Implicit first class genericity,
or Dynamic Object Instantiation

J.D.M. Kuśmierrek
University of Warsaw,
Institute of Informatics,
Warsaw Banacha 2

ABSTRACT
While the software systems are growing bigger, and the market demands shorter software production cycles, IT companies need more and more extensive reuse of existing components. Additionally, sophisticated reuse scenarios demand flexible components which are able to adapt to different needs. As a result, it is required that all the components are as customizable as possible.

In object-oriented (oo) programming languages, the base tool of reuse and customization is inheritance, which is widely known and exploited, however it shows its limitations when it comes to reuse complicated components. There exist several techniques which allow a stronger and more flexible customization of components. Unfortunately, most of those techniques require additional work to be performed by the programmer in advance, or have other limitations. As a result, those techniques are not widely used, and there is still research to be done in this area.

In this paper we present a new proposal that offers a feature for replacing a class with a compatible subclass within a library without the need of modifying its sources. As a result, it increases the possible reuse and the customization scenarios of existing components, with little additional coding required. Our idea is presented as an extension of an arbitrary oo language, however it is studied in more details as an extension of Java called ImpliJava.

Keywords
object-oriented programming, reuse, inheritance, locally visible redefinitions

1. INTRODUCTION
The rapid development of software systems requires more and more extensive reuse of existing components in different scenarios. As a result, it is required that all the components are as customizable as possible. Unfortunately, language constructs supplied by the mainstream oo languages are not flexible enough for some of the customization scenarios met nowadays (as also pointed in [7, 11, 20, 27]).

One such a scenario is when one needs to reuse some class while refining the behavior of it. It might raise a problem when the class uses several another classes inside, delegating some calls to them, and the programmer needs to modify the behavior realized by the delegate's class (called also: subcomponent's class). Consider an example of LoginDialog component class written in Java (see Fig. 1). This component is used within some client application, in order to allow the user to login to a service. Then, suppose that, within the same application, we need to reuse this component, in order to allow the user to login to a different kind of service. Assume also that this service uses encrypted connection. As a result, we need to refine the behavior of LoginDialog by: (i) using MySSLSocket class instead of Socket to obtain an encrypted connection, (ii) changing the way the button is painted (by drawing the padlock symbol on it). We also need to keep the original version of LoginDialog and Socket classes, since those are used in another parts of application.

In such a case the inheritance mechanism, which is a base tool of the component customization in oo languages, does not work well. In fact, a modification of behavior of the subcomponents (implemented in Button and Socket classes) within the subclass of LoginDialog cannot be obtained without copying the whole constructor body. This, in turn, can cause the loss of compatibility with future versions of LoginDialog class. Another examples showing such limitations of the inheritance can be found in [6, 20, 27].

There exist many approaches (discussed later in this paper) which extend the expressiveness of oo languages, by allowing to declare, use and customize components in a more flexible way. However, not any of these proposals, combines all of the following properties:
allows the programmer to reuse an arbitrarily complicated component, with behavior redefined on any level;
• allows local redefinitions of base component, which does not influence other uses of the same component in other parts of the program;
• allows redefinitions without any anticipation by the designer of the original component (a property called supporting unanticipated changes in [25]);
• naturally separates the declaration of such redefinitions from their application in different local contexts;
• uses simple syntax and semantics, familiar for a programmer using a mainstream oo language (giving the chance of acceptance by the industry).

In this paper we present the concept of implicit first class genericity, which enables the programmer to replace a class with a compatible subclass within an existing component. This operation is made by using an annotation at the point of usage of that component, however without the need of modifying its source code. Our idea is presented as an extension of an arbitrary oo language, however it is studied in more details as an extension of Java called ImplJava.

The paper is structured as follows: Section 2 discusses existing solutions which can be used to partially solve problems similar to the one of LoginDialog. Section 3 presents our idea of the language extension, designed to allow intuitive, unanticipated, locally visible, class redefinitions. We present the details of our approach by introducing the ImplJava language, which is an extension of Java with constructs that realize our ideas. Section 4 describes the implementation scheme for our solution, which works by a translation from ImplJava to Java. Section 5 comments on the possible implementation variants of our approach, and on performance costs of a prototype implementation of our approach. The analysis of performance is supported by some benchmarks. Section 6 presents an extension of our approach, which makes the type-checker aware of class redefinitions. Section 7 discusses the possible extensions of our approach. Section 8 concludes the paper.

2. EXISTING SOLUTIONS

First class Generics.
A very general, safe, and well-founded tool aimed at the design of the customizable components is the mechanism of First Class Generics [3] implemented in MixGen language. In a language equipped with such a feature, a class \( C \) can be declared as parametric (via formal class parameters such as \( X \)), and, within the declaration of this class it is possible to write an expression creating an object from the parameter class \( X \) (using \( \texttt{new } X(\ldots) \) syntax). As a result, within the \( C \) object instances created with class \( D \) used as parameter \( X \), the actual execution of \( \texttt{new } X(\ldots) \) expression will create an object of class \( D \). In the contrast classical Java generics are second class, which means that generic parameters can only be used in type expressions and are erased in the compiled bytecode.

