Java Decaffeinated: experiences building a programming language from components

Linda Farragher and Simon Dobson

Department of Computer Science, Trinity College, Dublin, Ireland
simon.dobson@cs.tcd.ie

Abstract. Most modern programming languages are complex and feature rich. Whilst this is (sometimes) an advantage for industrial-strength applications, it complicates both language teaching and language research. We describe our experiences in the design of a reduced sub-set of the Java language and its implementation using the Vanilla language development framework. We argue that Vanilla's component-based approach allows the language's feature set to be varied quickly and simply compared with other development approaches.

Introduction

Modern programming language design are complex and feature rich. The extra features often provide important abstractions which facilitate the development of complex industrial systems. This richness does mean, however, that feature sets become entangled. This poses problems for two distinct communities of users.

Language researchers often want to study the impact of new features on languages. Experimentation with industrial-strength languages provides a realistic framework for such experiments, allowing the new feature to be studied within a well-tried language and avoiding many of the criticisms levelled at research using purpose-designed languages. However, feature entanglement can make it difficult to study the exact behaviour and ramifications of the feature under test. Moreover industrial-strength languages generally have industrial-strength compilers whose internal organisations and optimisation strategies are often not amenable to incremental experimentation.

Programmers learning a language, and especially those learning a first language, are also disadvantaged by this complexity. Many universities have abandoned teaching programming through Pascal and Scheme – simple systems with many pedagogical advantages – in favour of the industrial relevance of C++ and Java. Whatever we may think of this strategy from an educational perspective, it is undoubtedly the case that the latter languages have steeper learning curves due to their enhanced feature sets. One cannot, for example, learn Java's basic statements without wrapping them in an object-oriented harness. This can be a significant barrier to learning.

In both cases the problems of feature-richness and -entanglement may be addressed by using a compositional language design system, in which language features may be added and changed easily.

In this paper we describe our experiences in constructing a small, "decaffeinated" version of Java using the Vanilla language construction system. We had two main goals:

- to develop a small "kernel" Java suitable for language research; and
- to investigate the characteristics of component-based language construction by developing some small variations on this base language.

As a side effect we aimed to develop a family of Java sub-sets suitable for teaching the language (and programming in general), which would allow features to be introduced only when appropriate.

We begin with an overview of our development framework. We then describe the design process and feature sets of our language and discuss their implementation. We evaluate the results in terms of the performance of the resulting languages, their speed of development and their ease of modification – the last involving the development and integration of two variations to the standard feature set. We also compare the language against other reduced Java sub-sets from the literature, before summarising our experiences

An overview of Vanilla

Programming language development has traditionally involved a large amount of initial effort to develop simple parsers, type checkers and interpreters or code generators for the language. The Vanilla language framework[2] (`http://www.vanilla.ie`) begins from two observations: that the corpus of programming languages has a set of features which at the abstract type and semantic levels show little variation; and that these features are in many cases independent of one another, so that a compositional approach is justified as a way of reducing development costs and complexity.

Vanilla provides a component-based framework to the development and integration of language features into interpreters (figure 1). If we consider type checking as an example, the framework defines the key properties, operations and algorithms of type checking (free variables, substitution, sub-typing *etc*) while the components provide the realisations of (parts of) these operations. Essentially the framework defines *what is the same* for all type checkers while the components define *what is different* between individual languages. Component re-use occurs when languages exhibit substantial commonality in their typing. The same comments apply to interpretation and (to a lesser extent) parsing. A programming language is constructed by populating the sub-systems with the appropriate sets of components.

Each language feature is implemented as a set of components – collectively referred to as a *pod* – which collects together the abstract syntax, types, values, type rules, interpretive semantics and (optionally) conmcrete syntax for the feature. Any of these components may be varied, so providing (for example) lazily-evaluated functions involve replacing the interpreter in the functions pod, without affecting its typing rules. A

