Meta-Level Runtime Feature Awareness for Java

Olszak, Andrzej; Jensen, Martin Lykke Rytter; Jørgensen, Bo Nørregaard

Published in:
In Proceedings of the18th Working Conference on Reverse Engineering (WCRE 2011)

Publication date:
2011

Document version:
Final published version

Citation for pulished version (APA):
Olszak, A., Jensen, M. L. R., & Jørgensen, B. N. (2011). Meta-Level Runtime Feature Awareness for Java. In *In Proceedings of the18th Working Conference on Reverse Engineering (WCRE 2011)* (pp. 271 - 274). IEEE Computer Society Press.

2011 18th Working Conference on Reverse Engineering

Meta-Level Runtime Feature Awareness for Java

Andrzej Olszak, Martin Rytter and Bo Nørregaard Jørgensen
The Maersk Mc-Kinney Moller Institute
University of Southern Denmark
Campusvej 55, 5230 Odense M, Denmark
{ao, mlrj, bnj}@mmmi.sdu.dk

Abstract—**The existing body of feature-location research focuses on discovering feature-code traceability links for supporting programmers in understanding and modifying static artifacts of software. In this paper, we propose a different utilization of this type of reverse-engineering information. We introduce the concept of runtime feature awareness that enables a running program to establish and make use of its own feature-code traceability links. We present an implementation of this idea, a dynamic-analysis Java library called JAwareness. JAwareness provides a meta-level architecture that can be non-invasively added to a legacy program to enable it to act upon the activations of its own features. We demonstrate the feasibility of runtime feature awareness by discussing its three applications: error reporting, usage statistics and behavior adaptation.**

Keywords-features; dynamic analysis; meta-level architecture

I. INTRODUCTION

A *feature* is a unit of software functionality as perceived by a user [1]. In the majority of existing software systems, feature implementations crosscut software abstractions, e.g. components, classes and methods, thus making the software difficult to understand and modify. This problem is addressed by various feature-location approaches that establish traceability links between features and their implementations in unfamiliar source code, examples being [2] and [3].

As we will demonstrate, it is worthwhile to extend the utilization of traceability links beyond supporting *programmers* in working with *source code*. We propose to make it possible for *running programs* to access their own traceability links as they are being dynamically established, and thereby to enable them to interpret their own execution in terms of user-identifiable features. Here, the ability of a running program to observe and act upon the activations of its own features will be referred to as *runtime feature awareness*.

In this short paper we present *JAwareness*, a library-based meta-level architecture that provides runtime feature awareness for Java programs. Our approach establishes traceability links between an executing base-level program and its features, by using lightweight source code annotation and transparent execution tracing. These links are being exposed by the meta-level to the base-level program through an explicit *metaobject protocol* [11]. Using this protocol, a base-level program can reflect on how it is being used by a user and adapt its behavior.

To demonstrate feasibility of this idea, we discuss three applications of runtime feature awareness. Firstly, we show how to make error reports more meaningful by making them feature-aware. Secondly, we present how a program can collect and report information on how it is being used by users. Lastly, we discuss equipping a program with feature-aware logic to help users learning a program and exploring its functionality.

II. RUNTIME FEATURE AWARENESS FOR JAVA

In this section, we introduce the concept of runtime feature awareness and we present JAwareness – a library for runtime feature awareness. We also demonstrate how to apply it in an arbitrary legacy application.

Runtime feature awareness is the ability of a running program to reflect upon activation and deactivation of its own features. Hence, runtime feature awareness implies that feature implementations are *dynamically* located in an executing program and that the established links are accessible to the program being traced at all times.

Traditionally, feature-code traceability has been used for supporting programmers during software development activities, e.g. error correction [4] or restructuring [5]. In contrast, runtime-feature awareness seeks to enable a program to exploit similar information at runtime. To the best of our knowledge, this use of feature location has not been pursued in a systematic way before.

We have implemented runtime feature awareness as a Java library, JAwareness. JAwareness can be seen as a meta-level architecture [6]. The meta-level performs transparent feature tracing. The base-level requests or observes tracing information via an explicit meta-object protocol.

JAwareness is created as an extension to the FeatureTracer dynamic analysis library, whose implementation details can be found in [7]. In this short, paper we will give a high-level overview of how JAwareness works, and how it is to be used and only discuss essential implementation details.

We will use Fig. 1 to illustrate the use of JAwareness. Our example of a base-level program is a drawing application. To make the program feature-aware, the programmer must attach to it the JAwareness meta-level. This is done in three steps.

