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

Plugins, especially in the realm of plugins that wrap existing research command-line tools, are notoriously badly designed. Academics simply do not have the resources and expertise to execute on the design and implementation of a quality plugin. Partly this is due to the fact that there are few examples of best practices in the area, and partly it is because plugin development is viewed as the dirtiest of the dirty-but-necessary jobs of "selling" systems technology.

Typically, a researcher has developed a novel tool for Java programming, lets call it the `CommandLineWidget`. They want others to use this tool, but few people these days want to mess about with downloading and building source code and, sadly, the barrier to entry for command-line tools is not insignificant these days. Instead, the researcher wants to "sell" their tool by wrapping it in the `CommandLineFeature` for Eclipse, since Eclipse has the mind-share of most Java developers. But the researcher does not know how to think about the UI design of an Eclipse plugin, design and program the plugin, nor are the really interested in learning how to do these things.

A. Background

Eclipse plugin development is a tricky world. Concepts like features, plugins, extension points, windows, views, etc. abound. Enormous poorly documented APIs are prolific in the Eclipse ecosystem. To implement even the most basic of features sometimes takes hours of digging to find the right three lines of code, and then those lines change when a new major version of Eclipse comes out. This is a frustrating environment for researchers who want to package their demonstrable, useful tools for the Eclipse IDE.

This work is an attempt to help resolve these issues. First, we provide a *rigorous step-wise methodology through which one can do the analysis, architecture design, and UI design of a plugin for an arbitrary IDE*. Second, we provide *template example plugins* that can be reused by a programmer with a good understanding of Java but a poor understanding of Eclipse plugin development to develop a new Eclipse plugin or feature.

The methodology used is based upon the Business Object Notation, an analysis and design methodology promoted by Walden and Nerson in the mid-90s within the Eiffel community [**?**]. Ostroff, Paige, and Kiniry formalized parts of the BON language and reason about BON specifications [**?**], [**?**], [**?**]. Fairmichael, Kiniry, and Darulova developed the BONc and Beetz tools for reasoning about BON specifications and their refinement to JML-annotated Java [**?**], [**?**], [**?**]. Finally, Kiniry and Fairmichael have extended BON in a variety of ways to produce Extended BON (EBON), which permits one to add new domain-specific syntax and semantics to the core BON language [**?**], [**?**].

The methodology is illustrated on a case study taken from a real project in which an Eclipse environment *eRAISE* for the RAISE formal method [1] and specification language (RSL) [2] is currently being developed.

Originally RAISE was supported by the *eden* tool suite [3] that was successfully applied in a range of industrial project. However, as this tool suite was only available for SUN workstations, a new tool suite *rsltc* [4], [5], portable for any platform supporting the C language, was developed in 1998–2008.

The *rsltc* tool suite consists of a type checker and some extensions to it supporting activities such as pretty printing, extraction of module dependencies, translation to other languages, generation of proof obligations, formal verification, and generation and execution of test cases. It provides a command-line interface with different capabilities selected by options, and can also be used from Emacs, using a menu to select these capabilities. *rsltc* was successfully applied in industrial projects. However, although it is easy to use, the provision of a modern Eclipse based development environment for *rsltc* would be a great addition.

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 high level classifiers related to the system domain must be identified, explained and 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.

In the method described in this paper, such notions are captured using the EBON *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. Please notice how easy it is to capture notions and their description in EBON by just using natural language.

But the domain model comprises also terms describing what can be done with the notions already defined. In EBON this is translated using *query* and *command*. A *command* is a service that a *class* can provide, while the *query* is a request for information from a specific class. Looking at the case study domain model presented in Listing 1, *Console* offers the service of displaying informative messages or error messages. And the *command* is joined by a constraint stating that the console must be cleared before displaying a new message.

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

cluster_chart RSLPerspective
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 RSL
modules"
end