However, to achieve the desired level of flexibility of the MixGen code (that is, supporting all the potential future customization scenarios), it would require from each class to have a dozen of class-parameters, passed to methods and other subcomponent classes. This, in turn, would make the code harder to read, maintain and understand. Additionally, it would require the designer of a class to anticipate all the possible uses of such class.

Aspect-oriented programming.
A well known technique allowing the modification of behavior of existing code is Aspect-oriented programming (AOP). The examples of Java AOP extensions are: AspectJ [20], and CaesarJ [4].

The core idea of AOP is that the programmer can write sequences of instructions (called advices) in separate modules (called aspects), which will be executed at chosen points during the program execution, as specified in pointcut declarations. In particular, pointcut can specify that some advice should be executed before or after the execution of some chosen methods of classes declared elsewhere. Using such a solution the programmer can easily extend the behavior of existing class within the whole system. Additionally, by using advanced pointcuts (with \( \texttt{cflow} \) command) the programmer can specify that the advice will be executed only when a given method was called from some specific point in the program. Moreover, using inter-type declarations, existing classes can be extended with new methods and fields. Unfortunately, there is no possibility to restrict the range of such modifications, because those are always global.

In general, AOP is a powerful tool useful in many scenarios requiring a global change of behavior, spanning many classes. A typical application is implementation of cross-cutting concerns like testing, security, persistence etc. However, it is less suited for locally visible changes of existing classes. Additionally, it introduces several new language constructs with new syntax, therefore it requires some time for a programmer to make the most of it.

Object factories.
There exists a design pattern, aimed at making the software system open to the modifications of the creation procedure of subcomponents, called object factories (see [12]). Working according to this pattern, when a programmer needs to create a subcomponent, he declares a separate virtual method responsible for the creation of this object and uses this method in places when new objects need to be created. As a result, such a method can be later overridden in subclasses, to instantiate subcomponents from different classes. A variant of this approach is to use a separate object instance which is solely responsible for instantiation of subcomponents, to which all requests for a new subcomponent are delegated.

However, this approach is rather cumbersome, because it requires a significant amount of additional work to be performed by the designer of the original class. Therefore, in practice, it is used only for the most important subcomponents. It also requires from the class designer an anticipation of all potential customization needs.

Deferred sub-component setup.
Another methodology which allows to avoid customization problems described in the introduction is based on the separation of the task of subcomponents initialization from the task of initialization of the main component object. In such an approach, the client code, after creating an object of the given class, must initialize its subcomponent fields. This in turns, allows one to alter the behavior of main com-
ponent, by initializing a subcomponent fields using different classes.

However, it makes such component more difficult to use, needs more documentation, and is prone to errors, because a programmer writing the client code can forget to initialize some of the subcomponents, and this fact will not be detected during the compilation.

**Nested inheritance.**

Nystrom, Chong and Myers introduced the idea of nested inheritance [25], allowing the programmer to extend not only classes, but also another namespaces (like the packages in case of Java). Additionally, when extending the namespace, a programmer can also redefine some nested namespaces of the extended one. For example, when declaring package P2 as an extension of package P1, the programmer, can also redefine some class defined in P1. As a result, he obtains a version of package P1 which uses a modified version of given internal class. Similarly, when extending a class, one can also override nested classes of the given one.

This idea is also present in CaesarI language [4], which also supports mixin-composition of such package extensions (allowing to merge two extensions of one package), and combines it with the Aspect-Oriented features.

As a result, the nested inheritance allows the programmer to redefine subcomponents of main component, assuming that declaration of classes of the subcomponent fields are also statically nested in the declaration of the main component class, or at least are placed in the same package. However, when a component uses subcomponents instantiated from classes declared elsewhere, the programmer is not able to modify their behavior.

**Virtual classes.**

Madsen and Möller-Pedersen described a language construct called virtual classes [22], which was implemented as a part of the language BETA [23]. This mechanism permits to declare a class attribute as a member of a class (in addition to field and method declarations). Such a class attribute can be used to construct new objects from and as a parent class in local class declarations. Additionally, when such a class attribute is declared as virtual, it can be also redefined in the subclasses. Moreover, all the references to the virtual class attribute (in the object instantiation and the class extensions) are resolved dynamically. Therefore, in the subclass, the inherited code will use the redefined class attribute.

As a result, virtual classes enable one to redefine subcomponents of existing components. However, this approach requires the anticipation of future needs of possible users of a given class. This anticipation must imply declarations of class attributes and references via class attributes, instead of direct references to specific classes. Therefore it is less suited for performing unanticipated changes.

**Multimethods.**

In languages supporting multimethods (known also as multiple-dispatch), like Common Lisp Object System [24], Dylan [13], and MultiJava [11], the dynamic dispatch of the method is based not only on the first, implicit, parameter (as in Smalltalk [16], Java [17], C# [19],...), but also on the types of remaining explicit parameters. As a result, a method does not have to be declared in the class on which its dispatch depends. Therefore, in such languages, a programmer is able to extend the functionality of existing classes without modifying their source code. As a result, all the objects created from those classes will start to behave differently.

However, the scope of those changes is global. Therefore, an unaware programmer who needs to modify the behavior of a given class in some particular context might by accident cause unpredictable results in different parts of the program (which use the same class for different purposes). Therefore, this approach is less suited for performing locally visible changes of the behavior of existing classes, used within the bigger system.

