Language Constructs for Improving Reusability in Object-Oriented Software

Marko van Dooren
University of Leuven
Leuven, Belgium
Marko.vanDooren@cs.kuleuven.be

Eric Steegmans
University of Leuven
Leuven, Belgium
Eric.Steegmans@cs.kuleuven.be

ABSTRACT

The objective of this research project is to improve the reusability of object-oriented software. We have introduced anchored exception declarations to allow checked exceptions to be used conveniently in reusable software elements such as design patterns. We are now investigating an innovative inheritance mechanism based on existing inheritance mechanisms like that of Eiffel and on traits. The resulting mechanism should allow a programmer to construct a class from existing components with little effort.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features; D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*Semantics*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Error handling and recovery*; D.1.5 [**Programming Techniques**]: Object-oriented Programming

General Terms

Languages, theory, reliability, verification, design

Keywords

Exception handling, anchoring, inheritance, component

1. DESCRIPTION OF PURPOSE

The purpose of this research project is to increase the reusability of object-oriented software.

With current programming languages, it still takes too much effort to transform a high-level design into a working program. Similar code is written over and over again because either the programming language does not provide the necessary reuse facilities, or reuse incurs too much overhead compared to the amount of work that would be saved. In particular, we found that checked exceptions are difficult to use in most design patterns, and we think that current inheritance mechanisms do not suffice.

Checked Exceptions

An exception handling mechanism increases the reusability of software by removing specific exception handling code from a component. Exceptions can be divided into two categories: *checked* exceptions and *unchecked* exceptions. Checked exceptions must be propagated explicitly by listing them in the method header, while unchecked exceptions are propagated implicitly. Note that this is different from the categorisation *caught-uncaught*, which denotes whether an exception can exit a method body. But while checked exceptions increase the robustness of software, they also decrease the adaptability and flexibility. It is often necessary to needlessly adapt other methods when the exceptional behavior of a single method has changed. In addition, checked exceptions must often be handled even if they cannot be signalled [4]. As a result, many programming languages completely omit checked exceptions, and even if they are available, developers often do not use them. The root of the problems is the lack of expressiveness of the exception clause in current programming languages – e.g. the `throws` clause in Java.

OOPSLA'05, October 16–20, 2005, San Diego, California, USA.
ACM 1-59593-193-7/05/0010.

Inheritance

Existing approaches to increase the reuse facilities of inheritance either focus on multiple inheritance including subtyping, or on implementation inheritance. No language however offers both in a convenient way, and we think that even the specialty of each approach has its problems.

Traits are composable units of behavior [3], which can be used, like code inheritance, to reuse parts of related code without creating a subtype relation or using delegation. We think that traits, which are used in Scala [2], are a big step in the right direction for allowing better reuse of code, but that the approach is still limited. First of all, traits do not provide multiple inheritance even when it would be natural to use it. Implementation inheritance with traits technically works fine, but we think it is not convenient enough. For example, to connect a trait to another, all required methods must be connected one at a time. In addition, the contracts of the required methods must be duplicated every time they are used. Another problem is that traits are invisible to the clients of a class, making it more difficult to understand the behavior of that class. If a client is already familiar with a certain trait, she could understand a part of a class by simply looking at the type of the trait. But if that type is hidden, she must reconstruct her understanding of that part of the class by reading the contracts of the methods, which needlessly complicates matters. Traits are also less suitable for keeping the interface of a class simple. While traits allow the programmer to remove certain methods of a trait, meaning that the programmer of the trait can provide as much methods as she wants without worrying about bloat in the parent classes, they do not allow methods to be hidden. By doing this, you may deprive certain clients of valuable methods. We think it would be better if a programmer could choose to export methods based on the estimated use of the class, but still make the others available. After all, it is difficult to predict in advance how clients will use the class. In addition, traits cannot contain state, resulting in more work for the client of the trait, even when a default representation makes perfect sense.

Multiple inheritance in Eiffel can also be used to improve the reusability, but its inheritance mechanism is not suited for implementation inheritance. Specifically, if a number of methods is duplicated, and they invoke each other, such an invocation will always be bound to the default method for the signature selected by that invocation, which is selected in the subclass. As a result, the mechanism cannot be used to add a component – in the sense of a trait – more than once to a class without modifying the inherited code to use hook methods because all internal invocations are bound to the default selections.

