A PLEA FOR PLUGGABLE PROGRAMMING LANGUAGE FEATURES

Bernhard G. Humm
Darmstadt University of Applied Sciences, Darmstadt, Germany

Ralf S. Engelschall
Capgemini sd&m Research, Munich, Germany

Keywords: Programming Language Features, Aspects, Flexibility, Evolutionary Prototyping, Plug-in.

Abstract: Current programming languages are inflexible regarding their use of language features like typing, access control, contracts, etc. In some languages, the programmer is forced to use them, in others he may not. This article pleads for pluggable programming language features, a concept that allows greater flexibility for application programmers without losing control over the use of those features.

1 INTRODUCTION

Flexibility is one of the most basic and important design goals in software engineering (Ghezzi et al., 2002) (Sommerville, 2004). Flexibility allows for adaptation of applications to different and possibly varying needs. This not only applies to the resulting application, but also to the tools for creating them.

However, when analyzing current programming languages and their features concerning their flexibility of use, the result is rather disappointing. Consider just the following two examples.

- Current mainstream programming languages like C/C++, Java, and C# are all statically typed. Static typing is mandatory there and the programmer has no flexibility as to omit type specifications where sensible. Contrarily, dynamic languages like Smalltalk, Scheme, Python and PHP are all dynamically typed and the programmer has no option whatsoever to explicitly specify type declarations statically where sensible.
- Access control in Java is mandatory. For all classes, interfaces and members, accessibility must be declared (public, protected, private or package local as default). The programmer has no option of omitting access control specifications where sensible. On the other hand, declaring access control for packages is not possible at all in Java. Also, in dynamic languages like Smalltalk, explicit access control on class and method level is not possible. The programmer has no option of specifying access control where sensible.

This inflexibility causes a number of problems:

Coding Overhead: In many industrial software projects, the implementation technology is predefined, e.g., Java. Implementing application parts like scripts, code generators, or data migration routines — for which scripting languages are most suitable — in Java result in unnecessary coding overhead due to mandatory language features, whose use is not necessary in this context.

Poor Quality: Contrarily, implementing critical application parts in a dynamic language like Smalltalk may reduce quality — in this case safety — due to missing compile-time checks (Meijer and Drayton, 2005).

One might argue that in such a case, the language choice is simply wrong and an industrial strength language like Java should have been used. This leads us to the next problem.

Incremental Development Impeded: In many project situations it is sensible to develop software incrementally (Martin, 2002), e.g., with evolutionary prototyping (Floyd, 1984; Berger et al., 2004; Gordon and Bieman, 1993; Hekmatpour, 1987). This means that an application or a part of it is quickly prototyped first and then,

G. Humm B. and S. Engelschall R. (2010).
A PLEA FOR PLUGGABLE PROGRAMMING LANGUAGE FEATURES.
In *Proceedings of the Fifth International Conference on Evaluation of Novel Approaches to Software Engineering*, pages 5-11
DOI: 10.5220/0002925600050011

incrementally, the code quality is enhanced. If the target application is critical and an industry-scale language like Java is chosen, then quick prototyping is impeded due to many mandatory language features.

The overall picture today is that programming languages define a fixed set of features that are to be used. The programmer has no flexibility as to use less strict features where acceptable or even to specify more advanced features, e.g., access control on package level in Java, where necessary.

To alleviate those problems, we plead for a concept that we call "pluggable programming language features" and argue from the application programmer's point of view, i.e., from the view of the user of a programming language. In Sect. 2 we present the concept. Sect. 3 and 4 present a sample application and a research prototype of the concept. Sect. 5 discusses the results. Sect. 6 concludes this article.

2 PLUGGABLE PROGRAMMING LANGUAGE FEATURES

Before introducing the concept of pluggable programming language features, we need to distinguish two kinds of programming language features.

Core programming language features are essential for implementing applications at all. Examples are objects, classes, operations, variables, and control constructs like loops.

Additional programming language features specify aspects of core language features. Examples are access control for classes, type declaration of variables, and pre- and postconditions of operations. Additional language features are not essential in the sense that it is possible to implement applications without using additional programming language features.