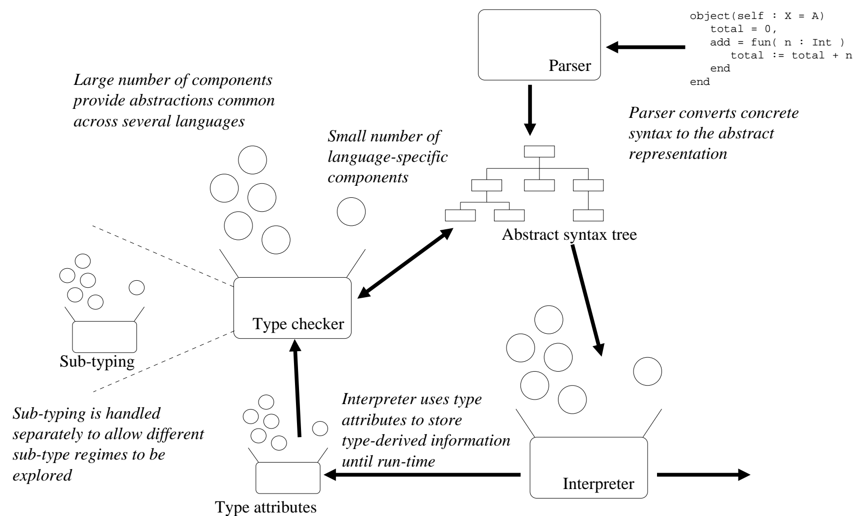

Fig. 1. Vanilla constructs languages by combining component fragments

simple system of parser combinators permits the construction of full parsers from parser fragments; alternatively a traditional "all in one" parser may be used. A language is built by composing the desired feature pods with an appropriate parser.

Vanilla interpreters are completely un-optimised in favour of semantic and implementational clarity, allowing the internal structures of language features to be expressed cleanly. The full Vanilla system provides a set of components implementing a wide variety of features commonly encountered in experimental and mainstream programming languages.

Java from components

Few modern languages are defined from whole cloth: it is generally possible to identify a set of orthogonal features which may be defined and implemented independently. The process of *language design* then collapses to a process of *feature design and composition*, with the benefits that features may be added, removed and varied largely independently. Although perfect orthogonality is rare, this perspective radically reduces the overheads and allows simple variation.

Examining Java

The design of Java Decaffeinated entailed three stages:

1. deciding on the components we wished for the language;

2. identifying those components already contained within the Vanilla standard set; and
3. implementing any omissions.

The criteria for deciding which components to include in the language were that only constructs with simple, obvious and significant contributions would be included. The result should be a strict sub-set of the Java language with “awkward” features (for implementors or users) removed.

Feature	Source
Arithmetic	Core pod
Simple conditionals	Core pod, conditionals pod
Complex conditionals	New pod needed to provide switch statements
Top-tested loops	Loops pod
Bottom-tested loops	New pod moving the test down
Bounded loops	Loops pod
Classes and objects	Records pod plus a new object model
Static variables	Built into the object model
Functions	Functions pod
Input/output	Simple I/O pod (see below)

Table 1. Java has a set of orthogonal components, many re-used from other systems

Some features not necessary for Java Decaffeinated presented themselves immediately. Exceptions for example were seen as unnecessary, and very unsuitable for a beginner. Nested classes and interfaces were seen similarly, as they provide only minimal extra functionality for small-scale applications.

Some features were less obvious. Protection of variables and methods is needed, however for the beginner student should it be necessary to enforce explicitly stating the protection at every declaration? Pedegogacally it seems correct to enforce the explicit declaration of protection at every declaration, although this could become tedious for the student after the lesson has been learned.

The structure of files in the language has to be considered also. Languages which have been generated using Vanilla typically have the “mainline” (in the style of Pascal) which is the top-level unit of execution. This has the advantage that (for researchers) all initialisation can be centralised and (for learners) there is no need to understand objects and `main()` routines before beginning programming.

The resulting language (table 1) is not semantically very different from Java, differing only in what it omits. Classes may be declared in the same way as in Java, as can methods, and variables. Recusive functions may be declared, as can recursive objects. It is possible to convert a program written in Java to Java Decaffeinated fairly simply (as long as it does not import any packages).

The lack of package imports of course makes the entire standard library inaccessible – including any input/output functions. This was addressed by explicitly recognising

some simple actions (for example `System.out.println()`) in the parser and converting them to simplified internal forms (in this case a call to Vanilla's simple I/O pod). This shows that a single feature (in this case I/O) can be presented in the language in a number of ways.

