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 R. 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

Plug-ins, especially in the realm of plug-ins 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 plug-in. Partly this is due to the fact that there are few examples of best practices in the area, and partly it is because plug-in 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 plug-in, design and program the plug-in, nor is she really interested in learning how to do these things.

Eclipse plug-in development is a tricky world. Concepts like features, plug-ins, 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 must change when a new major version of Eclipse comes out. This is a frustrating world 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 user interface (UI) design of a plug-in for an arbitrary integrated development environment (IDE)*. Second, we provide *template example plug-ins* that can be reused by a programmer with a good understanding of Java, but a poor understanding of Eclipse plug-in development, to develop a new Eclipse plug-in or feature.

The methodology used is based upon the Business Object Notation (BON), an analysis and design methodology promoted by Walden and Nerson in the mid-90s within the Eiffel community [1]. Ostroff, Paige, and Kiniry formalized parts of the BON language and reasoned about BON specifications [2], [3], [4], [5]. Fairmichael, Kiniry, and Darulova developed the BONc and Beetz tools for reasoning about BON specifications and their refinement to JML-annotated Java.[1] 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 [6].

For the reader who has never heard of EBON, think of it as the subset of UML that might actually have a clear, unambiguous semantics. EBON's core features are that it is *seamless*, insofar as you use the same specification language for everything from domain analysis to formal architecture specification and its behavior, *reversible* insofar as code generation and reverse engineering to and from code to EBON is straightforward and tool-supported, and *contracted* as formal abstract state-based contracts (invariants, pre, and postconditions) are the fundamental notion used to specify system behavior. EBON has both a textual and a graphical syntax, a formal semantics expressed in higher-order logic, a formal semantics of refinement to and from OO software, and tool support for reasoning about specifications, expressing specifications textually or graphically, generating code from models and models from code, and reasoning about refinement to code.

The methodology is illustrated on a case study that develops an Eclipse environment for the RAISE formal method [7] and specification language (RSL) [8]. The project is called *eRAISE* and it is currently under development at DTU.

The *rsltc* tool suite [9], [10] consists of a type checker and some extensions to it supporting activities such as pretty

[1] See http://tinyurl.com/brgcrzc for more information.

printing, extraction of module dependencies, translation to other languages, generation of proof obligations, formal verification, and generation and execution of test cases. *rsltc* has a command-line interface that exposes different capabilities selected via switches, but is also used from Emacs using menus and key-binding. However, although it is easy to use for the user comfortable with command-line tools or Emacs, we expect that the creation of a modern Eclipse-based development environment for *rsltc* would broaden its appeal to mainstream software engineers and better enable its use for university-level pedagogy.

mention the fact that the method has been used for 20 years on dozens of projects Questions from the reviewer 1: Are these steps performed in sequence (waterfall model) or in some iterative manner (e.g., spiral)? Also, I do not understand what about this methodology is plug-in development specific? The authors should make it clearer in the paper

II. Related work

There is little published work that focuses on methodologies specific to plug-in development. E.g., Lamprecht et al. reflect over some simplicity principles elicited by many years' experience in plug-in development [11], but do not provide a methodology.

We speculate that there is not much published work because plug-in development was not the focus of scientists until recently. Moreover, it is a fair question whether or not plug-in development is any different from normal systems development where a GUI is involved. We believe that plug-in development is different from normal GUI development as plug-ins must integrate into the larger framework of the IDE, deal with non-GUI events, and work in arbitrary compositions.

III. Analysis and Design Method

The EBON methodology as applied to Eclipse plug-in development has six stages, each described in a separate paragraph 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*.

Step 1: Domain Modeling. In the first step the most important entities and high level classifiers related to the system domain are identified, explained and documented. The identified notions are documented as classes, which can be grouped under clusters and all these make up a unique system. Listing 1 illustrates a caption of the eRAISE System specified in EBON notation

The domain model also describes how concepts behave and how their behavior is constrained. In Listing 1, *class_chart Console* offers the service of displaying informative messages or error messages with the *constraint* that it must be cleared before displaying a new message.