Only additional programming language features may be pluggable. For a programming language to adhere to the concept of pluggable programming language features we postulate the following requirements.

Optional Language Features: The language must allow for implementing applications without using any additional programming language features at all. In particular, static typing must not be mandatory.

Independent Language Features: The language must allow for specifying additional programming language features independently and to check for their conformance at an adequate point in time. In particular, static type checking must be possible.

Extensible Language Features: The programming language must be extensible to allow for the implementation of new additional language features.

Language Feature Configuration: The programming language must allow for configuring the use of additional language features in an application or parts of them. The use may be enabled mandatory or optionally, or disabled. Enabled language features will be checked, e.g., by the compiler.

In total, the concept allows for plugging in additional language features, either pre-defined ones or new ones. Arbitrary use of language features is avoided via language feature configuration.

We see two major use cases for pluggable programming language features.

Customizing Features according to Requirements: Pluggable programming language features allows system architects to customize a programming language with respect to the quality requirements of an application to be developed. Depending on the criticality, more language features may be plugged in — even additional ones that have not been pre-defined in the programming language. The configuration enforces the use of those language features by the programmers.

Customizing Features per Development Stage: Pluggable programming language features allows for efficient incremental software development, particularly with evolutionary prototyping. In an early stage of development, additional language features may be omitted completely by programmers. This allows for rapid prototyping. Such a rapid prototype may be used to get user feedback quickly, as well as checking for architectural integrity of the application. Gradually, the code quality of the application may be enhanced by plugging in additional language features. Language feature configuration gives control over this process. For different stages in the development process, e.g., "Proof of Concept", "Alpha Release", "Beta Release", and "Final Product", specific language features may be enforced.

We now demonstrate the concept via a research prototype and its use via a sample application that focuses on the second use case.

3 EXAMPLE DOMAIN: CUSTOMER MANAGEMENT COMPONENT INTERFACES

3.1 Customer Management

We demonstrate pluggable programming language features exemplary via interfaces for a customer management component of a business information system.

Our example is the `create-customer` operation with parameters `name`, `address`, and `date-of-birth` for adding a new customer object to a customer management data store. The example seems trivial but may be quite complex in practice. For instance, `address` may be checked for validity syntactically as well as semantically via city map data.

3.2 Interface Specification Aspects

An interface is specified by a *name* and its *operations*. An operation's signature is minimally specified by its *name* and the *parameter names*. Additionally, the following aspects may be specified:

- Access control, e.g., public, private
- Parameter mode: input, output, input/output
- Parameter obligation: mandatory, optional (e.g., expressed by `null` values)
- Parameter types, e.g., primitive types like `Integer` and complex types like `Customer`
- Type restrictions, e.g., only positive `Integer` values for a bank transfer. Note: Type restrictions may be implemented as separate types, e.g., `Positive-Integer`
- Pre- and postconditions: constraints before and after operation execution, respectively — e.g., `date-of-birth < now`. Note: parameter modes, obligations, types and type restrictions may all be specified as pre- and postconditions
- Exceptions: specification of exceptional situations that are externally visible, e.g., duplicate customer
- Side effects, e.g., read-only, modifying
- Semantics documentation: specification of the operation's behavior and documentation of its parameters, usually informally in prose. Note: pre- and postconditions are part of the semantic specification, too
- Non-functional characteristics: specifying (formally or informally) performance and other non-functional characteristics

Quality Level	Operation Names	Parameter Names	Parameter Modes	Parameter Obligation	Parameter Types	Visibility	Exceptions	Type Restrictions	Pre- / Postconditions	Side Effects	Semantics and Parameter Documentation	Non-functional Characteristics
Proof of Concept	x	x										
Alpha Release	x	x	x	x	x							
Beta Release	x	x	x	x	x	x	x	x	x			
Final Product	x	x	x	x	x	x	x	x	x	x	x	x

Figure 1: Language Feature Configuration.

Generally speaking, for a production-grade business information system, the more complete the interface specification, i.e., the more intrinsic information is specified explicitly, the better.

3.3 Language Feature Configuration

For instance, consider the development stages "Proof of Concept", "Alpha Release", "Beta Release", and "Final Product". Then, language features may be assigned to the development stages as shown in Fig. 1.

