# Pluggable Programming Language Features for Incremental Code Quality Enhancement

Bernhard G. Humm and Ralf S. Engelschall

Darmstadt University of Applied Sciences, Darmstadt, Germany
Capgemini, CSD Research, Munich, Germany
bernhard.humm@h-da.de, ralf.engelschall@capgemini.com

**Abstract.** Evolutionary prototyping is an incremental software development method in which a proof of concept is, step by step, extended towards the final product. This article pleads for a programming approach termed “incremental code quality enhancement” when developing software incrementally. However, current programming languages are not well suited for incremental code quality enhancement. They 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 introduces pluggable programming language features, a concept that allows greater flexibility for application programmers without losing control over the use of those features. The approach is demonstrated exemplary by interface specifications for a business information system.

**Keywords:** Programming language features, Aspects, Flexibility, Evolutionary prototyping, Plug-in.

## 1 Introduction

Flexibility is one of the most basic and important design goals in software engineering [8] [21]. 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

L.A. Maciaszek and P. Loucopoulos (Eds.): ENASE 2010, CCIS 230, pp. 1–15, 2011.
© Springer-Verlag Berlin Heidelberg 2011

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 pre-defined, 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 [18].

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 [16], e.g., with evolutionary prototyping [6,2,9,10]. This means that an application or a part of it is quickly prototyped first and then, incrementally, the code quality is being 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 where necessary, e.g., access control on package level in Java.

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. Pluggable programming language features are particularly useful for a development method which we call *"incremental code quality enhancement"*.

The article is structured as follows. Sect. 2 motivates for the topic with a discussion on typing. In Sect. 3 we present the concept of pluggable programming language features. Sect. 4 and 5 present a sample application and a research prototype of the concept. Sect. 6 introcudes incremental code quality enhancement and how this can be achieved via pluggable programming language features. Sect. 7 discusses the results. Sect. 8 concludes this article.

## 2 Static versus Dynamic Typing

In statically typed programming languages, variables and operation parameters are assigned a type which is checked at compile type. In contrast, in dynamically typed languages type checks are performed at run-time. While typing is only one of many

programming language features, there is, currently, a strong correlation between dynamically typed and RAD languages and statically typed and mainstream application development languages, respectively. Languages that allow static as well as dynamic typing are rare. Examples are VisualBasic, Perl6, Common Lisp (in part), and, recently, C# 4.0.

The differences between statically and dynamically typed languages are sometimes exaggerated as “language war”. Advocates of static typing claim:

- Earlier detection of programming mistakes, e.g. preventing adding an integer to a boolean
- Better documentation in the form of type signatures, e.g. incorporating types of arguments when resolving operation names)
- More opportunities for compiler optimizations, e.g. replacing virtual calls by direct calls when the exact type of the receiver is known statically
- A better design time developer experience, e.g. via auto-completion by the development environment

Advocates of dynamic typing claim:

- Higher coding efficiency since the resulting code is less verbose
- Higher expressiveness via language features like closures, typically found in dynamically typed languages
- Better reusability since variables and operations are not (unnecessarily) restricted in use by types

Language wars are not at all necessary – we fully agree with Meiyer and Drayton in their article “Static Typing Where Possible, Dynamic Typing When Needed: The End of the Cold War Between Programming Languages” [18].

Typing is only one aspect of a whole spectrum of contracts [19] between providers and users of components – all with the purpose of reducing errors and, therefore, increasing quality. But why stop with typing? We claim that every intrinsic aspect of an operation should be expressed explicitly as part of the contract and should be checked as early as possible: value restrictions, pre- postconditions, invariants, exceptions, usage protocol, etc.

In contrast, statically typed languages allow – and even require – to specify the types of all parameters but do not provide language features to specify more[1]. Static typing alone seems like an arbitrary point in the whole spectrum – for some interfaces it may be adequate to specify constraints more strictly, for others less.

While the adequate degree of contract specification relates to the application in its final, production-ready state, we add another dimension to the argument: the development time. In early development stages, particularly during prototyping, it is not necessary to specify full-featured contracts. The demand for precise contracts increases gradually during the development process.

[1] Eiffel does allow for specifying pre- and postconditions but Eiffel is, currently, not in widespread use.