Step 2: User Interface. This step determines all the things a user can do from the plug-in's UI by creating a mock-up user interface for each user action that is relevant and important for the plug-in. While the user interface is being drawn, product requirements are documented using EBON *scenario_charts*.

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

cluster_chart RSLPerspective
class Console
description "Displays the output of different components
e.g. TypeChecker, SMLTranslator etc"

class_chart Console
command "Displays informative or error messages",
constraint "Delete content before displaying a new message"
```

Listing 1. System chart describing the eRAISE system.

```
scenario_chart RSLMENU
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"
```

Listing 2. Scenario chart for the RSL menu item.

Listing 2 presents the requirement for a sub-menu item of RSL menu.

Step 3: Events. In this stage, the system is seen as a black box. The focus is on the external actions that make the system react and on the system's outgoing responses. An incoming external event is any action that determines the system to change its state while an outgoing internal event is the response the system sends as a reaction to an incoming external event. Listing 3 illustrates an incoming external event named *TYPECHECKALL*, triggered by a user action. The system response to this action is captured in an EBON *outgoing event* in Listing 4 .

Step 4: Components. This stage looks *inside* the system at the components that constitute its architecture. The place to start identifying the system architecture components is the domain model created in Step 1. 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. Thus the system chart presented in Listing 1 becomes the *static_diagram SystemArchitecture* from Listing 5

Step 5: Components Communication. This step captures how components interact with each other and what interfaces they present to the other components that want to communicate with them. The events identified in Step 3 help determine the components that react first to external stimuli and the ones 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 Step 2. Listing 3 presents through the *involves* part, the components interacting after the incoming event was triggered. Components contracts are described using parameterized classes that contain formally specified *features*. Listing 6 presents the interface of the *Console* class.

Step 6: Code Generation. In the last step a tool named Beetlz [12] is applied to automatically generate JML-annotated, Javadoc documented Java code from the EBON *system_chart* and *static_diagrams* created in previous steps. Beetlz also performs refinement analysis so that architecture drift is automatically identified as the system evolves, either

```
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
```

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
```

Listing 4. Outgoing event chart for typechecking features.

at the model-level in EBON, or within the implementation in Java.

REFERENCES

[1] K. Waldén and J.-M. Nerson, *Seamless Object-Oriented Software Architecture - Analysis and Design of Reliable Systems*. Prentice–Hall, Inc., 1995.

[2] J. Lancaric, J. Ostroff, and R. Paige, "The BON CASE tool," Details available via http://www.cs.yorku.ca/∼eiffel/bon_case_tool/, Mar. 2002.

[3] J. R. Kiniry, "The Extended BON tool suite," 2001, available via http://ebon.sourceforge.net/.

[4] R. Paige, L. Kaminskaya, J. Ostroff, and J. Lancaric, "BON-CASE: An extensible CASE tool for formal specification and reasoning," *Journal of Object Technology*, vol. 1, no. 3, 2002, special issue: TOOLS USA 2002 Proceedings. Available online at http://www.jot.fm/.

[5] R. F. Paige and J. Ostroff, "Metamodelling and conformance checking with PVS," in *Proceedings of Fundamental Aspects of Software Engineering*, ser. Lecture Notes in Computer Science, vol. 2029. Springer-Verlag, Apr. 2001, also available via http://www.cs.yorku.ca/techreports/2000/CS-2000-03.html.

[6] J. R. Kiniry, "Kind theory," Ph.D. dissertation, Department of Computer Science, California Institute of Technology, 2002.

[7] 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.

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

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

[10] 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

[11] S. Naujokat, A. Lamprecht, B. Steffen, S. Jorges, and T. Margaria, "Simplicity principles for plug-in development: The jabc approach," in *Developing Tools as Plug-ins (TOPI), 2012 2nd Workshop on*, june 2012, pp. 7 –12.

[12] E. Darulová, "Beetlz - BON software model consistency checker for Eclipse," Master's thesis, University College Dublin, 2009.


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

Listing 5. eRAISE system architecture

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

Listing 6. Console component interface