2. GOAL STATEMENTS

Checked Exceptions

We introduced anchored exception declarations for declaring the exceptional behavior of a method relative to that of other methods.

Inheritance

We want to create a new inheritance mechanism based on the inheritance mechanism in Eiffel and on traits. The new inheritance mechanism should allow a class to be constructed from relatively small components – subclassing – but should still allow a limited form of multiple inheritance – subclassing and subtyping. It should also offer the possibility to hide methods inherited through implementation inheritance but still make them available to clients of the inheriting class. We will also study to what extent it is possible to allow the duplication of state without encountering the traditional diamond problems.

3. APPROACH

Checked Exceptions

We have solved the problems with checked exceptions by introducing *anchored exception declarations*. They allow the exceptional behavior of a method to be declared relative to that of others, denoted as `like t.m(args) propagating (`*`Exception List`*`) blocking (`*`Exception List`*`)`. This way, we prevent gratuitous adaptation of methods when the exceptional behavior of a method further in the call-chain is modified. In addition, anchored exception declarations allow call-site type information to be used to limit the set of checked exceptions that can be signalled by a method invocation, removing the need for inconvenient and dangerous dummy catch clauses.

We have formalised the semantics of anchored exception declarations, and to ensure soundness, we have defined rules that specify whether or not an exception clause EC_a is stronger than EC_b. These rules are used to ensure that a method cannot signal more exceptions than the methods it overrides, and that the implementation of a method cannot signal an exception when it is not specified by the exception clause of the method.

On the methodological side, we have shown that anchored exception declarations can be used without violating the principle of information hiding, and have provided a guideline for when to use them. In addition, we have defined criteria to determine which modifications – triggered by a change in the exceptional behavior of a method – of code are bad and which are good. We have shown that anchored exception declarations eliminate most of the bad modifications while still forcing all good modifications to be made.

We have implemented the construct in Cappuccino, an extension to the ClassicJava programming language.

The work on anchored exception declarations will be presented at the OOPSLA 2005 conference [4].

Inheritance

The new inheritance mechanism will make a distinction between subtyping and subclassing relation, as done in e.g. the programming language Timor. In this research project, we currently focus on the subclassing relation for implementation inheritance, and on the combination of subclassing and subtyping for inheritance as it is used in Eiffel. Duplication of methods and state will likely not be allowed for the subtyping and subclassing combination, whereas it will be allowed or maybe even be mandatory for subclassing relations. To prevent violation of the integrity of related state variables, we intend to use the concept of *data groups* [1]. We will only allow the duplication of entire data groups.

A novelty of our approach is to give a role name to an inheritance relation. This name will be used to connect components (traits), to document the type of the component and its role in the parent class, and to make methods that are hidden but not removed available to clients of the parent class. Components can now require to be connected to a particular type of component by listing it as a required parameter for the parent class – like a generic parameter. This prevents duplication of the behavioral specification of the required component. The name of the inheritance relation also reduces the dependency of super calls on the hierarchy by providing an extra indirection. Instead of using a class name to select a method, the name of the inheritance path can be used.

We also plan to introduce a facility for exploiting patterns in the names of methods. For example, the methods of a component for associations will be named `getX`, `setX`, `containsX`,.... If a parameter can be used in these names, it becomes more practical to use it as a building block for a class.

We plan to implement the resulting inheritance mechanism in a modified version of Java, and provide a prototype compiler. To demonstrate the mechanism, we are working on components to model unary or binary associations, and plan to extend this work to components for building a graph structure on top of those associations. Such components should enable a developer to quickly translate the structural part of his design into working code, and easily add complex methods that traverse the associations among classes by using the graph components.

4. REFERENCES

[1] K. R. M. Leino. Data groups: specifying the modification of extended state. In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 144–153, New York, NY, USA, 1998. ACM Press.

[2] M. Odersky and al. An overview of the Scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[3] N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable units of behavior. In *Proceedings ECOOP 2003 (European Conference on Object-Oriented Programming)*, volume 2743 of *LNCS*, pages 248–274. Springer Verlag, July 2003.

[4] M. van Dooren and E. Steegmans. Combining the robustness of checked exceptions with the flexibility of unchecked exceptions using anchored exception declarations. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*. ACM Press, 2005.