In the following section, we describe language features for specifying some of the interface specification aspects exemplary.

4 LANGUAGE FEATURES FOR INTERFACE SPECIFICATION

4.1 Research Prototype in Lisp

We have chosen Lisp[1] (McCarthy, 1960) as the implementation language for our research prototype to demonstrate pluggable programming language features. Lisp is dynamically typed but contains a powerful type system as well as a built-in macro processor for implementing new language features.

We have implemented a custom macro `define-function` that extends the basic built-in `defun` macro for defining an operation. `define-function` is a real extension of `defun` in the sense that it accepts all declarations of `defun` but, additionally, optional aspects.

In the next sections, we show some of the language features exemplary step by step by means of the example `create-customer`, thereby incrementally enhancing code quality by the use of pluggable language features.

4.2 Operation and Parameter Naming

In the simplest form (development stage "Proof of Concept"), the *name of an operation* and its *param-*

[1]More specifically: Allegro Common Lisp, a professional implementation of the ANSI Common Lisp standard

eter names are specified only.

```
(define-function create-customer
    (name address date-of-birth))
```

This expression declares the operation `create-customer` with input parameters `name`, `address`, and `date-of-birth`. No additional language features need to be specified at this stage.

4.3 Parameter Typing

The *type of an input parameter* (necessary for development stage "Alpha Release") is specified via the keyword `:type` in a list per parameter. The *type of the operation result* (out parameter) is specified via the keyword `:result-type` in an options list following the parameter list.

```
(define-function create-customer
 ((name          :type Structured-Name)
  (address       :type Structured-Address)
  (date-of-birth :type Date))
 (:result-type Customer))
```

The parameter `name` is of type `Structured-Name`, the parameter `address` of type `Structured-Address`, etc.

4.4 Pre- and Postconditions

Pre- and postconditions (necessary for development stages "Beta Release" and "Final Product") are specified via the keywords `:pre` and `:post` in the options list, followed by a Lisp boolean expression that can be evaluated at run-time.

```
(define-function create-customer
 ((name          :type Structured-Name)
  (address       :type Structured-Address)
  (date-of-birth :type Date))
 (:result-type   Customer
  :pre           (is-valid? address)
  :pre           (lies-in-past?
                  date-of-birth)
  :pre           "No duplicate of
                 previously created
                 customer"
  :post          (get-id result))
```

The first precondition is satisfied if the operation `is-valid?` with the actual parameter `address` evaluates to true (non-`nil`). This checks for valid addresses. `lies-in-past?` checks whether the birth date is plausible. The third pre-condition regarding duplicate checking is treated as an informal comment. The postcondition specifies that the resulting `Customer` object contains a non-`nil` identifier.

4.5 Documentation of Semantics

To document the semantics of the operation and the input and output parameters, the keywords `:documentation` and `:result-documentation` are used in the options list and the parameters lists.

```
(define-function create-customer
 ((name          :type Structured-Name
                 :documentation "Customer
                  name consists of ...")
  (address       :type Structured-Address
                 :documentation "Postal
                  address consists of ...")
  (date-of-birth :type Date
                 :documentation "Customer
                  birth date"))
 (:result-type   Customer
                 :result-documentation
                  "New Customer object"
  :pre           (is-valid? address)
  :pre           (lies-in-past? date-of-birth)
  :pre           "No duplicate of previously
                 created customer object"
  :post          (get-id result)
                 :documentation "Creates
                  a new Customer object"))
```

4.6 Additional Language Features

Analogously, we have implemented the following additional language features: access control, modes, obligations, exception specification, and non-functional characteristics. None of those are natively provided in the core language feature set of Lisp. With our extensions, application programmers may optionally and independently use all of those additional programming language features.

4.7 Conformance Checking

It is not enough to provide language features for specifying interface aspects — the specification conformance has to be checked, too. Therefore, we have implemented the macro `define-function` to generate conformance checks. Type specifications are, if possible, checked at compile time. Pre- and post-conditions are checked at runtime. All specification aspects are compiled into the built-in function documentation of Lisp.