## 3 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, namely customizing features per development stage.

# 4 Example Domain: Customer Management Component Interfaces

## 4.1 Customer Management

We demonstrate pluggable programming language features exemplary via interfaces for a customer management component of a business information system. We use the term *component* as a functionally coherent unit of software with specified interfaces (provided and required). An *interface* represents the external view of a component. It consists of operations. An *operation* provides functionality of a component. It is specified via syntax (signature) and semantics (behavior). See, e.g., [22].

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.

## 4.2 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 specification, e.g., read-only, modifying

| 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 |

**Fig. 1.** Language Feature Configuration

- 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
- Parameter Documentation: describing the meaning of the operation parameters
- Non-functional characteristics: specifying (formally or informally) performance and other non-functional characteristics

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.

### 4.3 Language Feature Configuration

Consider, for instance, 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. For the proof of concept, operation names and parameter names are sufficient. For Alpha Release and Beta Release, the architect demands additional language features like parameter typing, visibility, and exceptions. For the final product, full documentation is mandatory.

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

## 5 Language Features for Interface Specification

### 5.1 Research Prototype in Lisp

We have chosen Lisp[2] [17] as the implementation language for our research prototype to demonstrate pluggable programming language features.

The following features make Lisp ideal for experimenting with language extensions:

**Typing.** Lisp is dynamically typed yet provides a powerful type system as well as features for object-oriented programming.

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

**Code is Data.** Lisp has a minimalistic syntax with the list as the basic data structure. Lists are not only used to express application data but also to express Lisp code itself. This makes it particularly convenient to transform Lisp programs via Lisp programs.

**Macro Processor.** The built-in macro processor allows introducing new language features efficiently and with limited effort.

Unlike Java and C#, Lisp does not provide an explicit language feature for interfaces. But as in other languages like C++, the concept may be emulated.

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.