**Feature oriented programming.**

Feature oriented programming (FOP) [26] is a paradigm in which a program is defined as a composition of parts of the program (called features). Such a feature can contain declarations of new classes, as well as redefinitions of existing classes. One of the motivations behind FOP was to support the development and maintenance of product lines, which share most of the code and differ only in small parts.

AHED (Algebraic Hierarchical Equations for Application Design [5]) is an architectural model for FOP with implemented tools (AHED Tool Suite) supporting compositional program construction according to the rules of FOP.

The advantage of the feature oriented approach are: generality, which makes it applicable to different programming languages, and the fact that at the language level it does not introduce many new constructs. Thus, it is easy to understand and use. However, all the extensions of classes added by new features have global scope, therefore it is not possible to use different versions of the same class in one program.

**Classboxes.**

Bergel, Ducasse, Nierstrasz and Wuyts introduced the notion of classbox. A classbox is an environment containing standard declarations of classes, but it can also contain definitions of “class refinements”. A class refinement is a declaration specifying modifications of an existing class defined elsewhere. Such a modification includes declarations of new methods and fields, as well as redefinitions of existing methods. Every such refinement is valid within the classbox in which it is declared. As a result, each method call originating from the given classbox (or other code called from within this classbox), referring to a method of a refined class, will execute the refined method body. A distinguishable feature of this approach is that different calls to the same method of the same object will behave differently depending on from which classbox the call came (directly and indirectly).

When the classbox imports another classbox, then it automatically “uses” all the refinements of the imported classbox. On one side it allows the programmer to use easily a large set of refinements, while on another side it does not allow the programmer to use only some of the refinements declared in the classbox.

The classbox approach was first used to the develop an extension [7] of Squeak (which is a dialect of the Smalltalk language [16]). Later, Bergel et al. applied also the same methodology to Java obtaining a language ClassBox/J [6]. In this language, a Java package declaration plays the role of a classbox. Both classbox for Squeak and ClassBox/J have implemented prototype compilers.

In the classbox approach, the programmer can define “lo-
cally visible class redefinitions", and control their scope. However, those redefinitions use syntax and semantics distinct from the classical refinement (by inheritance) and therefore, in our opinion is more difficult to control and understand. Additionally, this approach is subject to the "diamond problem" of conflicting class redefinitions, which can occur when one classbox imports two other classboxes containing refinements of the same class.

Context-Oriented programming.

Context-oriented programming (COP) [27] is a set of programming techniques, which allows the software to dynamically adopt the behavior to the execution context. In this approach, a system can dynamically adopt to changes on the actor side, on the environment side, and on the system side. In the Context-Oriented language, apart from classical class declarations, a programmer can use layer declarations to specify class refinements. Each of such refinements can contain redefinitions of existing methods of a given class, as well as additions of new fields and methods. Then those layers can be dynamically imported and disabled explicitly at the point of the call (client side), or by considering various conditions on the server side.

Currently there exist at least five languages supporting COP. Amongst them there are a Java extension called Context*, and a Common Lisp extension named ContextL.

By adopting those languages, a programmer gains a great flexibility in influencing the behavior of the existing code, also in ways unanticipated by the designer of the original classes. Additionally, it allows numerous changes of behavior of existing objects during their lifetime. However, such a great power in flexibility may come with a price: (i) it might make the analysis of the program execution more difficult, (ii) it might make formal reasoning about the actual type of objects more difficult, (iii) the lookup procedure of a method call is more complicated, which might imply significant performance cost.

Dynamic scope of identifiers.

In Common Lisp [18], which is a language supporting dynamically scoped identifiers, a programmer can introduce and reference dynamic identifiers (called special), which will be resolved dynamically by using the callstack. With this technique, a programmer can define dynamically resolved function which will be responsible for object creation and then redefine it on the client side.

This approach is rather general and powerful, however it is more like a design pattern, not an explicit language support for unanticipated class redefinitions. As a result, it requires some implicit contract between the designer of the component class and the client of that class. Such a contract will not be verified statically (as the whole language is dynamically checked), therefore in cases of complicated component classes, it becomes difficult for a programmer writing the client code to check whether the code will not cause any errors, resulting from improper redefinitions, and the actual range of such redefinitions.

Changeboxes.

Changeboxes [14] are a mechanism which allows the programmer to treat changes of a program as first-class entities, by encapsulating them in changeboxes. Each changebox consists of specifications of new classes, new fields and methods added to existing classes, refined methods, or even changes of names of classes and class members. A changebox can represent some bug fix, or a new version of the software. Using the mechanism of changeboxes, a programmer can dynamically change the version of the software system during its execution. Additionally, tools supporting this approach can generate changebox’s specification on the fly, when a programmer changes the source code of a program.

As a result, changeboxes are a useful mechanism to work with critical application which cannot be stopped even for a short period of time. It is also aimed to model evolution of software in time, and to control different branches of an application, rather than to develop different versions of one component within one program (which is, instead, our goal).

One of the downsides of the current implementation is that it shows significant performance overhead in practical applications, resulting in 4-5 longer execution times. However, since the implementation is more prototype, it’s performance can improve in the future.