1095-1350/11 $26.00 © 2011 IEEE
DOI 10.1109/WCRE.2011.40

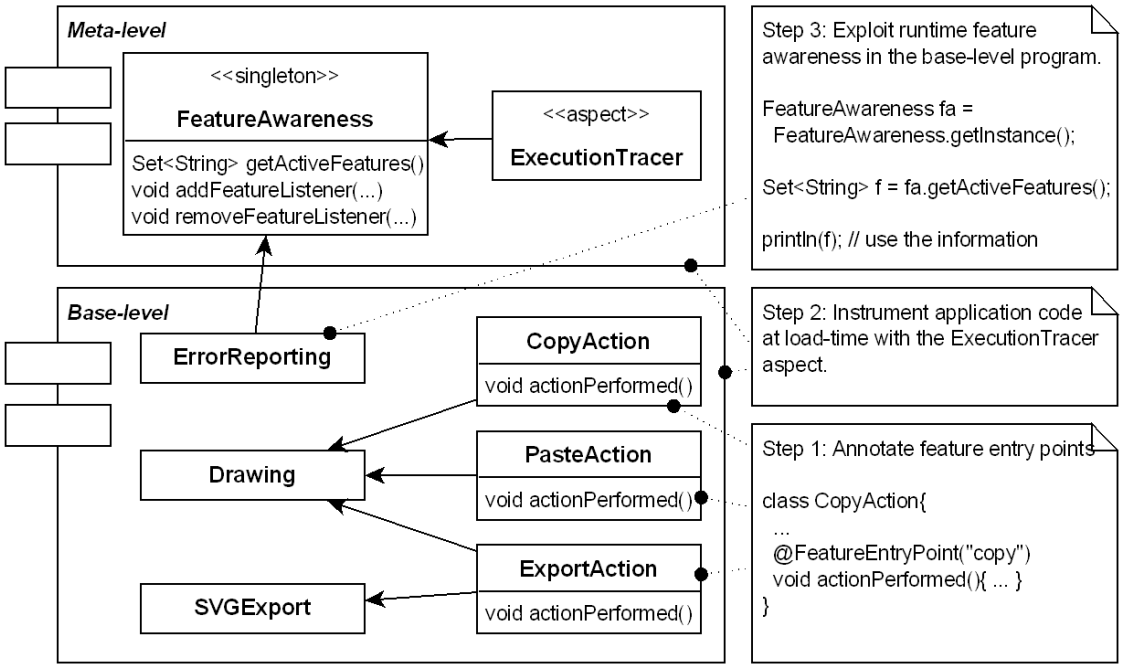

Figure 1. Adding meta-level feature awareness to a simple drawing program.

First, the programmer must annotate features in the target program (see *Step 1* in Fig. 1).

We have divided our drawing program into three features: "copy", "paste" and "export". It is important to note that this set of possible features is not absolute. As discussed in [7], there are different ways of deciding what constitutes a feature. We recommend choosing a set of features that matches a program's requirements.

For each feature, the programmer must specify its *feature-entry points* in the base-level program. A feature-entry point is a method in the code where a program's control flow enters a feature. There may be multiple feature-entry points per feature. Each one is to be specified using a provided Java method-annotation parameterized with the feature name, e.g. @FeatureEntryPoint("copy"). Besides being used for dynamic feature location, the annotations also serve as a useful documentation of the special role of particular program methods.

Second, the program must be instrumented to support meta-level feature tracing (see *Step 2* in Fig. 1).

In JAwareness, we use AspectJ to instrument the base-level program's byte code at load-time [8]. Hereby, the program becomes instrumented with the ExecutionTracer aspect that detects feature *activations* and *deactivations*. When a method annotated with @FeatureEntryPoint is invoked, we say that the corresponding feature is activated. When this invocation has been completed and the method returns (possibly with an exception), we say that the feature is deactivated.

The information about feature activation and deactivation is forwarded by ExecutionTracer to a singleton named FeatureAwareness. FeatureAwareness maintains information about the set of currently executing features. Our model implies that multiple features can be executing simultaneously, either by one feature invoking another, or by being located on separate threads of execution.

Third, code inside a base-level program must be made feature-aware, i.e. it must be made to act upon the available feature information (see *Step 3* in Fig. 1).

The FeatureAwareness singleton exposes the gathered meta-level information through an explicit metaobject protocol to the base-level program. The two possible ways of using this metaobject protocol in a base-level program are to proactively query FeatureAwareness for currently executing features or to register a feature-activation and deactivation listener object and be notified about information changes.

In our drawing program, we use feature execution information to improve error reporting. If the program crashes unexpectedly, it is a good practice to print relevant information to an error log, e.g. a stack trace or an exception message. In our program, we also print information about the features that were executing at the time of the crash. This information makes it easier for the programmers to understand and reproduce the context in which the error happened to a user, i.e. 'problem happened while executing the "copy" feature'. In Section 3.A we will discuss this application of runtime feature awareness in more detail.