### 5.2 Operation and Parameter Naming

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

### 5.3 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.

### 5.4 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. 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.

### 5.5 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"))
```

### 5.6 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.

### 5.7 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 postconditions 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.

The architect can configure the conformance levels per application or per application parts, e.g., components, and adapt the configuration over development time. Developers get direct feedback whether their code complies to the current conformance level.

## 6 Incremental Code Quality Enhancement

### 6.1 Code Quality

Software engineering [8,5], in essence, aims at developing *high-quality* software at *reasonable cost*. *Software quality* [15] can be assessed via quality models like ISO 9126 [11] and has internal and external aspects. Internal aspects address the application developer via ease of development and maintenance. *Code quality* is an important internal aspect which is produced during programming.

In a software development project, the implementation usually is based on an upfront design of a particular architecture. Unfortunately, during implementation one often has to discover deficiencies in this architecture the first time: insufficient separation of concerns, cyclic dependencies, inconsistent interfaces, unsuitable couplings, violated layering, etc. To really fix these kinds of deficiencies, an extensive *refactoring* [7] would be required. Due to time and budget constraints this is either refused and the resulting software is shipped in time but with lower quality, or it is performed and the resulting high-quality software causes the the project to suffer from time and budget overrun. High-quality results and reasonable costs are two contrary goals which are hard to bring into balance.

### 6.2 Incremental Software Development

One can tackle the problem in advance in two ways: either by investigating more into the design discipline to avoid expensive refactorings at all, or by following an *incremental software development* approach where "merciless refactoring" [1] results in small and less expensive steps all the time. The latter approach is a central aspect in agile software engineering methods [16,12]. One variant is *evolutionary prototyping* [6]. In evolutionary prototyping, an application is implemented prototypically first as a proof of

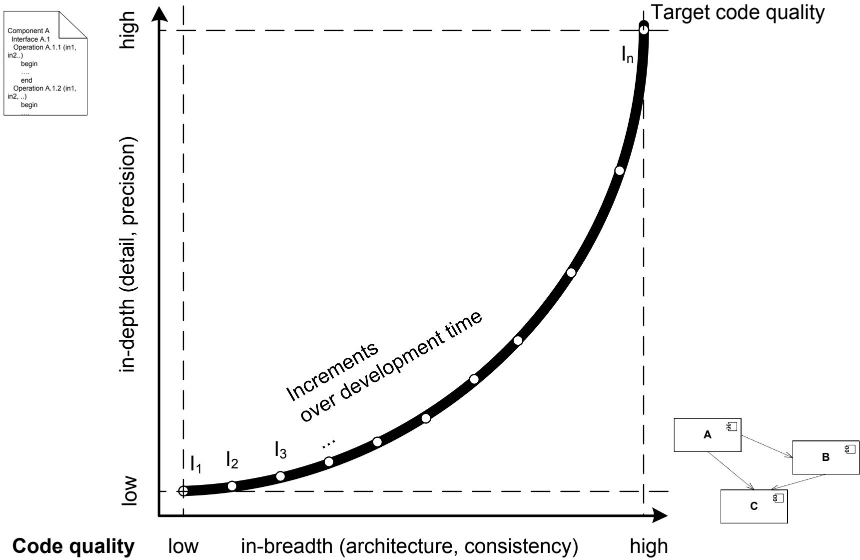


**Fig. 2.** Code Quality Dimensions

concept which is then, incrementally, refined towards the final product, thus constantly enhancing code quality. Evolutionary prototyping may reduce development time and costs while improving user involvement. While evolutionary prototyping is not suitable for all kinds of software projects, its benefits have been proven in numerous projects of different sizes [2,9,14,10].

But how to incrementally enhance code quality in evolutionary prototyping *effectively* and *efficiently* during implementation? To answer this question, we distinguish two *dimensions* (see Fig. 2) of *internal code quality* [11].

**In-breadth** code quality concerns the structural consistency of the entire application, i.e., the application architecture. Example: conformance of application components to a layer concept.

**In-depth** code quality concerns correctness and precision of code in detail. Example: specification and validation of an operation's pre-condition.

We plead for optimizing in-breadth internal code quality *before* in-depth internal code quality during incremental development (see Fig. 2). This allows for effective and efficient quality enhancement – for the following reasons.

**Efficiency.** Optimizing in-breadth quality of relatively small code reduces the costs for refactoring [7] compared to refactoring voluminous code with all details implemented and documented already. For example, splitting a component that does not conform to a layering concept may induce a a lot of refactoring if all operations have already been implemented in detail.

**Effectiveness.** In-breadth code quality can be optimized largely independently of details. E.g., conformance to a layering concept is independent of individual preconditions. Thus, the effectiveness is not compromised by optimizing in-breadth quality before in-depth quality.

### 6.3 A Method for Incremental Code Quality Enhancement

When developing incrementally, particularly via evolutionary prototyping, then the architect and the programmers should proceed according to the following method.

1. Select a programming language and programming environment that fulfills the requirements from Sect. 3.
2. Define a language feature configuration according to the quality requirements of the final product. Provide automatic conformance validators where feasible and economically worthwhile.
3. Define language feature configurations for development stages and possibly per application part (e.g., component). Provide automatic conformance validators where feasible and economically worthwhile.
4. Develop the application incrementally. Develop a prototype quickly and with as little effort as possible. Spend sufficient time for optimizing in-breadth code quality and this way improve the architecture.
5. Continue incrementally implementing the application, optimizing in-depth code quality. Control the quality of the increments via conformance checks.
6. Put the application into operation only after the conformance checks for the final product have passed.

