The Journal of Systems and Software 000 (2016) 1–23

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

XTRAITJ: Traits for the Java platform☆

Lorenzo Bettini[a,*], Ferruccio Damiani[b]

[a] Dipartimento di Statistica, Informatica, Applicazioni, Università di Firenze, Italy
[b] Dipartimento di Informatica, Università di Torino, Italy

ARTICLE INFO

Article history:
Received 17 June 2015
Revised 6 July 2016
Accepted 24 July 2016
Available online xxx

Keywords:
Java
Trait
IDE
Implementation
Eclipse

ABSTRACT

Traits were proposed as a mechanism for fine-grained code reuse to overcome many limitations of class-based inheritance. A trait is a set of methods that is independent from any class hierarchy and can be flexibly used to build other traits or classes by means of a suite of composition operations. In this paper we present the new version of XTRAITJ, a trait-based programming language that features complete compatibility and interoperability with the JAVA platform. XTRAITJ is implemented in XTEXT and XBASE, and it provides a full Eclipse IDE that supports an incremental adoption of traits in existing JAVA projects. The new version of XTRAITJ allows traits to be accessed from any JAVA project or library, even if the original XTRAITJ source code is not available, since traits can be accessed in their byte-code format. This allows developers to create XTRAITJ libraries that can be provided in their binary only format. We detail the technique we used to achieve such an implementation; this technique can be reused in other languages implemented in XTEXT for the JAVA platform. We formalize our traits by means of flattening semantics and we provide some performance benchmarks that show that the runtime overhead introduced by our traits is acceptable.

1. Introduction

The problems of class-based inheritance and in particular its poor support for code reuse were emphasized by Schärli et al. (2003) (see also Ducasse et al. (2006)): both single and multiple class-based inheritance are often inappropriate as a reuse mechanism. The main reason is that classes play two competing roles: a class is both a *generator of instances* and a *unit of reuse*. To accomplish the first role, a class must provide a *complete* set of basic features, and to accomplish the second role it must provide a *minimal* set of sensibly reusable features. Schärli et al. (2003) also observed that *mixins* (Hendler, 1986; Bracha and Cook, 1990; Limberghen and Mens, 1996; Flatt et al., 1998; Bettini et al., 2003a; Ancona et al., 2003), which are subclasses parametrized over their superclasses, are not necessarily appropriate for composing units of reuse. Indeed, mixin composition is linear, because it is still based on the ordinary single inheritance operator—note that the formulation of mixins given by Bracha in JIGSAW (Bracha, 1992) does not suffer from this problem, but most of the subsequent formulations of the mixin construct do.

For the above reasons, *traits* were proposed by Schärli et al. (2003) as pure units of behavior, aiming to support fine-grained reuse. The goal of traits is to provide a flexible solution to the problems of class-based inheritance with respect to code reuse, avoiding the two traditional competing roles of classes as object generators and units of code reuse mentioned above (see also Ducasse et al., 2006; Murphy-Hill et al., 2005; Cassou et al., 2009 for discussions and examples). A trait provides a set of methods that is completely independent of any class hierarchy. The rationale is that the common methods of a set of classes can be factored into a trait. The distinguishing features of traits are that:

- Traits can be composed in an arbitrary order (leading to a class or another trait); and
- The resulting composite unit has complete control over the conflicts that may arise in the composition, and must solve these conflicts explicitly.

These features make traits simpler and more flexible than *mixins*—the "trait" construct incorporated in SCALA (Odersky, 2007) is indeed a form of mixin. The original proposal of traits (Schärli et al., 2003; Ducasse et al., 2006) was given in SQUEAK/SMALLTALK, that is, in a dynamically typed setting. Various formulations of traits in a JAVA-like statically typed setting can be found in the

☆ This work has been partially supported by: project HyVar (www.hyvar-project.eu), which has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 644298; by ICT COST Action IC1402 ARVI (www.costarvi.eu); and by Ateneo/CSP D16D15000360005 project RunVar.

* Corresponding author. Fax: +39 055 2751525.
E-mail addresses: lorenzo.bettini@unifi.it (L. Bettini), ferruccio.damiani@unito.it (F. Damiani).

http://dx.doi.org/10.1016/j.jss.2016.07.035
0164-1212/

literature (see, e.g., Quitslund, 2004; Smith and Drossopoulou, 2005; Nierstrasz et al., 2006; Bono et al., 2007; Reppy and Turon, 2007; Bono et al., 2008; Liquori and Spiwack, 2008; Bettini et al., 2013d; 2013b).

In most of the above proposals, trait composition and class-based inheritance live together. In some formulations (Smith and Drossopoulou, 2005; Nierstrasz et al., 2006; Liquori and Spiwack, 2008) trait names are types, just like class names and interface names in Java—this choice limits the reuse potential of traits, since the role of *unit of reuse* and the role of *type* are competing (see, e.g., Snyder (1986) and Cook et al. (1990)). This does not happen in *pure trait-based programming languages* (Bono et al., 2007, 2008; Bettini et al., 2013d), where:

- Class-based inheritance is not present, and
- Traits are not types.

The rationale for these choices is that pure trait-based programming languages aim to maximize the opportunity for reuse: class-based inheritance is ruled out in order to prevent programmers from writing code that might be difficult to reuse, and traits are not types to rule out the interplay between the competing roles of *unit of reuse* and *type* that would restrict traits' flexibility. These design choices do not reduce the expressivity and usability of the language. In fact, even though class-based inheritance is not present, type subsumption is still supported by Java-like interfaces. Moreover, not using trait names as types in the source program does not prevent us from analyzing each trait definition in isolation from the classes and the traits that use it. This way, it is not necessary to reanalyze a trait whenever it is used by a different class.