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

class_chart Console
indexing
in_cluster: "RSLPerspective"
explanation "Displays the output of different components
e.g. TypeChecker, SMLTranslator"
command "Displays informative or error messages",
constraint "Delete content before displaying a new message"
```

Listing 1. System chart describing 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 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.

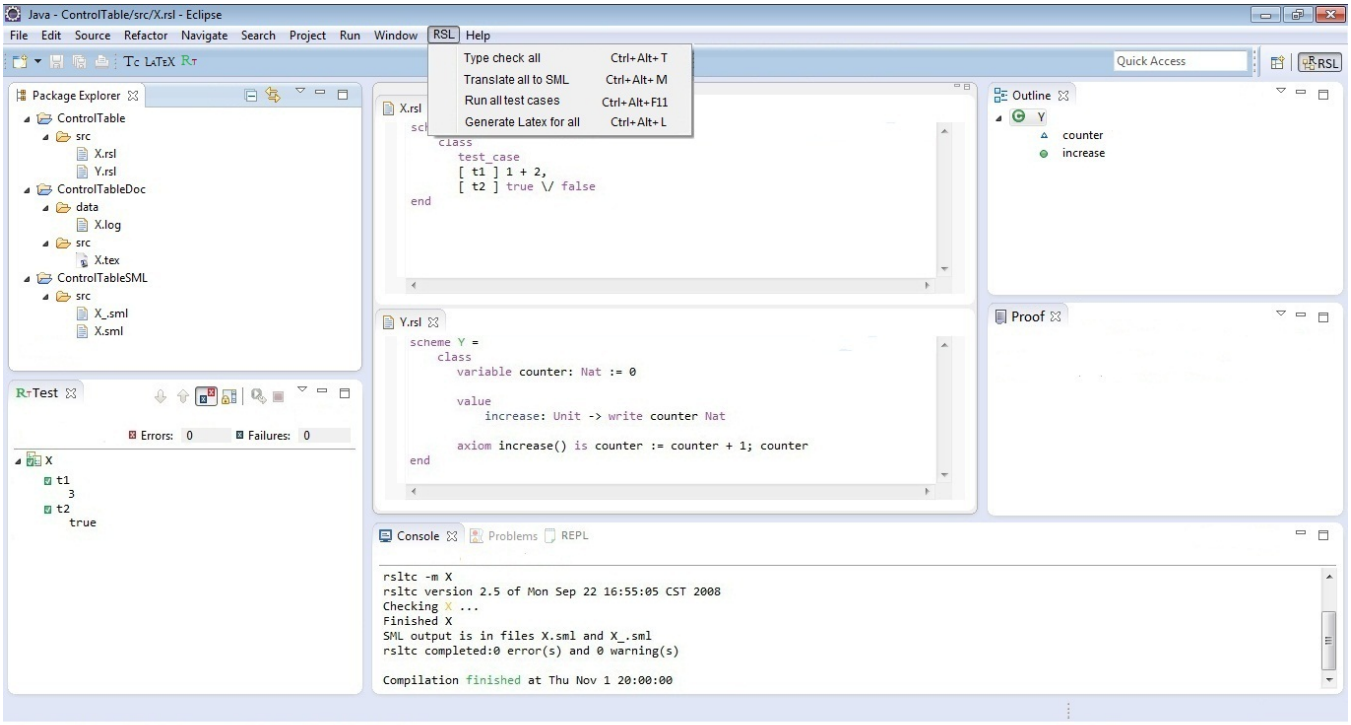

Fig. 1. Eclipse user interface displaying the RSL menu item

```
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.

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 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 depending on the type of the events they capture. Since the two user incoming events that have just been identified aims for the same functionality, they are grouped

```
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 TypeChecker, Console, ConsoleToProblems, Problems
```

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
event "PROBLEMSUPDATE: Problems view update"
involves TypeChecker, Console, ConsoleToProblems, Problems
```

Listing 4. Outgoing event chart for typechecking features.

under the same name (*TYPECHECKALL*) and captured in Listing 3. Please ignore for the moment the *involves* part in Listing 3, since next section, subsection II-E explains in detail how this part is obtained.

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*. Listing 4 also contains an *involves* part, but it will be explained in detail in next section, subsection II-E.

D. Components

This subsection, in contrast with the previous one, looks inside the system, at the components that form its architecture. These components can be referring to concrete elements like a *Zoom in button* or they can be abstract concepts like *User authentication* which covers everything in the system responsible for authenticating a user. Also multiple components can be grouped in a bigger component which can also be part of an even bigger component.

The place to start identifying the system architecture components is the domain model presented in subsection II-A. The high level classifiers captured there must be transformed into concrete data types in order to bring the system development closer to the implementation phase. The advantage of using EBON is that it simplifies the transit between the domain

```
static_diagram SystemArchitecture
--eRAISE system architecture
component
        cluster RSLPerspective
        component
                class Editor
                class Console
                class TypeChecker
                class Problems

                cluster Translator
                component
                        class SMLTranslator
                        class LatexGenerator
```

Listing 5. System architecture caption

model and architecture and manages to capture the concrete data types in language independent fashion. This is done by taking the entities captured in *system charts*, *cluster charts* and *class charts* and transfer them in *static diagrams*. An EBON *static_diagram* contains multiple *components* which can be *clusters* or *classes* and which have the same meaning as the ones composing the *system_chart* in subsection II-A.

Applying this step on the case study, the *static_diagram SystemArchitecture* presented in Listing 5 is obtained. Please observe the correlation between Listing 5 and Listing 1 and how all clusters and classes from domain modeling remain the same in the static diagram.

E. Components communication

This section concern is how components interact with each other and what interfaces they present to the other components that want to communicate with them. The starting point is the list of incoming and their corresponding outgoing actions identified in subsection II-C. This helps identifying the components that reactfirst to an external stimuli and the components responsible for the outgoing actions. Once the starting and ending point of the data flow is established, the other interacting components are determined by evaluating the scenarios in subsection II-B. In EBON notation, the components communication is seen in terms of *client, supplier* relationship. The component providing the interface is a *supplier* and all components using it are *clients*. In the Eclipse case study, one incoming event is *TYPECHECKALL* presented in Listing 3 and its corespondent outgoing events are *CONSOLEUPDATE* and *PROBLEMSUPDATE* presented in Listing 4. Thus the first component that reacts to user *TYPECHECKALL* action is the *TypeChecker*. The component responsible for the *CONSOLEUPDATE* is the *Console* and the one for *PROBLEMSUPDATE* is *Problems*. The *TypeChecker* component typecheks the input and can directly inform the *Console* about the status. Therefore the *TypeChecker* is a client of *Console* and this is expressed in EBON as: *TypeChecker client Console*. The client relations must be added in the *static_diagram* after the components declaration.

Should I make a new BON example comprising client relations? -mvf

Since *TypeChecker* is directly stimulated by the *TYPECHECKALL* event and it is a client of *Console* which generates the outgoing response, it can be said that *TYPECHECKALL* event involves the *TypeChecker* and the *Console* components.