### 6.4 Pluggable Programming Langauge Features and Incremental Code Quality Enhancement

In a way, incremental code quality enhancement “happens” implicitly in most software development projects, today. However, it is not explicitly and as vigorously pursued in a controlled manner as described in the method above. Pluggable programming language features ideally support incremental code quality enhancement. This is because language features can be added optionally and independently during development and, at the same time, there is full control via language feature configurations.

## 7 Discussion

### 7.1 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 (incremental code quality enhancement). Language feature configuration prevents arbitrary use of language features at the programmers' goodwill. Certain quality levels at certain development stages can be enforced.

### 7.2 Language Support Today

Current programming languages, both in industry and academia, only poorly support pluggable programming language features. 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). More advanced features like pre- and postconditions are not directly provided and may only indirectly be provided, e.g.,via byte-code injection.

**RAD Supporting Languages, Dynamically Typed.** RAD supporting languages like Perl, Smalltalk, 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. Language features may possibly be added – e.g., with Lisp macros as demonstrated 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, the first mainstream programming language has recently incorporated dynamic typing optionally — one important step towards pluggable programming language features.

### 7.3 Related Work

In [18], Meijer and Drayton plead for typing as a pluggable programming language feature. 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" [3] is similar. His implementation of Strongtalk [4] on the basis of Smalltalk is an example of a pluggable language feature, namely typing.

With the Scala programming language [20], 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) [13] 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.

## 8 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 language feature configurations. It is particularly suited for incremental code quality enhancement, a development method in which in-breadth code quality is optimized before in-depth code quality.

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 [18].

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 and incremental code quality enhancement 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 and incremental code quality enhancement 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.

## References


1. Beck, K., Andres, C.: Extreme Programming Explained: Embrace Change, 2nd edn. Addison Wesley, Reading (2005)
2. Berger, H., Beynon-Davies, P., Cleary, P.: 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 (2004)
3. Bracha, G.: Pluggable type systems. In: OOPSLA Workshop on Revival of Dynamic Languages (2004)
4. Bracha, G., Griswold, D.: Strongtalk: Typechecking Smalltalk in a production environment. In: Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications, OOPSLA 1993 (1993)
5. Broy, M., Jarke, M., Nagl, M., Rombach, H.D.: Dagstuhl-Manifest zur Strategischen Bedeutung des Software Engineering in Deutschland. In: Perspectives Workshop Dagstuhl, Germany (2006)
6. Floyd, C.: A systematic look at prototyping. In: Approaches to Prototyping, pp. 1–18 (1984)
7. Fowler, M.: Refactoring: improving the design of existing code. Addison-Wesley Longman Publishing Co., Inc., Boston (1999)
8. Ghezzi, C., Jazayeri, M., Mandrioli, D.: Fundamentals of Software Engineering. Prentice Hall PTR, Upper Saddle River (2002)
9. Gordon, V.S., Bieman, J.M.: Reported Effects of Rapid Prototyping on Industrial Software Quality (1993)
10. Hekmatpour, S.: Experience with evolutionary prototyping in a large software project. SIGSOFT Softw. Eng. Notes 12(1), 38–41 (1987)
11. ISO. TR 9126-4: Software Quality (2004), `http://www.iso.org/iso/catalogue_detail.htm?csnumber=39752`
12. Kelter, U., Monecke, M., Schild, M.: Do we need 'agile' Software Development Tools? In: NetObjectDays (2002)
13. Kiczales, G., Lamping, J., Mendhekar, Videira Lopes, C., Loingtier, J.-M., Irwin, J.: Aspect-Oriented Programming. In: Aksit, M., Auletta, V. (eds.) ECOOP 1997. LNCS, vol. 1241, Springer, Heidelberg (1997)
14. Lichter, H., Schneider-Hufschmidt, M., Züllighoven, H.: Prototyping in industrial software projects—bridging the gap between theory and practice. In: ICSE 1993: Proceedings of the 15th International Conference on Software Engineering, pp. 221–229. IEEE Computer Society Press, Los Alamitos (1993)

15. Liggesmeyer, P.: Software-Qualität. Testen, Analysieren und Verifizieren von Software. Spektrum Akademischer Verlag (2002)
16. Martin, R.C.: Agile Software Development, Principles, Patterns, and Practices. Prentice Hall, Englewood Cliffs (2002)
17. McCarthy, J.: Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. Communications of the ACM 3(4), 184–195 (1960)
18. Meijer, E., Drayton, P.: Static Typing Where Possible, Dynamic Typing When Needed. In: Workshop on Revival of Dynamic Languages (2005)
19. Meyer, B.: Object-Oriented Software Construction, 1st edn. Prentice-Hall, Inc., Upper Saddle River (1988)
20. Odersky, M.: An Overview of the Scala Programming Language: EPFL Technical Report IC/2004/64 (2004)
21. Sommerville, I.: Software Engineering, 7th edn. International Computer Science Series. Addison Wesley, Reading (2004)
22. Szyperski, C.: Component software. Addison-Wesley, Harlow (1998)