In summary, the three steps presented above are all that is required to make an arbitrary Java program feature-aware. JAwareness is freely available for experimentation at [9].

III. APPLICATION SCENARIOS

In this section we discuss three interesting applications of runtime feature awareness. Each of them can be readily integrated into existing software projects by using JAwareness.

A. Feature-Aware Error Reporting

Understanding program error messages can be a complex task. Due to the technical nature of stack traces and exception messages, it is often difficult to connect them with domain-related behavioral contexts. In other words, standard error messages do not explicitly state which features caused them, especially when methods are shared by multiple features.

Feature-centric interpretation of errors is, however, intrinsic to all stages of handling error reports.

During the reception of an error message, programmers must determine the features affected by the error to correctly prioritize it and to allocate the task to a person who knows most about the implementation of a given feature.

To fix an error, it needs to be recognized how to reproduce it by triggering appropriate features. Oftentimes, no such descriptions are reported by the user, and even if they are, they may turn out incomplete or inaccurate. Precise reproduction of an error becomes especially problematic when a precondition of the error is created by a different feature than the one in which the error occurred (so-called *feature interaction*).

After fixing an error, the program must be tested to ensure that no undesired side-effects were introduced. Testing effort can be significantly reduced by knowing which concrete features could have been affected by the bug fix.

Using JAwareness, it is possible to non-invasively enrich error messages with identifiers of their corresponding features. Alternatively, using a logging library, such as java.logging, this can be done by implementing a log preprocessor that transparently prepares all messages with identifiers of features. In the case of Java's System.err stream, messages in the stream can be manipulated by redirecting the stream to a custom handler using the System.setErr method.

Having a program add the names of its active features to the low-level technically-oriented error messages, makes it easier to correlate errors with the activities of a user that caused them. This information supports a programmer in reproducing an error, correcting it and verifying the resulting program.

B. Feature-Aware Program Usage Statistics

Understanding the way the end-users use a program is essential for optimizing effort allocation during several activities in software development and maintenance.

Program usage statistics can be used to prioritize development and maintenance efforts. Given that the most important features are being used most often by the users, it is important to first correct and enhance the features used most often. Correspondingly, a lack of usage of a particular set of features may suggest that the provided features are superfluous, or that something prevents users from using them (e.g. usability issues, lack of documentation).

Program usage statistics can reveal emergent behavioral patterns of the users. Discovery of a particular sequence of feature activations, which a group of users tend to execute during their interaction with a program, hints at an opportunity for improving program usability. Based on such a discovery, programmers may decide to implement the most important of such usage patterns as automated wizards or to merge features that are always being used together.

It is, however, difficult to realize the mentioned scenarios without an explicit link between the usage data and the program's requirements. To establish such a link, and report the usage of actual features and not only methods and classes, we propose applying runtime feature awareness.

Using JAwareness, it is possible to capture and report the set of features being used. By using the FeatureAwareness.addFeatureListener method, one can register an observer object that will get notified about each activation and deactivation of features. Thereby, it is possible not only to tell which features are being activated, but also how often and in which order they are being activated.

Implementing a reporting mechanism that will transfer the data back to the software vendor is then the only step needed for using feature-aware program usage statistics for optimizing development and maintenance efforts.

In comparison to other methods of collecting usage statistics, e.g. UI Gestures in NetBeans IDE [10], usage of runtime feature awareness is localized and non-invasive. Due to our use of meta-level and load-time instrumentation, the logic of a base-level program does not have to be modified to include explicit invocation of any usage-probe methods. Therefore, feature-aware usage statistics can be transparently added and removed from a program without introducing crosscutting concerns to the base-level source code.

C. Feature-Aware Program Adaptation

An interesting concept that builds upon the mentioned feature-aware usage statistics is what we call *feature-aware program adaptation*. The idea here is to let the program itself decide how to react to the usage of its features and thereby to let it adapt to the personalized needs of a particular user. This vision can be realized in a multitude of scenarios.

By knowing what features have already been used by a user, a program can come up with more intelligent proposals in a 'tip of the day' screen, which is commonly used in many applications. This can help a user to discover new features that she might not have been aware of, while avoiding the information about features that she already uses often, and presumably knows well.

Going one step farther, runtime feature awareness can be used to automatically adapt the user interface of a program to the needs of the user. Programmers may equip menus and menu bars of their base-level programs with the ability to adjust the appearance and ordering of the set of menu items or buttons related to the most frequently used features. This way, a program would make it easier for a user to find the features she uses the most. This vision can be achieved by creating a single base class for all GUI action handlers and using the

template method pattern to query features being active when the action is performed. Then, the appearance of the action GUI-triggers can be modified accordingly. Furthermore, the prioritization of actions can be made sensitive to temporal ordering and dependencies between activations of individual features. A base-level program could recognize and learn common usage sequences of a particular user and use this data to adjust the GUI to prioritize features that are most likely to be triggered next.

