# Generics and Reverse Generics for Pharo

Alexandre Bergel[1]* and Lorenzo Bettini[2]†
[1]*Pleiad Lab, Computer Science Department (DCC), University of Chile, Santiago, Chile*
[2]*Dipartimento di Informatica, Università di Torino, Torino, Italy*

Keywords: Generic Programming, Pharo, Dynamically Typed Languages.

Abstract: Generic programming is a mechanism for re-using code by abstracting specific types used in classes and programs. In this paper, we present a mechanism for adding generic programming in dynamically typed languages, showing how programmers can benefit from generic programming. Furthermore, we enhance the expressiveness of generic programming with *reverse generics*, a mechanism for automatically deriving new generic code starting from existing non-generic one. We implemented generics and reverse generics in Pharo Smalltalk, and we successfully used them to solve a problem of reusing unit test cases. This helped us to identify a number of bugs and anomalies in the stream class hierarchy.



## 1 INTRODUCTION

The notion of *generic programming* has been originally introduced in statically typed programming languages to ease manipulation and reuse of collection classes and algorithms. On the other hand, because of their flexible type systems, dynamically typed object-oriented languages have been left out of the scope of generic programming. The need for generic programming in a dynamically typed setting has been less prominent since no restriction applies over the kind of elements a collection may contain.

Furthermore, in a dynamically typed language like Smalltalk, where types are absent in declarations of linguistic entities (like methods, fields, local variables), it might look odd to talk about generic programming. However, there is still a crucial context where types (i.e., class names) appear statically: class references. When creating an object, the class name is hardcoded in the program, and this makes the object instantiation process hard to abstract from.

There are well-known patterns to deal with this problem, such as *Factory Method* (Gamma et al., 1995), *Dependency Injection* (Fowler, 2004), *Virtual classes* (Bracha et al., 2010) and ad-hoc linguistic constructs (Cohen and Gil, 2007). However, these mechanisms are effective when future extensions are foreseen. They provide little help in a scenario of unanticipated code evolution in which the programming language does provide dedicated evolutionary construct. This paper is about fixing this issue for dynamically typed languages using generics.

As popularized by mainstream statically typed programming languages, generic programming provides a mechanism for defining *template* classes where some types are variables/parameters and then for providing arguments for those type variables, thus *instantiating* template classes into concrete and complete classes. In the following, we then use the term template class to refer to a class where some types are parametrized; accordingly, we refer to a concrete/complete class when all the arguments for parametrized types are provided.

*Reverse generics* (Bergel and Bettini, 2011) is the dual mechanism that enables the definition of a template class from a complete class. We call this mechanism *generalization*. It allows obtaining a brand new generic class from an existing class by "removing" hardwired class references, and by replacing them with parametrized types. For instance,

```
G<T> = C>C'<
```

generates the new generic class `G<T>` starting from the class `C` by replacing each reference of a class `C'` contained in `C` with the type parameter `T` in `G`. It is the dual operation of the *instantiation* operation offered by generics. The generic `G` may be instantiated into `G<U>` for a provided class `U`. Note that, the reverse

*This author has been partially supported by Program U-INICIA 11/06 VID 2011, grant U-INICIA 11/06, University of Chile, and FONDECYT 1120094.

†This author was partially supported by MIUR (Project PRIN 2008 DISCO).

Bergel A. and Bettini L..
Generics and Reverse Generics for Pharo.
DOI: 10.5220/0004027503630372
In *Proceedings of the 7th International Conference on Software Paradigm Trends* (ICSOFT-2012), pages 363-372
ISBN: 978-989-8565-19-8

generics mechanism satisfies the property

C = (C>T<) <T>.