3. ADDING THE SUPPORT FOR UNANTIC-PATATED CHANGES

In this section we present our idea of how to add support for locally visible unanticipated changes (see Sec. 1) to a class-based, oo language. We will present the details of this approach by applying it to Java [17], thus obtaining a proposal for a new language called ImpliJava. However, with minor modifications this idea can be applied to most of the mainstream oo languages.

3.1 Base idea

We equip Java with a feature for replacing a class used in a new ... expression with a compatible subclass. This replacement works on existing code without modifying it’s sources, by making the modification active only for calls coming from chosen parts of the client code. The replacement is enabled by placing an additional annotation in the client code at the point where the replacement is needed (for example, at the point of call to a method or a constructor), and the replacement is valid only within the scope of such an annotation (for example, only during the execution of the given call). We obtain such a result by adding a new operator with which works as follows.

Consider the constructor body of the class LoginDialog (Fig. 1), which contains the expression new Button(…). A programmer writing a call to this constructor can specify (using a with operator) that, during this specific constructor call, every object created from Button class should be created from ButtonPadLock instead, which has the method paint overridden (see Fig. 2). This works as if inside the definition of LoginDialog constructor every occurrence of new Button(…) have been replaced with new ButtonPadLock(…), however this replacement is valid only for that particular call to the constructor.

As a result, the programmer can use the LoginDialog component as if it was declared as a generic class, even though the declaration of this component does not contain any explicit class parameters. The replacement of Button with ButtonPadLock is allowed only if ButtonPadLock fulfills some requirements in order to be able to play the role of Button in the new ... expressions. The requirements that a class must fulfill in order to type-safely replace another
class ButtonPadLock extends Button
{
    public paint(...) {...} //constructors with the
    ButtonPadLock() {...} //same signature
    ButtonPadLock(String title) //as in Button class
}
... //within the range of those
with (Socket -> MySSLSocket) //operators objects will be
with (Button -> ButtonPadLock) //created from ButtonPadLock
{
    obj = new LoginDialog(...); //and MySSLSocket instead
} //of Button and Socket.

Figure 2: An example of ImpliJava code customizing the LoginDialog from Fig. 1

class are specified in Sec. 3.2. As shown on Fig. 2, the programmer can also replace the class Socket with the class MySSLSocket. In general, using this construct, it is possible to perform local redefinitions of classes in arbitrary code.

3.2 Simulating class relation

In order to be able to safely use the mechanism introduced above, we need to define a relation between classes which will guarantee that the substitution of the given class with another one will not make the code type-unsafe. In most of the oo languages (like Java [17], C++ [19]), the fact that class D is a subclass of class C ensures that every object created from class C can be safely substituted with an object of class D. In particular: in any context a new C(\(\tau\)) can be substituted with new D(\(\tau\)), assuming those expressions are properly type checked.

However, unfortunately, it may happen that even though new C(\(\tau\)) expression is properly type checked, new D(\(\tau\)) is not. It happens in situations when class C contains a declaration of a constructor with types of parameters matching types of \(\tau\), while class D does not. This, in turn, can happen because in Java, even though a subclass inherits all the methods, it does not inherit the constructors (see [17]). In some formalizations [2, 9], which assign types to classes (which are a different concept from types of objects), it means that the type of class D is not a subtype of the type of class C.

Therefore, in order to be able to verify that class C can be safely replaced with D in an object creation expression, we need to define new relation between classes (which is more restrictive than a subclass relation). We will use term “D simulates C” to denote that C can be replaced with D in all object creation expressions. In the above case we will call D a simulating class, and C a simulated one. The requirements needed for class D to simulate C, are the following:

- objects created from class D, can be cast to any type of the objects created from class C, which means that class D needs to be a subclass of class C.
- class D needs to support all the constructors of class C, which means that for each constructor of class C, class D needs to have exactly one constructor with the same types of parameters and compatible throw list.

Therefore it is easy to see, that class ButtonPadLock (see Fig. 2) simulates Button (which has two constructors with identical signatures). Thanks to the above definition, the simulation relation has two properties:

**Property 1.** For every properly type-checked program \(P\), classes C, D such that D simulates C, and every set \(S\) of new C(\(\tau\)) expressions appearing in it, a program \(P'\) obtained from \(P\) by replacing all expressions from \(S\) with new D(\(\tau\)) is also properly type-checked.

**Property 2.** The simulation relation is transitive.

One might argue that the requirement for the simulating class to support all constructors of the simulated class is restrictive. This is true for Java (and some other languages), since the programmer implementing the simulating class needs to copy all the signatures of the simulated class constructors and implement each of them with a correct super(...) call. Additionally, when the simulated class is extended with a new constructor, a simulating class must also be extended adequately. However, this problem can be avoided using tools available in other languages (like Delphi [1]) and also in some Java extensions like Java Layers [10] (using constructor propagation), and JavaMIP [8]. In the mentioned solutions, a subclass can automatically inherit all the superclasses’ constructors, so no additional work is required neither at the moment when the simulating class is declared, nor when the simulated class is extended.

Notice also that the requirement to support the constructors of the simulated class is caused by the fact that the simulating class needs to be able to substitute the given class in any context. For comparison, in the MixGen language [3], which allows the use of explicit first class genericity, each generic class can specify the list of constructors it expects from the actual parameter class. As a result, in MixGen, the class supplied to the given generic must provide only the constructors required. However, this is only possible with the usage of explicit specifications of the constructors used in the implementation of each generic class, therefore it does not support unanticipated changes.