In the case of long-running features, which either occupy a dedicated thread or are called from a scheduler, runtime feature awareness can be used to inform the user about the progress and let him interrupt the ongoing activities. Using JAwareness, one can easily implement a monitor window that lists all the currently executing features. By inserting additional hooks in each of the features, one can easily enable user-driven cancellation of feature execution or disable the possibility of it being triggered in the future. Furthermore, feature awareness could enable a program to assign higher priority to threads running important features.

D. Discussion

To practically realize the discussed scenarios, we have implemented and integrated them with JHotDraw 7.2 [12]. We have equipped this program with feature-aware error reporting that prints feature-aware usage statistics at program shutdown. We have also implemented feature-aware prioritization of toolbar-triggered features by using a heat-map coloring scheme to display frequency of their usage. Our proof-of-concept implementations are available for reuse and further experimentation at the website of JAwareness [9].

During our experiences with JAwareness, we have not observed any significant performance or memory consumption overhead. The only noticeable slowdown of the base-level application occurs when the AspectJ load-time weaver [8] performs load-time instrumentation of byte-code at program startup. If necessary, this could be eliminated by using the AspectJ compiler to compile base-level programs. Further performance improvements could be achieved by removing the dependency on the FeatureTracer library [7]. Presence of this library enables JAwareness-enabled programs to access their precise feature-methods traceability links (which could be used in some advanced forms of usage statistics or behavioral adaptations), but it also introduces additional computation and memory footprints.

IV. SUMMARY

We have introduced the concept of runtime feature awareness, the ability of a program to recognize execution of its features at runtime and to act upon them. Thereby, we have proposed to apply feature-code traceability to enabling self-monitoring and self-adaptation of running applications based on concrete functionality executed by a user.

We have introduced JAwareness, an implementation of runtime feature awareness for Java. The implementation is essentially a meta-level architecture, where the meta-level performs feature tracing, while the base-level can observe and retrieve this information via an explicit metaobject protocol.

Finally, we have discussed three application scenarios that could be built on top of a runtime feature awareness library such as JAwareness. We believe that the presented application scenarios clearly display the feasibility and practicality of runtime feature awareness.

We recognize that the approach presented in this paper is only the first step towards a practical exploration of the concept of runtime feature awareness and runtime usage of other sources of reverse-engineering information in general. We hope that JAwareness and its sketched application scenarios will encourage researchers and developers to contribute to further exploration of this interesting topic.

REFERENCES

[1] C. R. Turner, A. Fuggetta, L. Lavazza, and A. L. Wolf, "A conceptual basis for feature engineering," J. Syst. Softw., vol. 49, no. 1, pp. 3-15, Dec. 1999.

[2] A. D. Eisenberg and K. De Volder, "Dynamic feature traces: Finding features in unfamiliar code," in ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance, Washington, DC, USA: IEEE Computer Society, 2005, pp. 337-346.

[3] M. Eaddy, A. V. Aho, G. Antoniol, and Y. G. Guéhéneuc, "CERBERUS: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis," in ICPC '08: Proceedings of the 2008 The 16th IEEE International Conference on Program Comprehension, Washington, DC, USA: IEEE Computer Society, 2008, pp. 53-62.

[4] D. Röthlisberger, O. Greevy, and O. Nierstrasz, "Feature driven browsing," in ICDL '07: Proceedings of the 2007 international conference on Dynamic languages. New York, NY, USA: ACM, 2007, pp. 79-100.

[5] A. Mehta and G. T. Heineman, "Evolving legacy system features into fine-grained components," in ICSE '02: Proceedings of the 24th International Conference on Software Engineering. New York, NY, USA: ACM, 2002, pp. 417-427.

[6] C. Zimmermann, “Metalevels, MOPs and What the Fuzz is All About”, Advances in Object-Oriented Metalevel Architectures and Reflection, CRC Press, pp. 3-24.

[7] A. Olszak and B. N. Jørgensen, “Remodularizing Java programs for improved locality of feature implementations in source code,” Science of Computer Programming, In Press, Available online 6 November 2010, ISSN 0167-6423.

[8] AspectJ, http://www.eclipse.org/aspectj/

[9] JAwareness, http://www.featureous.org/jawareness/

[10] UI Gestures, http://platform.netbeans.org/tutorials/nbm-gesture.html

[11] G. Kiczales, J. des Rivières and D. G. Bobrow, “The art of metaobject protocol,” MIT Press, Cambridge, MA, 1991.

[12] JHotDraw, http://www.jhotdraw.org/