Finally, an important point is that the original class C remains unmodified. Indeed, reverse generics are useful under the basic assumptions that (i) the code to be reused has to be left intact (it cannot be the subject of refactoring) and (ii) the host programming does not implicitly support for looking up classes dynamically (as this is the case in most dynamically languages, except NewSpeak the supports virtual classes (Bracha et al., 2010)). In particular, we aim at providing, through our implementation of reverse generics, a *generative* approach, where new generic code is (automatically) generated starting from existing one, and the latter will not be modified at all; for this reason, reverse generics are not, and they do not aim at, a refactoring technique (we also refer to Section 7).

This paper extends the Pharo Smalltalk programming language with *generics* and *reverse generics*. We adapted the reverse generics to cope with the lack of static type information (in (Bergel and Bettini, 2011) reverse generics were studied in the context of statically typed languages such as Java and C++). Requirements on type parameters can be defined as a safety net for a sound instantiation; we provide mechanisms for structural and nominal requirements both for generics and reverse generics in Pharo.

The generic mechanisms we implemented do not depend on any Pharo facilities suggesting that generics and reverse generics are likely to be transposable to other dynamically typed languages. Although it has been realized in a dialect of Smalltalk, nothing prevents them from being applied to Ruby and Python. Even though similar mechanisms have been proposed in Groovy (Axelsen and Krogdahl, 2009), to the best of our knowledge, this is the first attempt to add a generic-like construct to Smalltalk. (The Groovy case is discussed in the related work section).

We employed reverse generics to face a classical code reuse problem. Unit tests in Pharo are inherited from Squeak, a Smalltalk dialect that served as a base for Pharo. Those tests have been written in a rather disorganized and ad-hoc fashion. This situation serves as the running example of this paper and was encountered when evolving the Pharo runtime. This helped us identify a number of bugs and anomalies in the stream class hierarchy.

The contributions and innovations of this paper are summarized as follows: **(i)** definition of a mechanism for generics in Pharo (Section 2); **(ii)** description of the reverse generics model in Pharo (Section 4); **(iii)** description of the implementation of both mechanisms (Section 5); **(iv)** applicability to a non trivial case study (Section 6). Section 7 summarizes the related work and Section 8 concludes the paper and gives some perspectives on future work.

## 2 GENERICS IN PHARO

This section presents a mechanism for generic programming for the Pharo/Smalltalk programming language[3]. The presentation of the mechanism is driven by a test-reuse scenario. We will first define a test called GCollectionTest. This test will be free from a particular class of the collection framework. GCollectionTest will be instantiated twice, for two different fixtures based on OrderedCollection and SortedCollection[4].

Consider the following code snippet containing a test that verifies elements addition.

```
"Creation of the class T"
GenericParameter subclass: #T

"Creation of the class GCollectionTest with a variable"
TestCase subclass: #GCollectionTest
  instanceVariableNames: 'collection'

"Definition of the setUp method"
"It instantiates T and add 3 numbers in it"
GCollectionTest>> setUp
  collection := T new.
  collection add: 4; add: 5; add: 10.

"Definition of the test method testAddition"
"It adds an element in the collection defined in setUp"
GCollectionTest>> testAddition
  | initialSize |
  initialSize := collection size.
  collection add: 20.
  self assert: (collection includes: 20).
  self assert: (collection size = (initialSize + 1)).
```

GCollectionTest is a pretty standard unit test in the spirit of the xUnit framework (most of the 115 classes that test the Pharo collection library follow a very similar structure). No reference to a collection class is made by GCollectionTest. The method setUp refers to the empty class T. GCollectionTest may be instantiated into OrderedCollectionTest and SortedCollectionTest as follows:

```
"Instantiate GCollectionTest and replace
 occurrences of T by OrderedCollection"
(GCollectionTest @ T-> OrderedCollection)
        as: #OrderedCollectionTest

"Replace T by SortedCollection"
(GCollectionTest @ T -> SortedCollection)
        as: #SortedCollectionTest
```

[3] http://www.pharo-project.org

[4] A fixture refers to the fixed state used as a baseline for tests. We consider the setUp method only in our situation.