3.3 Syntax of the redefinition operator

For each class D, which simulates class C, it is possible to use a redefinition operator with(C->D). Such an operator is always combined with the sequence of instructions whose behavior is to be altered. Such a redefinition is declared using the following syntax:

```java
with (C -> D)
{
    //11;
    ...
    //IN;
}
```

The meaning of such a redefinition instruction is the following: instructions 11... IN will execute as normal, with the only exception that every object created during the execution of those instructions, which normally would be created from class C, will be instead created from class D using the constructor with the same signature. This modification applies to any object initialization expression present in those instructions, also to the ones declared inside methods called directly or indirectly from 11... IN. In such a redefinition instruction, we call C a replaced class, and D a replacing class. For each such an expression, the compiler needs to verify if the replacing class simulates a replaced one.

In the case of multi-threaded programs we extend our definition in the following way: such a redefinition operator is valid for the time of execution of instructions 11... IN in the current thread, and also for the whole time of execution of any thread created during the execution of instructions 11... IN (and recursively for any thread created by this thread).
3.4 Redefinition operator nesting

When the redefinition operator is used in many different points of program, those occurrences can be nested. Such nesting can occur in one method and between different methods, for example, when inside the redefinition operator there is a call to a method which contains another redefinition operator.

When the object creation expression is executed in scope of multiple redefinition operators, the class to create an object from is chosen in the following way: We start with the class used in this expression and the list of redefinitions in scope is traversed starting from the most inner one. Every-time chosen class matches the one replaced by the redefinition we exchange it for the replacing one. The class which is left at the end is used to create an object from.

Therefore, when replacing and/or replaced classes of nested operators coincide, one can witness the following scenarios:

- A replaced classes of the outer and inner operator can be identical:

```java
with (A -> B)
{ new A (...);
 with (A -> C)
{ new A (...);
 new A (...);
 }
}
```

In this case the first and the last `new` expressions will create objects from class B. However the inner redefinition operator “hides” the external one; thus second `new` expression will use class C instead.

- A replaced class of the outer operator can be a replacing class of the inner operator:

```java
with (B -> C)
{ ..
 with (A -> B)
{ new A();
 }
 ..
}
```

In this case, inner redefinition instruction alone would create an object of class B, therefore in this case it is affected further by the outer operator, and will create an object of class C.

- A replacing class of the outer operator can be a replaced class of the inner operator:

```java
with (A -> B)
{ ..
 with (B -> C)
{ new A();
 }
 ..
}
```

In this case the inner operator will not have any effect on this initialization expression, and the object will be created from class B.

4. IMPLEMENTATION SCHEMA

In this section we present a translation from ImpliJava to Java which serves as a core of the prototype ImpliJava compiler, and can be also interpreted as a formulation of ImpliJava semantics. It is important to notice that this translation with insignificant modifications can be used to extend other languages with the implicit genericity feature.

The function transforming an ImpliJava program into a Java one for most of the fragments of the code is an identity function. The exceptions are listed below (and described in detail in further sections):

- For each non-final class, (since there does not exist any nontrivial redefinition possible for a final class), we define a `construction interface`.
- In each class, we define a `construction handler` static field object.
- We use a `construction handlers list`, storing actual class redefinitions with `construction handlers`.
- Every use of `with` operator is replaced with the registration of `construction handler`.
- Every new expression creating object from non-final class, is replaced with a call to `construction handler` looked up in the `construction handlers list`.
- Each class declared as inheriting directly from `Thread` class is modified to inherit from `ImpliThread` class.

4.1 Construction interface

For each non-final class `C1` we generate a declaration of public interface `C1_ConsInt`

For each constructor of `C1`, interface `C1_ConsInt` contains a declaration of method named `create`, with the same parameters and `throws` list as the constructor (see example on Fig. 4).

4.2 Construction handler

For each class `C1` we declare a local “handler” class (named `C1_cHandler`), and a static field (`cHandler`) used for storing the only instance of `C1_cHandler` class. The handler class (responsible for object construction) is declared to implement all the “construction interfaces” (see previous section) of the classes simulated by `C1`. In other words: it implements the “construction interface” of each ancestor of `C1`, which does not contain any constructor not supported by `C1`. Each non-final class implements also its own “construction interface”.

In the handler class, we generate one method named `create` for each constructor of class `C1`, with the same parameters and `throws` list as the corresponding constructor. Every such a method in handler class executes the corresponding constructor in original class and returns an instantiated object as a value. Example of such generated handler class can be seen on Fig. 4.

4.3 Construction handlers list

We declare `construction handlers list` (see Fig. 8), which stores actual class redefinitions. Those redefinitions are represented as a mapping from replaced classes, to stacks of construction handlers (see Sec. 4.2). On the top of each stack there is a handler for the most recent (therefore active) redefinition for the given class.