But checking the specified aspects is only one kind of conformance check. The macro also checks the conformance of the application code with the language feature configuration during compilation. In case of violations, warnings are being generated. For example, static parameter type checking is enforced from development stage "Beta Release" on as in any statically typed language like Java.

Note: not all all conformance checks can be fully automated. For example, a conformance checker can

not decide whether or not there are meaningful preconditions for an operation.

5 DISCUSSION

5.1 Evaluation

This article is a *plea* for pluggable programming language features. We cannot empirically prove the usefulness of the approach. However, our confidence stems from our long-time experience in developing large-scale business information systems and the promising results of our research prototype and sample implementation. Furthermore, we qualitatively justify our approach by evaluating it and the sample implementation against the problems identified in Sect. 1.

Coding Overhead: Pluggable programming language features allow to reduce coding overhead by omitting unnecessary language features in certain application contexts, like scripts, code generators, or data migration routines. A language switch towards a scripting language is not necessary since the programming language itself offers the necessary flexibility.

Poor Quality: Pluggable programming language features allow critical applications to be implemented in a strict manner thus improving code quality. Not only language features common in industrial-strength programming languages can be used. Additionally, even more strict language features may be plugged in. Examples are pre- and postconditions or advanced access control which extends towards packages and components.

Incremental Development Impeded: Pluggable programming language features particularly boost incremental application development, e.g., with evolutionary prototyping. An application or a part of it may be quickly prototyped first and then, incrementally, the code quality may be enhanced. Language feature configuration prevents arbitrary use of language features at the programmers' goodwill. Certain quality levels at certain development steps can be enforced.

5.2 Language Support Today

Current programming languages, both in industry and academia, support pluggable programming language features very poorly. Today, there is a strict demarcation of languages focusing either on rapid application development (RAD) or on industry scale development.

Industry Scale Languages, Statically Typed: Languages like Java and C# are currently in mainstream use for developing large-scale, high-quality applications. They are all statically typed and are not well suitable for rapid application development (RAD). Some language features like visibility (public, private) are, indeed, optional. However, more advanced quality features like pre- and postconditions are not directly provided and can only be indirectly injected at the byte-code level.

RAD Supporting Languages, Dynamically Typed: RAD supporting languages like Smalltalk, Perl, Python, Ruby, Groovy, Scala and F#, conversely, are currently not in mainstream use for developing industry-scale applications. They are either used for throw-away prototyping or for developing special-purpose applications like web sites. Most of them are dynamically typed and do not allow for static typing. Quality features may be added as we have shown with Lisp macros in this article but this is not commonly done.

Hybrid Typing Languages: A few languages like VisualBasic, Perl 6 and Lisp (partially) exist that allow for static as well as dynamic typing. They also allow, in limited ways, for extending the language by new quality features. Neither is in mainstream use. However, with C# 4.0, now in beta release, the first mainstream programming language will incorporate dynamic typing optionally — one important step towards pluggable programming language features.

5.3 Related Work

In their article "Static Typing Where Possible, Dynamic Typing When Needed: The End of the Cold War Between Programming Languages" (Meijer and Drayton, 2005), Meijer and Drayton plead for typing as a pluggable programming language feature. Though typing is only one of many features of programming languages, the difference between static and dynamic typing are sometimes exaggerated as "language war". We fully agree with Meijer and Drayton — language wars are not at all necessary.

We extend their point of view in three ways. Firstly, we regard typing as one language feature only. Although most important, it represents only one point in a whole spectrum between flexible prototype development and extremely strict development of critical applications. Secondly, we allow for true plugging

of programming language features in the sense that new features may be added to the language. Finally, we add the concept of language feature configuration which gives control over the use of language features.

The comparison with Bracha's article "Pluggable Type Systems" (Bracha, 2004) is similar. His implementation of Strongtalk (Bracha and Griswold, 1993) on the basis of Smalltalk is an example of a pluggable language feature, namely typing.

With the Scala programming language (Odersky, 2004), Odersky targets at scalability and flexibility, too. He tries to reduce the set of language features as much as possible and, instead, provides features in libraries. However, on the level of additional language features like typing and access control, Scala is still inflexible. Scala uses type inference to ease the programmer from the burden of specifying types unnecessarily often but is still statically typed at any time.