The generic class GCollectionTest has been instantiated twice, each time assigning a different class to the parameter T. We adopted the convention of defining generic parameter as subclasses of GenericParameter. This convention has a number of advantages, as discussed in Section 5. Since GCollectionTest contains references to T, it is a generic class. There is therefore no syntactic distinction between a class and a generic class. GCollectionTest is a generic class only because T is a generic parameter and T is referenced in setUp.

Pharo has been extended to support the (... @ ... -> ...) as: ... construct. These three operators defines the life cycle of a generic in Pharo.

Compared to the Java generics mechanism, generics for Pharo operates on class references instead of types. A class provided as parameter may be freely instantiated, as in the example above. Generics in Pharo are similar to a macro mechanism. In that sense, it shares similarities with C++ templates but using a dynamically type stance.

# 3 REQUIREMENTS FOR GENERIC PARAMETERS

In order for a generic class to be instantiated, a class needs to be provided for each generic parameter. To prevent generic instantiation to be ill-founded, requirements for a generic parameter may be declared. These requirements are enforced when a generic class is instantiated. Requirements are formulated along nominal and structural definitions of the base code.

**Nominal Requirements.** Static relationship between types may be verified when instantiating a generic class. In the example above, T must be a subtype of Collection[5]. This is specified by defining a method requirements that returns myself inheritsFrom: Collection:

```
T>> requirements
  ^(myself inheritsFrom: Collection)
```

In that case, instantiation of GCollectionTest raises an error if a class that is not a subclass of Collection is provided as parameter.

Note that we introduced the myself pseudo variable. This variable will be bound to the class provided as the generic parameter when being instantiated. The variable self, which references the receiver object, cannot be used within requirements.

[5]We use the following convention: a class is a type when considered at compile time, and it is an object factory at runtime.

**Structural Requirements.** In addition to nominal requirements, a generic parameter may be also structurally constrained. A constraint is satisfied based on the presence of some particular methods. In the example above, a method check may return

```
myself includesSelectors: {#add: . #includes: . #size}
```

In that case, only a class that implements the method add:, includes:, and size can be provided in place of T.

We express a requirement as a boolean expression. The keyword inheritsFrom: and includesSelectors: are predicates. They may therefore be combined using boolean logic operators. For instance, we can express all the above requirements as follows:

```
T>> requirements
  ^(myself inheritsFrom: Collection)
      and: [myself includesSelectors:
                {#add: . #includes: . #size}]
```

Dynamically typed languages favor sophisticated debugging and testing sessions over static source code verification. The lack of static type annotation makes any isolated check on a generic not feasible. Completeness of T's requirements cannot be verified by the compiler, thus, it is up to the programmers to provide a set of satisfactory requirements when defining generic parameters. In practice, this has not been a source of difficulties.

# 4 REVERSE GENERICS IN PHARO

This section presents the reverse generics mechanism in Pharo; we will use a scenario that consists of reusing unit tests. Consider the following class WriteStreamTest taken from an earlier version of Pharo:

```
ClassTestCase subclass: #WriteStreamTest

WriteStreamTest >> testIsEmpty
  | stream |
  stream := WriteStream on: String new.
  self assert: stream isEmpty.
  stream nextPut: $a.
  self deny: stream isEmpty.
  stream reset.
  self deny: stream isEmpty.
```

The class WriteStreamTest is defined as a subclass of ClassTestCase, itself a subclass of SUnit's TestCase. WriteStreamTest defines the method testIsEmpty, which checks that a new instance of WriteStream is empty (i.e., answers true when isEmpty is sent). When the character $a is added into the stream, it is not empty anymore. And resetting a stream moves the stream

pointer at the beginning of the stream, without removing its contents. WriteStreamTest has 5 other similar methods that verify the protocol of WriteStream.