Development

The development of the language proceeded as a simple implementation within the usual Vanilla style. Existing features – the majority of the language – were re-used directly, omitting only their concrete syntax components. New features were developed in isolation or by building on existing functionality. For the object model – the only major development in the project – the steps were:

1. Define the abstract syntax of classes and objects including classes, inheritance, methods, instance variables, new() expressions etc. Method definitions re-used the functions defined in the function pod under the usual encoding of methods (without self types) as closures.
2. Define the types and values. Objects were defined for class types, the resulting object types, and object instance values.
3. Define the type and sub-type rules. This was simply a matter of re-writing the existing Java type rules into Java syntax within Vanilla. Since the rules are available from the literature, this is largely a transcription exercise.
4. Similarly define the interpretation rules.

One simplification was to represent recursive object types – for example a class A containing a method `public A oneOfMe()` – using explicit manipulations of the type environments. The "correct" way, using μ-recursive types, is simpler for a type theorist but perhaps less than intuitive for many practical language experimenters.

The complete language was provided with a single overall parser converting concrete into abstract syntax. While this makes variation more complex (see below), it is again more intuitive for the first-time designer.

Additions and variations

Java Decaffeinated can, because of its component oriented structure grow and shrink as a language with very little effort. The actual language is constructed by specifying its component pods, with the addition or removal of a particular pod being achieved by adding or deleting an entry. Although the language's concrete syntax may need to be changed this is a relatively simple operation, especially in the presence of parser combinators.

As an experiment we explored a number of variations to Java, including a Python-style `forall` loop written in an afternoon and used as a drop-in replacement for (or

indeed addition to) the standard `for` construct. This is a radical simplification over more monolithic language construction involving no change to any other code in the system (other than the parser).

A further generalisation was to allow methods on a class to be extracted into variables and called without losing their self bindings – the first-class methods found in object calculi[1].

For language teachers, perhaps a more important capability is removing features until they have been taught. We defined a family of Java sub-sets (arithmetic only, arithmetic plus assignment, simple looping, simple objects, objects with inheritance, *etc*) each introducing new concepts in a controlled way.

One useful feature of the system is that we may eliminate a feature from the user while retaining it for the language itself. One may, for example, hide functions by removing the concrete syntax which allows them to be introduced, while retaining the ability to use functions in the implementation of other constructs if desired.

Evaluation

Component-based language development

During the course of the development the advantages of a component-based language quickly became apparent. The reuse of Vanilla pods greatly speeds-up development of a language. To the "beginning" language developer it is a great advantage to be able to see and modify predefined components, as well as seeing a language evolve gradually. A monolithic parser, type checker or interpreter for a language could never seriously be considered accessible to a beginner. However with the code occurring in small parts it is easy to see how each particular component works.

A language built from components also has the obvious advantage of code readablility and reuse. This allows for rapid development of test languages and their easy modification. Having more than one version of a language is simple to achieve, with a minimum of waste of space, since only some of the components will have changed, and the inclusion of a newer component is simply a matter of changing thre language definition file.

Performance – of programs and developers

Java Decaffeinated is interpreted using an interpreter written in Java, which runs on top of the Java Virtual Machine – itself a notoriously inefficient interpreter. It might be expected that this cascade of inefficiencies would render the system unusable. In practice this turned out not to be the case: performance, while less than Java by an order of magnitude or more, is perfectly acceptable for small studies or exercises. In any case, as Every[4] has succinctly put it,

[1] This variation – and why it is omitted from the Java language definition – is discussed in [6].

> 10 or 20 years ago, when computers were literally hunderds or thousands of times slower than today, we could not sacrifice performance for convenience – but the reality of today may have changed things. Don't forget that computers are doubling performance every 18 months, and programming costs are still increasing. Interpreted programming will slowly take over in development.

...to which we would only add "and in experimental language development too". The speed of development is (and will remain) the dominant factor in language research, and a component-based approach both radically reduces this development time and applies that time more productively. Using components also facilitates the generation of a language by more than one person, again reducing development times.

Java sub-sets

The existence of a set of Java sub-sets – recognisably Java but simpler and easier to experiment with – is obviously a major boon for researchers wanting to explore the integration of new features into a mainstream language without the pain of adapting a full compiler.

