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: aeha@dtu.dk

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

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. 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).*

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 methodand specification language (RSL) [7]. The project is called *eRAISE* and it is currently under development at DTU. RAISE tool suite (*rsltc*) [8], [9] consists of a type checker and some extensions to it supporting activities such as pretty printing, translation to other languages, generation of proof obligations, 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.

II. Analysis and Design Method

The EBON methodology as applied to Eclipse plug-in development has six steps described shortly in the following. These steps can either be performed in sequence or in some iterative manner. More details on the steps and the full

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

```
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 components"
...

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

Listing 1. Excerpts of a system chart describing the eRAISE system.

```
scenario "TypeCheckAllMenu"
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 for a menu in eRAISE.

specification of our case study will be available in a technical report [11].

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.

Step 2: User Interface. In this step, for each user action relevant for the plug-in, a mock-up user interface is drawn, and the requirements for the actions are documented in EBON *scenario_chart* elements. As an example, Listing 2 presents the requirements for one of the menus in the eRAISE case study.

Step 3: Events. This step identifies the external actions that make the system react and the system's outgoing responses. The external actions are captured as *incoming events* and the possible responses as *outgoing events* in EBON *event charts*. For the eRAISE case study, one of incoming events is shown in Listing 3. One of the possible system responses to this action is captured in Listing 4.

Step 4: Components. This step looks *inside* the system at the components that constitute its architecture. The high level classifiers described in the system domain model captured in step 1 are transformed into concrete data types.

Step 5: Components Communication. First, by inspecting the events from step 3 and the scenarios from step 2, it is identified which components interact with each other. Then component interfaces are described using parameterized classes that contain formally specified features.

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 specifications created in the previous steps.

III. 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 [10], 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.

```
event "TYPECHECKALL: User clicks RSL menu and then clicks on
Type Check all option or presses Ctrl+Alt+T"
involves TypeChecker, Console
```

Listing 3. An incoming event in eRAISE.

```
event "CONSOLEUPDATE: Successs or failure messages displayed
in console"
involves Console, TypeChecker
```

Listing 4. An outgoing event in eRAISE.

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

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

[9] 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

[10] 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.

[11] M. V. Fasie, "An Eclipse based Development Environment for RAISE," Master's thesis, DTU Compute, Technical University of Denmark, to appear May 2013.

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