We consider that most of the important features of WriteStream are well tested. However, WriteStream has 27 subclasses, which did not receive the same attention in terms of testing. Only 3 of these 27 classes have dedicated tests (FileStream, ReadWriteStream and MultiByteFileStream). Manually scrutinizing these 3 classes reveals that the features tested are different than the one tested in WriteStreamTest[6].

The remaining 24 subclasses of WriteStream are either not tested, or indirectly tested. An example of an indirect testing: CompressedSourceStream is a subclass of WriteStream for which the feature of WriteStream are not tested. CompressedSourceStream is essentially used by the file system with FileDirectory, which is tested in FileDirectoryTest.

The situation may be summarized as follows: WriteStream is properly tested and has 22 subclasses, but none of these subclasses have the features defined in WriteStream tested for their particular class.

This situation has been addressed by refactoring the collection framework using TraitTest (Ducasse et al., 2009). We make a different assumption here: the base system must be preserved, which implies that a refactoring is not desirable. Refactoring may have some implications on the overall behavior, especially in terms of robustness and efficiency. It has been shown that inheritance is not that helpful in this situation (Flatt and Felleisen, 1998; Bergel et al., 2005).

With our implementation of reverse generics in Pharo, a generic class GStreamTest can be obtained from the class WriteStreamTest by turning all references of WriteStream into a parameter that we name T.

```
Generic
  named: #GStreamTest
  for: WriteStream -> T @ WriteStreamTest
```

Following a Java-like syntax (Bergel and Bettini, 2011), the above code corresponds to the following reverse generic definition:

```
class GStreamTest<T> = WriteStreamTest>WriteStream<
```

The generic GStreamTest is defined as a copy of WriteStreamTest for which all references to WriteStream have been replaced by the type T introduced in the previous section (Section 2). GStreamTest may now be instantiated by replacing all references of WriteStream with untested subclasses of WriteStream as illustrated in Section 2:

[6]According to our experience, this is a general pattern. Often programmers focus essentially on testing added methods and variable when subclassing.

```
"Instantiate GStreamTest and replace occurrences of T
 by ZipWriteStream"
(GStreamTest @ T-> ZipWriteStream)
        as: #ZipWriteStreamTest

"Replace T by HtmlFileStream"
(GStreamTest @ T -> HtmlFileStream)
        as: #HtmlFileStreamTest
```

Figure 1 summarizes the generalization and instantiation of the WriteStreamTest example. Reverse generic targets class instantiation and sending messages to a class.

The above scenario could be solved by having a super abstract class in which the class to be tested is returned by a method. This method could then be overridden in subclasses (*factory method* design pattern (Gamma et al., 1995)). However, this solution is not always the best approach: First, tests of the collection libraries cannot be optimally organized using single inheritance (Ducasse et al., 2009). Second, the code to be reused may not always be editable and modifiable. This is often a desired property to minimize ripple effects across packages versions.

## 4.1 Requirements when Generalizing

We have previously seen that requirements may be defined on generic parameters (Section 3). These requirements equally apply when generalizing a class. Turning references of WriteStream into a parameter T may be constrained with the following requirements:

```
T>> requirements
  ^(myself inheritsFrom: Stream)
  and: [ myself includesSelectors: {#isEmpty . #reset} ]
```

Further requirements could be that the parameter T understands the class-side message on:, and the instance-side message nextPut:. However, this will be redundant with the requirement myself inheritsFrom: Stream, since Stream defines the method nextPut: and on:.

Requirements may also be set for class methods, e.g.,myself class includesSelector: {#new: } makes the presence of the class method new: mandatory.

## 4.2 Capturing Inherited Methods

Instantiating a generic G, which is obtained from generalizing a class C, makes copies of C with connections to different classes. This process may also copy superclasses of C when methods defined in superclasses need to have new references of classes. This situation is illustrated in Figure 2.

A different example is adopted in this figure. The class AbstractFactory has an abstract method create.