A huge number of Java sub-sets and extensions have been described in the literature – indeed, recent conference proceedings would suggest that exploring the definition of new concepts into Java is becoming *de rigeur* for a language research paper! The variations range from theoretical minimal sub-sets such as Igarashi, Pierce and Wadler's "Featherweight Java"[5], to the addition of new features such as generic types and mixins[1]. Between these extremes are languages intended for use as teaching and research platforms such as JJ[3], which shares many characteristics with our own work.

JJ was designed as a language which would be used to teach students Java. The syntax of JJ is quite different to standard Java, with much emphasis placed on meaningful error messages, saying what the compiler thinks went wrong rather than what the compiler expected. JJ also supports design by contract and enforces functions having no side effects.

JJ has not been formally tested, but it is easy to see it is very suitable for a person wishing to learn programming properly the first time. The support for design by contract is an excellent feature, previously only fully implemented in Eiffel. Its premise is similar to Java Decaffeinated - however the results have been quite different. Java Decaffeinated has syntax identical to Java; JJ's is more similar to Eiffel. Java Decaffeinated differs from Java only in what it does not contain; JJ differs from Java in a very many respects. We also believe that design by contract could be added – as a stand-alone pod – to Java Decaffeinated with minimal effort.

Vanilla

The Vanilla system contains a sizeable fraction of the common language features predefined. The creation of Java Decaffeinated entailed simply deciding which pods to

reuse and then writing pods for whatever was not defined already in Vanilla. After deciding which parts of Java to use it was noted that the only parts of the language which needed to be defined were the bottom-tested (`do...while`) loop, the `switch` statement and the class and object structures. Deciding on which parts of the language would be placed in which pod was simply an exercise in seeing which parts of the language could be considered together. The `do-while` loop could be integrated into the loops pod, and the `switch` statement could be integrated with the conditionals pod, or have its own pod. Vanilla's standard – and for our purposes overly complex – object model separates classes and objects into their own pods, while for Java Decaffeinated it was felt more appropriate to develop a single unified Java-style object model pod.

The use of components not only speeds up development (in that developers using Vanilla do not need to write common components), but also makes the full code of examples of components accessible to the developer. Our experience suggests that Vanilla's overall effect is to bring language development within reach of non-specialist users and researchers – including final-year undergraduates with little or no experience in language technology. By lowering the learning curve, exposing the individual constructions and separating concerns, it seems to provide a good test-bed for both learning and practicing the crafts of language design and research.

Conclusion

We have described using language components to implement a family of Java sub-sets. The component-based approach allowed us to vary the feature sets of the resulting languages very easily, and facilitated the introduction of new or variant features into the languages for exploratory purposes. We illustrated this through (with a research perspective) the development of new looping constructs and object models, and (with a teaching perspective) the deployment of features incrementally into a simplified teaching language.

We believe that this work has two main contributions.

Firstly, the results of our work show that the component-based approach to language development provided by Vanilla has significant advantages to the language developer. While the implementations are in no way optimised, their performance is acceptable for small-scale experiments. Furthermore they are almost trivial to vary and extend, making it extremely easy to experiment with new features or to remove features for teaching purposes.

Secondly, the languages developed have exposed the independent features within Java that may profitably be varied for research or teaching purposes. This has shown that Vanilla's premise that most language features are orthogonal holds for Java at least.

Acknowledgements

This work was conducted as part of the first author's undergraduate degree project at Trinity College.

References

1. Davide Ancona, Giovanni Lagorio, and Elena Zucca. Jam – a smooth extension of Java with mixins. In Elisa Bertino, editor, *ECOOP 2000 – object-oriented programming*, volume 1850 of *LNCS*. Springer Verlag, 2000.
2. Simon Dobson, Paddy Nixon, Vincent Wade, Sotirios Terzis, and John Fuller. Vanilla: an open language framework. In Krzysztof Czarnecki and Ulrich Eisenecker, editors, *Generative and component-based software engineering*, LNCS. Springer-Verlag, 1999.
3. David Epstein, Joseph Kiniry, and John Motil. What is JJ?, 2000.
4. David Every. What is Java? http://www.mackido.com/Dojo/Java/html.
5. Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Feathewrweight Java: a minimal core calculus for GJ.
6. Sun Microsystems. About Microsoft's "delegates". http://java.sun.com/docs/white/delegates.html.