1For simplicity we assume that there does not exist `C1_ConsInt` declaration within the package of `C1`
public class C1 extends D {
    ...  
    public C1(int x, int y) { ...;}
    public C1(String s) throws IOException { ...;}
}

Figure 3: ImpliJava class C1 before the translation to Java

```
public interface C1_ConsInt  //construction
{ public C1 create (int x, int y) // interface
    public C1 create (String s) throws IOException //class
    }

class C1_cHandler implements C1_ConsInt, C2_ConsInt
{ public C1 create (int x, int y) //construction
        return new C1_cHandler(); // handler object
        return new C1(x, y); //handler
    public C1 create (String s) throws IOException //class
    { return new C1(s); } }

public class C1 extends D
{ public static Object cHandler // construction
        new C1_cHandler(); // handler object
    ...  
    public C1(int x, int y) { ...;}
    public C1(String s) throws IOException { ...;}
}
```

Figure 4: Result of the translation of ImpliJava class C1 from Fig. 3 to Java

Methods regHandler, and unregHandler are used to modify the current handlers list when entering and leaving the scope of redefinition operator. Method getHandler is used at the point of object creation, in order to find a valid handler for the given class.

4.4 The with instruction translation

Every use of redefinition operator of the form:

```
with (C->D)
{ I; }
```

is replaced with:

```
ImpliJava.Handlers.regHandler(C.class, D.class, D.cHandler);
try
{ I; }
finally
 ImpliJava.Handlers.unregHandler(C.class); }
```

4.5 The new expression translation

Each new expression of the form (where C is not a final class):

```
new C(\(\overline{e}\))
```

is replaced with the expression calling the create method of handler looked up in the construction handlers list:

```
((C_ConsInt) ImpliJava.Handlers.
getHandler(C.class, C.cHandler)).create(\(\overline{e}\))
```

The fact that the result of getHandler will be of type C_ConsInt (so the cast will not raise exception) is guaranteed by the condition, that in every redefinition operator, a replacing class simulates a replaced class (see Sec. 3.3).

4.6 The ImpliThread class definition

Each class extending directly java.lang.Thread is modified, to extend the ImpliJava.ImpliThread class instead. Class ImpliThread is defined as a subclass of classical Thread, which for each constructor of parent class, contains a declaration of corresponding constructor with the same sequence of parameters as in the original one. Each such a constructor contains two instructions: Firstly, it makes adequate super call. Then it performs the operation of copying the current class redefinitions of creating thread (stored in ImpliJava.Handlers.map) to this newly created thread. In this paper we skip the detailed source code of ImpliThread class, since it is rather obvious.

5. NOTES ON THE IMPLEMENTATION

The above defined semantics gave raise to the prototype implementation of the ImpliJava compiler. Such compiler works by translating an ImpliJava code into a Java one, and then executing the Java compiler. To optimize the speed of compiled code, we implemented some simple caching in the ImpliJava.Handlers class. Then we performed some tests, to estimate the time cost of “dynamic lookup of the initialization class” used in ImpliJava when compared with the statical object creation mechanism used in Java.

We started with the test consisting solely of creation of 5 million of simple objects (with no fields). To understand the amount of created objects, notice that this is roughly the maximum number of such objects Sun’s JVM heap can take (in default configuration). This test was performed in a constant environment with some redefinitions, which were not changed between creation of objects. Next, we performed similar test with little bigger (thus more realistic) objects, containing four int fields, yet still only a trivial constructor.

Another test, denoted as “interlacing”, was the “malicious” scenario in which, before every object creation we changed the environment. Thus, every even object was created in the range of one redefinition operator and every odd object in range of another one. As a result the caching mechanism was not effective. In the end we performed a test with chosen real-life classes: a not connected Socket, and Button (without placing it on any Window).

On Fig. 5 we present the results of the tests2, which show that object initialization in ImpliJava is few percent slower than in Java. Additionally, when interpreting these results, it is important to notice that:

- most of the cost of this mechanism is due to searching HashMaps containing redefinition information assigned to a class and a thread. This could be reduced if the JVM were modified to allow storing such redefinition information directly in the class definition structures;
- a typical program not only creates objects, but also performs computation via means of method execution. Furthermore, when the objects are created from re-defined classes, then the bodies of executed methods also change accordingly. This, however, works solely thanks to the original Java virtual method lookup mechanism, which is not affected by our approach. Therefore it does not suffer from any performance impact.

Therefore, we believe that in real-life cases, the actual cost of mechanism of implicit genericity is negligible. Thus, we

---

2 Results have been obtained on a single processor core PC machine (Celeron 2.8 Ghz with 1GB memory). All the presented execution times had been chosen as the best results of 5 tries of each test.
### Figure 5: ImpliJava vs Java performance comparison (time in ms)

<table>
<thead>
<tr>
<th>class</th>
<th>objects</th>
<th>Java time</th>
<th>ImpliJava time</th>
<th>Interlacing time</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 Fields</td>
<td>5 mln</td>
<td>1781</td>
<td>2000</td>
<td>12%</td>
<td>4172</td>
</tr>
<tr>
<td>4 Fields</td>
<td>5 mln</td>
<td>3563</td>
<td>3719</td>
<td>4.3%</td>
<td>6000</td>
</tr>
<tr>
<td>Socket</td>
<td>200 000</td>
<td>1843</td>
<td>1907</td>
<td>3.4%</td>
<td>2140</td>
</tr>
<tr>
<td>JButton</td>
<td>50 000</td>
<td>5375</td>
<td>5421</td>
<td>0.8%</td>
<td>5734</td>
</tr>
</tbody>
</table>

### Figure 6: ImpliJava code requiring explicit type casts

```java
public class ServiceLocator
{
    public Socket connect() {... return new Socket(...);}
}
```

```java
ServiceLocator loc = new ServiceLocator(...);
with (Socket -> MySSLSocket)
{
    MySSLSocket x = loc.connect(); // type error
    MySSLSocket x = (MySSLSocket) loc.connect(); // type OK
}
```

### 6. TYPE CHECKED IMPLIJAVA

Redefinition mechanism of ImpliJava gives a programmer the freedom to reuse any component in multiple ways, with various extensions of it’s behavior. However in some situations this approach can suffer from the lack of static type information about the change of behavior of given code executed in the context of redefinition operator.