Finally, we see a strong relationship between Aspect-Oriented Programming (AOP) (Kiczales et al., 1997) and pluggable programming language features. While not inherently tied to it, AOP in practice is used for implementing functionality for the end-user like, e.g., logging. On the other hand, pluggable programming language features target the application programmer by addressing internal code quality like maintainability, stability, reliability, etc. Hence, our approach follows the tradition of AOP, but with a different focus.

6 CONCLUSIONS AND FUTURE WORK

In this article, we plead for pluggable programming language features, a concept that adds flexibility to programming languages. It allows for using or omitting programming language features with full control via programming feature configuration.

We demonstrated the concept via a research prototype and a sample application in Lisp. While the concept has obvious benefits, it is not well supported by current programming languages. Furthermore, we agree with Meijer and Drayton, who identify a "huge cultural gap" between the communities of statically and dynamically typed languages (Meijer and Drayton, 2005).

However, we see a new trend towards dynamic programming languages in the last decade that are implemented on top of mainstream platforms. Examples are implementations of Python, Ruby, Groovy, and Scala on the Java Platform or F# and C# 4.0 on the .NET platform. Furthermore, there are a number of Lisp implementations on the Java platform, e.g., ABCL, Clojure, Jatha, and CLForJava.

This may allow for pluggable language features to eventually break through — for two reasons. Firstly, the technical integration of languages of different styles eases the implementation of pluggable language features. Optional typing in C# 4.0 is a perfect example for that. Secondly, a growing community of programmers who are proficient in both language styles will help closing the cultural gap. Additionally, if mainstream languages already had real support for pluggable programming language features, the necessity for numerous special languages would be reduced.

Our plea for pluggable programming language features is from the application programmers' point of view. We see future work in the following areas. Pluggable programming language features need to be implemented in programming languages on top of mainstream platforms. Integrated development environments need to support pluggable programming language features, particularly their configuration. Experience needs to be gained in industrial projects of different sizes. Finally, a development methodology that best utilizes pluggable programming language features needs to be developed.

REFERENCES

Berger, H., Beynon-Davies, P., and Cleary, P. (2004). The Utility of a Rapid Application Development (RAD) approach for a large complex Information Systems Development. In *Proceedings of the 13th European Conference on Information Systems (ECIS 2004), Turku, Finland*.

Bracha, G. (2004). Pluggable type systems. In *OOPSLA Workshop on Revival of Dynamic Languages, 2004*.

Bracha, G. and Griswold, D. (1993). Strongtalk: Typechecking Smalltalk in a production environment. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'93)*.

Floyd, C. (1984). A systematic look at prototyping. *Approaches to Prototyping*, pages 1–18.

Ghezzi, C., Jazayeri, M., and Mandrioli, D. (2002). *Fundamentals of Software Engineering*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

Gordon, V. S. and Bieman, J. M. (1993). Reported Effects of Rapid Prototyping on Industrial Software Quality.

Hekmatpour, S. (1987). Experience with evolutionary prototyping in a large software project. *SIGSOFT Softw. Eng. Notes*, 12(1):38–41.

Kiczales, G., Lamping, J., Mendhekar, Videira Lopes, C., Loingtier, J.-M., and Irwin, J. (1997). Aspect-Oriented Programming. In *Proceedings of the Eu-*

ropean Conference on Object-Oriented Programming (ECOOP'97). Springer-Verlag LNCS 1241.

Martin, R. C. (2002). *Agile Software Development, Principles, Patterns, and Practices*. Prentice Hall.

McCarthy, J. (1960). Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Communications of the ACM*, 3(4):184–195.

Meijer, E. and Drayton, P. (2005). Static Typing Where Possible, Dynamic Typing When Needed. In *Workshop on Revival of Dynamic Languages*.

Odersky, M. (2004). An Overview of the Scala Programming Language: EPFL Technical Report IC/2004/64.

Sommerville, I. (2004). *Software Engineering, 7th Edition (International Computer Science Series)*. Addison Wesley.