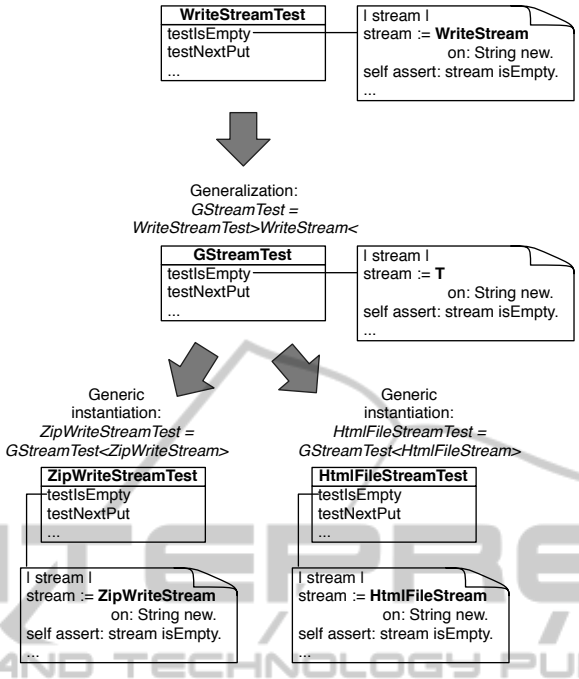


Figure 1: Reusing WriteStreamTest.

PointFactory is a subclass of it that creates instances of Point (not represented on the figure). This class is subclassed into EnhPointFactory that overrides create to count the number of instances that have been created.

Consider the generic

GEnhFactory<T> = EnhPointFactory>Point<.

This generic may be instantiated with a class Car to produce cars instead of points:

CarFactory = GEnhFactory<Car>.

The class Point is referenced by the superclass of EnhPointFactory. Generalizing and instantiating EnhPointFactory has to turn the Point reference contained in PointFactory into Car. This is realized in reverse generics by automatically copying also the superclass into a new generic class with a generated name.

The class inheritance is copied until the point in the hierarchy where no superclass references a generic parameter.

## 5 IMPLEMENTATION

The homogeneity of Pharo and in general of most of Smalltalk dialects greatly eases the manipulation of a program structural elements such as classes and methods. In Smalltalk, classes and methods are first-class entities. They can be manipulated as any object. A compiled method is a set of bytecode instructions with an array of literals. This array contains all references to classes being used by this compiled method (Goldberg and Robson, 1983).

Instantiating a generic is made by copying a class, assigning a different name, and adjusting the array of literals with a different set of class bindings. An example of this procedure is depicted in Figure 3.

A number of design decisions were made:

- The Pharo syntax has not been modified. This has the great advantage of not impacting the current development and source code management tools. This is possible since classes are first-class objects in Pharo.
- The Smalltalk meta-object protocol has not been extended. Again, this decision was made to limit the impact on the development tools. As a consequence, there is no distinction between a generic and a class, thus the generic mechanism can be implemented as a simple library to load.

Indeed these design choices are based also on past experience in Smalltalk extensions: the last significant change of the language was realized in 2004 (Lienhard, 2004), when traits have been introduced in Squeak, the predecessor of Pharo. In the current version of Pharo, the support of traits is fragile at best (bugs are remaining and many tools are not traits aware). This experience gained with traits suggests that realizing a major change in the programming language is challenging and extremely resource consuming.