```
class Console
        feature
        update  -> message: LIST[CHAR]
                -> channel: CHANNEL
                require
                        channel = 1 or channel = 2
                        -- 1-stdout 2-stderr
```

Listing 6. Console component interface

And this is how the *involves* part inListing 3 is constructed. The same method is applied to *PROBLEMSUPDATE* event. This event is triggered if there are errors displayed in the console. Therefore *TypeChecker* sends a message to *Console* and if the message is an error, a third component called *ConsoleToProblems* notifies the *Problems* component. Therefor the *PROBLEMSUPDATE* event involves the *TypeChecker, Console, ConsoleToProblems* and *Problems* components. This is captured in Listing 4.

Once it was decided what components are interacting, it must be established how do they accomplish that. This means establishing the contracts between components by identifying the information a client needs and the messages it sends to its supplier. EBON allows specifying fully typed interfaces in an language independent mode through the notion of *feature*. Everything a component exposes to its clients is called a feature. For example the *Console* component in the case study has a feature called *update* which allows the client components to sent it messages that will further be displayed to the user. Please refer to the Console command in Listing 1 for better understanding. These messages are just strings and can be informative messages or errors. The situation is captured in Listing 6, where the fully typed feature signature can be noticed. When a client calls *update* on the *Console* it must send the message to be displayed to the user and the information about the message being an error message or not. The assertion *require* is a precondition that makes sure that *channel* is one of the values 1 or 2, where 1 is for the standard output and 2 is for the standard error output. EBON has also post condition assertions named *ensure*.

All these software contracts will later, on in the implementation phase, result in plug-ins extensions and extension points.

F. Code generation

Once the analysis and design parts are finished, the next step is to create the code skeleton. This part is done using a tool named Beetlz [**?**], which automatically generates Java code from EBON specification. The input of this tool is the EBON *system_chart* and *static_diagram* that were obtained throughout the entire analysis and design. With just one click, Beetlz converts all EBON specification obtained in steps *A* to *E* into Java elements.

-Joe, can you please add a bib item for the Beetlz tool? I used it the previous paragraph cite{Beetlz}.
-And do we need a reference for Java?
-And the code presented further down is not styled -mvf

For example this is the Console class generated by Beetlz for the component with the same name captured in *static_diagram* in Listing 6:

```
public /*@ nullable_by_default @*/ class Console  {
   //@ ensures channel == 1 || channel == 2; public void
   update(List<Char> message, Channel channel){}
}
```

III. Conclusion

The full specification of our case study is available in a technical report version of this paper. Its appendix contains all of the UI sketches for the various plugins in the eRAISE feature, the high-level concept/domain analysis, the specification of events, the architecture specification, the formal component interface design, and the code generated from that design. The example plugins and feature are also available in our repository at GitHub.

The student responsible for the design and development of this plugin (the first author), had to learn and apply this methodology in only a few short weeks. That is evidence for its utility. Only after completing the plugin will we be able to reflect upon how well the analysis and design match the final implementation, but if past projects using the EBON methodology are any indication, we expect to see no architectural drift and full feature compliance[1]

There are several opportunities for next steps refining this work in the context of plugin development.

First, we would like to augment our advanced command-line options tool suite "CLOPS" for plugin development. CLOPS permits one to declaratively specific the syntax and semantics of the command-line options of a tool. CLOPS reasons about such specifications (it has a formal semantics) and generates a lexer, parser, well-formedness checker, and documentation. Extending CLOPS to generate a set of Eclipse plugins and architecture documentation, like that we hand-write in this methodology, would be a valuable exercise and product.

Second, the community should use this method to wrap several more

References

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

[3] P. Bruun and et al., "RAISE Tools Reference Manual," CRI: Computer Resources International, Tech. Rep. LACOS/CRI/DOC/17, 1995.

[4] "RAISE Tool User Guide," 2008. [Online]. Available: http://www.iist.unu.edu/newrh/III/3/1/docs/rsltc/user_guide/html/ug.html

[5] C. George, "The Development of the RAISE Tools," in *Formal Methods at the Crossroads. From Panacea to Foundational Support*, ser. Lecture Notes in Computer Science, B. K. Aichernig and T. Maibaum, Eds. Springer Berlin Heidelberg, 2003, vol. 2757, pp. 49–64. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-40007-3_4

[1]Note to the reviewers: we expect that development will be complete by the time of the workshop so that we can demonstrate it in San Francisco.