To illustrate this problem, consider the code on Fig. 6. In this example, even though the method `connect` is executed within the range of redefinition operator, the compiler cannot ensure that returned object will be of the redefined class `MySSLSocket`. The compiler cannot assure it at the point of method call, since in general case, the returned object could have been created somewhere before, not during the execution of method `connect`, thus not affected by the redefinition operator. However, since in this case it will be in fact of class `MySSLSocket` and in many similar examples the result of a method will be of a redefined type, ImpliJava could benefit from such statical inference of a result type of the method. If the compiler had the possibility to verify the redefined result type of a method, then a programmer would not have to write type casts, and he/she would be sure about the actual type (and resulting behavior) of the method executed in the context of redefinition operator.

In order for the type-checker to be able to verify that the result of a method will be of the redefined type, it needs to be sure that the resulting object was created within the range of redefinition operator. And in order to verify this, the type-checker needs to know some temporal relations between different actions executed within the program.

In general, there is an infinite number of possible dependencies between the moment of creation of a method result, and another actions (like moments of execution of another methods, different values of parameters etc). However, we have defined two method properties which: (i) easy to verify; (ii) hold in practice for many methods; and (iii) under some conditions guarantee that the result of a method executed in scope of a redefinition operator will be of a redefined type. Mentioned temporal properties are:

- **R.M** The result of the given method is created during the execution of that method.
- **R.O** The result of the method is created during the lifetime of an object, on behalf of which that method is executed.

In general, the decision problem: “Do the above properties hold for a given method?” is undecidable. Nevertheless, there exist simple and decidable rules, which allow the verification of those properties in most cases. However, in order to specify those rules we first need to introduce the following additional temporal properties, which extend the above two properties to fields, variables and expressions:

- **F.O** A value of the given non-static field of an object is created during the lifetime of that object.
- **E.O** A value of the given expression is created during the lifetime of an object in which it is executed.
- **E.M** A value of the given expression placed inside the method is created during the execution of this method.
- **V.M** A value of the given variable declared in the method is created during the execution of that method.
- **V.O** A value of the given variable declared in the method is created during the lifetime of the object of the class in which this method is declared.

All the above temporal properties of methods, fields, expressions and variables can be verified using the below rules:

- **R.M/R.O** holds for a given method, if: (i) for every `return e` instruction in it `E.M/E.O` (respectively) holds for `e`; and (ii) `R.M/R.O` holds for all overriding bodies of this method in subclasses;
- **E.M** holds for an expression if it is either:
  - `null` or new `C (...)` expression;
  - a `V.M` variable;
  - an `e.m(f)` where `R.M` holds for `m`, or `R.O` holds for `m` and `E.M` for `e`;
- **E.O** holds for an expression if it is either:
  - `null`, `this` or new `C (...)` expression;
  - a `V.M` or `V.O` variable;
  - an `e.m(f)` where `R.M` holds for `m`, or `R.O` holds for `m` and `E.O` for `e`;
- **F.O** field can only be assigned within methods of the owning class with `E.O` expressions;
- **V.M/V.O** variable can only be assigned with `E.M/E.O` expression;

It is important to notice that above rules can be used to:

- (i) verify if the existing set of annotation of temporal properties is correct; but also to (ii) infer the set of properties for the given non-annotated program. However, since most of the oo languages use explicitly written types, we believe that it will be more natural if ImpliJava will use those rules to verify annotations written by a programmer. As a result, having the program with temporal annotations verified using the above rules, the typechecker can assure that within the range of given `with (C -> D)` operator:

  - the result of `o.m(...)` expression is of the type `D` if: (i) the result of the method `m` is of type `C`, and (ii) `R.M` holds for `m`;
  - the result of `o.m(...) / o.f` expression is of type `D` if: (i) `m/f` is of type `C`, (ii) `R.O/F.O` holds for `m/f`, (iii) value of `o` was created within the range of the given redefinition operator.
public class ServiceLocator
{
  Socket *fSocket;
  public Socket *getSocket()
  { return fSocket; }
}

public Socket *connect()
{ return new Socket(...); }

public ServiceLocator()
{ fSocket = connect(); }

...
ServiceLocator sl1 = new ServiceLocator();
{ ServiceLocator sl2 = new ServiceLocator();
  MySSLSocket s = sl1.connect(); //Type OK thanks to R.M
  MySSLSocket s = sl2.getSocket(); //Type OK thanks to R.O
}

Figure 7: ImplJava code with temporal annotations

To show how the type-checking process can benefit from the above rules, we present the example on Fig. 7 consisting of: (i) class ServiceLocator with temporal properties annotations (verified using the above rules), (ii) redefinition instruction which benefits from those annotations. In this example we use "*" symbol next to method/field/variable name to specify R.O, F.O, and V.O properties. At the same time, we use "#" to specify R.M and V.M properties.