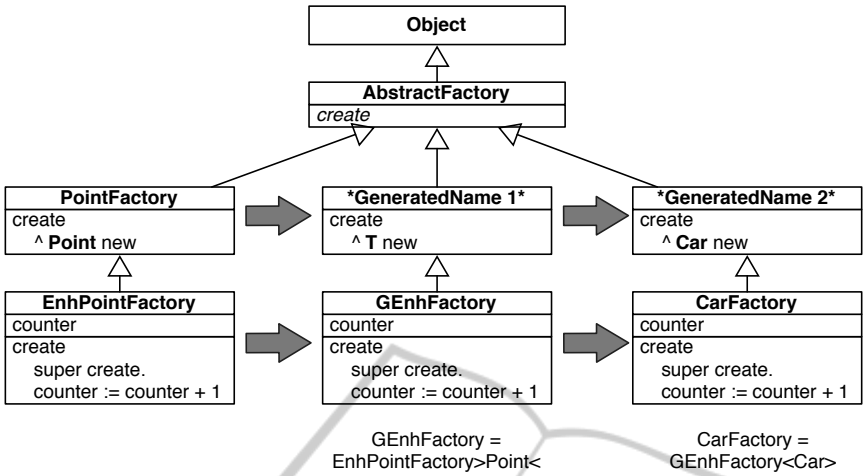


Figure 2: Copying superclasses, illustrated with a generic class factory.



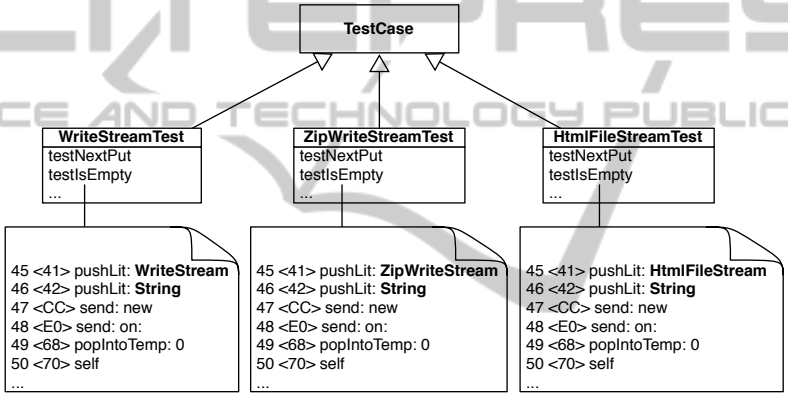


Figure 3: Final results when ZipWriteStreamTest and HtmlFileStreamTest have been produced.

Note that, by using our reverse generics, one can modify only the original existing code (i.e., the classes that are not generic), and then, automatically, spread the modifications to the one obtained by reverse generics.

The implementation presented in this paper is freely available (under the MIT license) at http://www.squeaksource.com/ReverseGeneric.html.

# 6 CASE STUDY: APPLICATION TO THE PHARO STREAM HIERARCHY

The situation described in Section 4 is an excerpt of the case study we realized. For each of the 24 subclasses of WriteStream, we instantiated GStreamTest. This way, about 24 new unit tests were generated. The WriteStreamTest class defines 6 test methods. We therefore generated 24 * 6 = 144 test methods. Each of the generated test is a subclass of ClassTestCase, which itself defines 3 test methods. Running these 24 unit tests executes 144 + 27 * 3 = 225 test methods.

Running these 225 test methods results in: 225 runs, 192 passed, 21 failures, 12 errors. Since the 6 tests in WriteStreamTest pass, this result essentially says that there are some functionalities that are verified for WriteStream, but they are not verified for some of its subclasses. An example of the culprit test methods for the failures are CrLfFileStreamTest>> testNew and LimitedWriteStreamTest>> testSetToEnd. The fact that these two tests fail uncovers some bugs in the classes CrLfFileStream and LimitedWriteStream.

The body of CrLfFileStreamTest>> testNew is

```
self should: [CrLfFileStream new] raise: Error
```

meaning that a CrLfFileStream should not be instantiated with new. However, the class can actually be instantiated with new, resulting in a meaningless and unusable object.

