *editor, console, typechecker, translator, SML translator, LaTeX generator, SML compiler, RSL Perspective* and so on. Some of these items can be grouped in a bigger entity, while others are big enough to covers multiple notions. For example *SML translator* and *LaTeX generator* can be grouped under the *translator* notion, since both are referring to the process of transforming a RSL specification into another type of specification. Likewise *RSL Perspective* can be seen as a notion that comprises all other items since inside Eclipse all RAISE elements can be grouped under a single perspective.

All those described before can easily be captured in EBON using *system_chart*, *cluster_chart* and *class_chart* elements. The notions that have been identified are documented as classes, which can be grouped under clusters and all these are composing a big and unique system. Listing 1 illustrates a caption of the RAISE System.

### B. User interface

The purpose of this step is to determine the plug-in functionality from the user's point of view. This means identifying all the things a user can do from the plugin's user interface (UI). This UI feature set consequently derives the requirements for the product (the plugin) and designs the UI in the same

```
system_chart RAISESystem
cluster RAISEPerspective
description "The Eclipse Rasie perspective. It contains all
components and functionality
relevant for a RAISE project "
end

cluster_chart RaisePerspective
class Editor
description "The RSL text editor"
class Console
description "Displays the output of different components
e.g. TypeChecker, SMLTranslator etc"
class TypeChecker
description "The RSL syntax and type checker"
class SMLCompiler
cluster Translator
description "Contains all translators applicable for the RSL
code"
end

cluster_chart Translator
class SMLTranslator
description "Translates RSL code to SML code"
class LatexGenerator
description "Integrates RSL specification in Latex"
end
```

Listing 1. System chart describing the RAISE system.

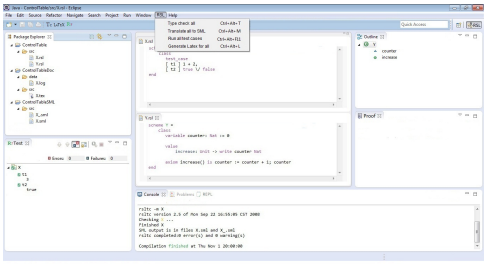

Fig. 1. Eclipse user interface displaying the RSL menu item

time. Therefore, for each user action that is relevant and important for the plug-in, a mock-up user interface is created. If many 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 features and how, or if, she wants to prioritize them.

The mock-up user interface can be a vague handmade sketch or a precise drawing made with an advanced graphical editing program. The intention here is not presentation and precision, but instead feature completeness and UI consistency.

For the Eclipse case study, it was decided, for example, that a user should have the possibility to typecheck all RAISE Specification Language (RSL) files (the primary file type of the RAISE tool suite). 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 these four actions should be grouped under a menu item which is called *RSL* and presented in the same UI, for consistency and simplicity.

These figures will have to span both columns to be viewable, I suspect. -jrk

Figure 1 illustrates the graphical user interface for the *RSL* menu item. This illustration was created by taking a screenshot

```
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"
```

Listing 2. Scenario chart for RSL menu.

of Eclipse and then hand-editing the resulting image in just a few minutes.

While the user interface is being drawn, product requirements are documented using EBON *scenario_chart* 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. For the case study, the requirements associated with the user interface in Figure 1 are captured in the *scenario_charts* in Listing 2.

### C. Events

In this section the entire system is seen as a black box. The focus is on the external actions that make the system react and on the system outgoing responses. However, not all systems outgoing events are of interest, but only the ones that are started by external stimuli.

An incoming external event is any action that determines the system to change its state. For example it can be a user clicking a button or another system sending a request. An outgoing internal event is the response the system sends to the one that initiated the incoming external event. The system outgoing event for the action of pressing the button could e.g. be the display of a new window or writing a message to the standard output.

Looking back at the scenario presented in subsection II-B, the user has the possibility to type check all RSL files. This is illustrated in Figure 1 by the presence of a sub-menu item named *Type check all*. Therefore, the incoming external action in this case is: *the user selects the Type check all sub-menu item*. And this external event has been determined just by looking at the scenarios previously identified. However there is another user event that triggers the same system reaction and that is using the shortcuts: *The user presses Ctrl+Alt+T*.

Once established, the user actions can be captured in EBON using *event_chart* elements. The *event_chart* can be ingoing or outgoing based on which events they capture. Since the two user incoming events that have just been identified aims for the same functionality, they are grouped under the same name and captured in Listing 3.

```
event_chart UserActions
incoming
explanation "External events triggering representative system
behaviour"
event "TYPECHECKALL: User clicks RSL menu and then clicks on
Type Check all option or presses Ctrl+Alt+T"
involves ProjectExplorer, TypeChecker, Console,
ConsoleToProblems, ProblemsView
```

Listing 3. Incoming event chart for typechecking features.

```
event_chart UserMessages
outgoing
explanation "Internal events triggering responses meant to
inform the user."
event "CONSOLEUPDATE: Successs or failure messages displayed
in console" involves Console, TypeChecker, SMLTranslator
event "PROBLEMSUPDATE: Problems view update" involves
TypeChecker, Console, ConsoleToProblems, ProblemsView
```

Listing 4. Outgoing event chart for typechecking features.

All incoming actions trigger changes in the system state. And the next task is to decide how should the system notify the one triggering the action, about the changes that have taken place. For the RAISE case study it was decided that after the user selects the *Type check all* sub-menu item, a message should be displayed on the standard output. The message informs the user about how the typechecking evolved and since the case study GUI is Eclipse based, the standard output is considered by default the Eclipse Console view. However in some cases the typechecking may not be successful due to some errors in the input files. In this case it would be nice to know what caused the problem and where can it be found. Therefore the system will present the necessary information in the Eclipse Problem view. To sum up, after the user selects the *Type check all* sub-menu item, the system updates the Console and Problem views with appropriate information. Listing 4 presents the two events captured in an outgoing *event_chart* under the names of *CONSOLEUPDATE* respectively *PROBLEMSUPDATE*.

### D. Components

Major components captured in BON *static_diagram*s obtained from *cluster_chart*, *cluster_chart* and *class_chart*.

-High level classifiers into concrete data types in language independent fashion.

-"fully typed class interfaces and formal specification of software contracts"

### E. Components communication

How components are arranged.

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.

### F. Code generation

Beetlz generates the Java code from BON specification.

## III. Conclusion

In conclusion


## Acknowledgment

The authors would like to thank themselves by buying tons of chocolate and beer. They really deserve it.


## References


[1] T. RAISE Method Group, *The RAISE Development Method*, ser. BCS Practitioner Series. Prentice Hall, 1995, available by ftp from ftp://ftp.iist.unu.edu/pub/RAISE/method_book.

[2] T. RAISE Language Group, *The RAISE Specification Language*, ser. BCS Practitioner Series. Prentice Hall, 1992.