As a result, thanks to richer method type signatures, the compiler can infer more specific types of method results or field dereferences placed within the scope of redefinition operators. Such results can be also achieved using the first-class generic declarations [3], however those would require to declare generic parameters in each method and to pass them explicitly at the method and constructor call.

Additionally, it is important to notice that the usage of temporal annotations is an option. Programs can be written without the use of such tools, or use them in chosen places only. Still the program will be able to perform the redefinitions -- the only consequence will be the need to use explicit typecasts in some situations.

7. POSSIBLE EXTENSIONS OF THE IMPLICIT POLYMORPHISM

7.1 Redefinitions with wider scope

The primary goal of our efforts was to create a tool which gives the programmer a very precise control over the scope of redefinitions. We believe that this is very important in many cases, however there are some scenarios in which we might need to use the same set of redefinitions in many different places within some part of application. This in turn would require from programmer to repeat the same redefinition operator in many methods. However, our approach can be easily extended to support the wider scope of redefinitions.

First of all, we can allow the programmer to place the redefinition operator in the header of class declaration: class C extends D with (G->H). This will be equivalent to putting "original" with (G->H) {...} operator around the body of each method of class C.

Secondly, we can also allow the programmer to place the redefinition operator in the header of the file, which will be equivalent to placing the same redefinition in the header of each class within this file (using the syntax defined above).

7.2 Extension to local classes

A natural extension of our approach is to permit not only to replace exactly the given class used in object creation expressions, but also to replace it in local subclass declarations (and similarly in an anonymous class declarations). In order to be able to do this safely, we would need to restrict the simulation relation, in a way in which it could assure, that for each subclass declaration, inheriting from class C, it still will be a type-safe declaration, when class C will be replaced with D.

However, to be able to ensure that any subclass declaration modified in the above way will be a proper Java subclass declaration, we additionally need to ensure that no conflict of declarations will occur in any such case. This in general means, that class D cannot declare any new identifiers (no new fields nor new methods), which heavily reduces a set of such simulating classes, and thus the usefulness of this approach.

This problem can be avoided, when the language does not suffer from accidental identifier clashes. Studies of solutions aimed at avoiding of the accidental clashes (together with study of the other approaches) can be found in [3, 21, 15]. However, the use of such extension requires a more detailed study of such solutions, which extend the subject of this paper.

8. FINAL REMARKS

In this paper we have presented a new approach to extend oo languages with the support of locally visible, unanticipated class redefinitions. Our approach can be seen, as a restricted subset of Context-Oriented programming technique, which supports the dynamic execution only at the point of object creation. However, we believe that it is a competitive solution since it is expressive, has a rather simple semantics, and does not require any design decision in advance in the original code.

On one side, our approach gives the programmer a fine grained control of when the redefinitions are used. Also, by separating the declaration of the redefinition (which is just a subclass declaration) and the usage of the redefinition, it allows easy and flexible reuse of redefinitions. On the other side, the semantics of how the class redefinition works is based on the classical notion of inheritance. Therefore only the notion of "when the redefinition is used" is newly-introduced. As a result, it seems that our approach can be easily learned by programmers familiar with mainstream oo languages. And in fact, by replacing the class used during the object instantiation, programmer can indirectly influence the whole execution of program, while still having the possibility to perform static analysis and formal reasoning about the program execution (see Sec. 6).

Furthermore, since our approach exploits inheritance, it does not need to redefine the method lookup algorithm, only the algorithm for the creation of new object. As a result in real-life cases the redefinition has a negligible impact on the execution speed (see Sec. 5). Therefore, we believe that the presented solution is a practical tool, which can be easily added as-it-is to existing languages.

9. REFERENCES

public final class Handlers
{
   public static Object getHandler
   (Class class_S, Object defaultH)
   {
      Object p = getHandlerInternal(class_S);
      if (p != null) return p;
      return defaultH;
   }
   public static void regHandler
   (Class class_S, Class class_D, Object handler)
   {
      Object h2 = getHandlerInternal(class_D);
      if (h2 != null) handler=h2;
      if (!map.containsKey(t) )
         map.put(t, new HashMap());
      if (!map.get(t).containsKey(class_S))
         map.get(t).put(class_S, new Stack());
      map.get(t).get(class_S).add(0,handler);
   }
   public static void unregHandler
   (Class class_S)
   {
      Stack s = tmap.get(class_S);
      if (s == null) return null;
      if (s.empty()) tmap.remove(class_S);
      s.pop();
   }
}

private static Object getHandlerInternal
   (Class class_S)
   {
      HashMap<Class, Stack> tmap=
         map.get(Thread.currentThread());
      Stack s = tmap.get(class_S);
      if (s.empty()) return null;
      return s.peek();
   }

public static Object getHandler
   (Class class_S, Object defaultH)
   {
      Object p = getHandlerInternal(class_S);
      if (p != null) return p;
      return defaultH;
   }

Figure 8: Java source code of the construction handlers list