Another example of a bug was found in Limited-

WriteStream. This class is used to limit the amount of data to be written in a stream. The body of LimitedWriteStreamTest>> testSetToEnd is:

```
LimitedWriteStreamTest>> testSetToEnd
  | string stream |
  string := 'hello'.
  stream := LimitedWriteStream with: ''.
  stream nextPutAll: string.
  self assert: stream position = string size.
  stream setToEnd.
  self assert: stream position = string size.
  self assert: stream contents = string
```

It essentially verifies the behavior of the stream index cursor. This test signals an error in the expression stream nextPutAll: string. By inspecting what triggered the error, we discovered that when a LimitedWriteStream is instantiated with with: '', the object is initialized with a nil value as the limit, resulting in a meaningless comparison (newEnd > limit in the method LimitedWriteStream>> nextPutAll:).

Not all the test methods that fail and raise an error are due to some bugs in the stream class hierarchy. We roughly estimate that only 11 test methods of these 33 methods have uncovered tangible bugs. The remaining failures and errors are due to some differences on how class should be initialized. For example, the test StandardFileStreamTest>> testSetToEnd raises an error because a StandardFileStream cannot be instantiated with the message with: (it is instantiated with fileNamed:, which requires a file name as argument). Although no bug have been located, this erroneous test method suggests that the method write: should be canceled (i.e., raise an explicit error saying it should be not invoked).

This experiment has a number of contributions:

- it demonstrates the applicability of our generics and reverse generics to a non-trivial scenario,
- it helped us identify a number of bugs and anomalies in the Pharo stream hierarchy.

# 7 RELATED WORK

When Java generics were designed, one of the main intent was to have the backward compatibility with the existing Java collection classes. The enabling mechanism is that all the generic type parameters must be "erased" after the compilation (*type erasure* model (Odersky and Wadler, 1997; Bracha et al., 1998)). Therefore, all the run-time type information about parametrized types are completely lost after the compilation, thus making impossible to execute all the operations which require run-time types, such as, e.g., object instantiations. This limits the expressiveness of Java generics (Allen and Cartwright, 2002): for instance, if T is a generic type, the code T x = new T() is not valid.

For these reasons, the generic type system of Java cannot be considered "first-class", since generic types cannot appear in any context where standard types can appear (Allen et al., 2003).

On the contrary, the generic programming mechanisms provided by C++ do not suffer from all these issues. In particular, the C++ compiler generates a different separate copy for each generic class instantiated with specific types (and the typechecking is performed on the instantiated code, not on the generic one). Therefore, while in Java a Collection<String> and a Collection<Integer> would basically refer to the same class (i.e., the type erased class Collection), in C++ they would refer to two separate classes, where all the type information remains available. Therefore, in C++, all the operations which require run-time types are still available in generic classes, and hence the C++ type generic system can be considered "first-class" (notably, C++ templates were formalized and proved type safe (Siek and Taha, 2006)). For instance, if T is a generic type, the code T *x = new T() in C++ is perfectly legal, since C++ *templates* are similar to a macro expansion mechanism[7]. We refer to Ghosh (Ghosh, 2004) and Batov's work (Batov, 2004) for a broader comparison between Java generics and C++ templates.

In order for generic types to be used and type checked in a generic class, those types must be constrained with some type requirements. Constraints on generic types are often referred to as *concepts* (Kapur et al., 1981; Austern, 1998). Java generics require explicit constraints, thus a concept is defined using a Java interface or a base class, and a type satisfies a concept if it implements that specific interface or it extends that specific base class. On the contrary, the C++ compiler itself infers type constraints on templates and automatically checks whether they are satisfied when such generic type is instantiated. In our implementation, generic parameters can be assigned constraints using nominal (similarly to Java) and structural requirements (similarly to concepts), as illustrated in Section 3.