In previous work (Bettini and Damiani, 2013; 2014) we introduced the prototype implementation of XTraitJ, a language for pure trait-based programming interoperable with the Java type system without reducing the flexibility of traits (Bettini and Damiani, 2013), and extended XTraitJ and its implementation with full support for Java generics and Java annotations (Bettini and Damiani, 2014). Such extensions allowed us to implement generic traits, classes and generic trait methods. XTraitJ programs are compiled into Java programs, which can then be compiled with a standard Java compiler.

XTraitJ is implemented with Xtext (2015), Bettini (2013). Xtext is a *language workbench* (such as MPS (Voelter, 2011) and Spoofax (Kats and Visser, 2010)): it takes as input a grammar definition and it generates a parser, an abstract syntax tree, and a full Eclipse-based IDE. Thus, by using XText we implement not only the compiler of XTraitJ, but also its Eclipse integration. Furthermore, for the syntax of our trait method bodies, we use XBase (Efftinge et al., 2012), a reusable Java-like expression language that facilitates full interoperability with the Java type system. Since XTraitJ code can coexist with Java code, single parts of a project can be refactored to use traits, without requiring a complete rewrite of the whole code-base. This allows incremental adoption of traits in existing Java projects.

In spite of the nice integration of XTraitJ with Eclipse and Java, the implementation of XTraitJ (Bettini and Damiani, 2014) still suffered from a crucial issue that would prevent the adoption of XTraitJ in a production environment: all the XTraitJ sources have to be available in a project that uses XTraitJ. This leads to the following drawbacks:

- All XTraitJ source files have to be loaded in a XTraitJ program. While this does not prevent us from type checking traits in isolation, it still forces us to compile XTraitJ sources that are provided as libraries.
- Connected to the previous issue, trait libraries cannot be provided in a binary only format.

These are in contrast with the very concept of library. In particular, library artifacts should not be recompiled when used in a program. In industry, shipping libraries with sources might not be acceptable.

Contributions of the paper. We present a new version of XTraitJ that addresses the above limitations. We rewrote most of the implementation of XTraitJ in order to achieve full integration of traits with the Java platform, including accessibility of traits in byte-code only format. This removes the above limitations, allows trait libraries to be provided in a binary only format, and makes XTraitJ effectively usable in production. Besides the increased usability of XTraitJ, we believe that the technique that we use to achieve full integration with Java could be easily re-used in other languages that aim at such integration, using XText/XBase. To the best of our knowledge, XTraitJ is the first DSL with non trivial linguistic features that uses such technique.[1] Since XText is the de-facto standard for implementing languages in the Eclipse eco-system, and since XBase is a powerful framework for implementing languages interoperable with Java, we think that our implementation could be useful to XText/XBase users. Furthermore, we formally specify the semantics of XTraitJ by means of a flattening translation (Ducasse et al., 2006; Nierstrasz et al., 2006). The flattening translation specifies that the semantics of a class that uses traits is equivalent to the semantics of the class obtained by inlining into the body of the class the methods provided by the traits that it uses. Finally, we evaluate XTraitJ in terms of the overhead introduced by method forwarding, which is used in the generated Java code to implement traits. The performance tests show that the overhead introduced is an acceptable tradeoff with respect to the code reuse of traits. We also evaluate the performance of the compiler of this new version of XTraitJ, which is improved with respect to the previous versions.

A preliminary version of some of the material presented in this paper appeared in Bettini and Damiani (2013, 2014). The specification of the semantics of XTraitJ (Section 3) and the technique to achieve binary level accessibility of traits (Section 4) are completely new, and both the description of the implementation (Section 5) and the evaluation of the achieved benefits (Section 6) have been revised and extended to reflect the new implementation, to provide more details, and (in Section 6.2) to illustrate performance results.

The implementation is available as an open source project and ready-to-use update site at http://xtraitj.-sf.net. We also provide pre-configured Eclipse distributions with XTraitJ installed, for several architectures. Moreover, XTraitJ programs can be processed with typical Java build tools, like Maven and Gradle, by relying on the Maven integration provided in recent versions of XText (Oehme, 2015). XTraitJ has been developed with *Test Driven Development* technologies, with almost 100% code coverage, using *Continuous Integration* systems (*Jenkins* and *Travis-CI*) and code quality tools, such as *SonarQube*.

Organization of the paper. Section 2 illustrates the syntax and, informally, the semantics of the XTraitJ programming language through examples. Section 3 formally specifies the semantics of XTraitJ by means of a translation that compiles traits away. Section 4 describes how XTraitJ has been fully integrated with Java and how binary only accessibility of traits has been achieved.

[1] Indeed, during the development of this new version we found a few issues with some internals of XBase, in particular, related to the implementation of generics under certain circumstances, which had not been considered. In our implementation, we solved them by customizing many parts of the XBase type system concerning generics, but we are also working on fixing these issues in the XBase code base as well—see https://bugs.eclipse.org/bugs/show_bug.cgi?id=468174.

Please cite this article as: L. Bettini, F. Damiani, XTraitJ: Traits for the Java platform, The Journal of Systems and Software (2016), http://dx.doi.org/10.1016/j.jss.2016.07.035