In dynamically typed languages, like Smalltalk, where types are not used in declarations, the context where generics are useful is in object instantia-

[7] Actually, C++ templates are much more than that: (partial) specialization of templates is one of the main features that enables computation at compile time, often referred to as *template metaprogramming* (Abrahams and Gurtovoy, 2004).

tion; thus, with this respect, the generics presented in this paper are related to C++ templates, rather than to Java generics. The generics needed in the context of Smalltalk act at a meta-level, by generating new classes starting from existing ones, thus, they have similarities with *generative programming* mechanisms (Eisenecker and Czarnecki, 2000) and C++ meta programming (Abrahams and Gurtovoy, 2004). This meta programming mechanism is evident also in our generics and reverse generics implementation in Pharo: new code is generated starting from existing one, without modifying the latter. This takes place in two steps: with reverse generics a brand new generic version is obtained starting from existing code; then, by instantiating generic classes, the generic code is adapted and reused in a new context.

There seem to be similarities among reverse generics and some refactoring approaches: however, the intent of reverse generics is not to perform reverse engineering or refactoring of existing code, (see, e.g., (Duggan, 1999; Dincklage and Diwan, 2004; Kiezun et al., 2007)) but to extrapolate possible generic "template" code from existing one, and reuse it for generating new code. Note that this programming methodology will permit modifying only the original existing code, and then, automatically, spread the modifications to the one obtained by reverse generics.

A first attempt to automatically extract generic class definitions from an existing library has been conveyed by Duggan (Duggan, 1999), well before the introduction of generics into Java. Besides the reverse engineering aspect, Duggan's work diverges from reverse generics regarding downcast insertion and parameter instantiation. Duggan makes use of *dynamic subtype constraint* that inserts runtime downcasts. A parametrized type may be instantiated, which requires some type-checking rules for the creation of an object: the actual type arguments must satisfy the upper bounds of the formal type parameters in the class type.

Kiezun et al. propose a type-constraints-based algorithm for converting non-generic libraries to add type parameters (Kiezun et al., 2007). It handles the full Java language and preserves backward compatibility. It is capable of inferring wildcard types and introducing type parameters for mutually-dependent classes. Reverse engineering approaches ensure that a library conversion preserves the original behavior of the legacy code. This is a natural intent since such a conversion is exploited as a refactoring. Instead, the purpose of reverse generics is to replace static types references contained in existing classes with specialized ones and then to produce a brand new class.

A limitation of first-order parametric polymorphism is that it is not possible to abstract over a type constructor. For instance, in List<T>, List is a type constructor, since, given an argument for T, e.g., Integer, it builds a new type, i.e., List<Integer>. However, the type constructor List itself is not abstracted. Therefore, one cannot pass a type constructor as a type argument to another type constructor. Template template parameters[8] (Weiss and Simonis, 2001) in C++ provides a means to abstract over type constructors. Moors, Piessens and Odersky (Moors et al., 2008) extended the Scala language (Odersky et al., 2008) with type construction polymorphism to allow type constructors as type parameters. Therefore, it is possible not only to abstract over a type, but also over a type constructor; for instance, a class can be parametrized over Container[T][9], where Container is a type constructor which is itself abstracted and can be instantiated with the actual collection, e.g., List or Stack, which are type constructors themselves. The generics mechanism presented in this paper acts at the same level of first-order parametric polymorphism, thus, it shares the same limitations. An interesting extension would be to be able to switch to the higher level of type constructor polymorphism, but this is an issue that still needs to be investigated.

The *Dependency Injection* pattern (Fowler, 2004) is used to "inject" actual implementation classes into a class hierarchy in a consistent way. This is useful when classes delegate specific functionalities to other classes: messages are simply forwarded to the object referenced in a field. These fields will have as type an interface (or a base class); then, these fields will be instantiated with derived classes implementing those interfaces. This way the actual behavior is abstracted, but we need to tackle the problem of "injecting" the actual implementation classes: we do not have the implementation classes' names hardcoded in the code of the classes that will use them, but we need to initialize those classes somewhere. Moreover, we need to make sure that, if we switch the implementation classes, we will do that consistently throughout the code. Typically this can be done with *factory method* and *abstract factory* patterns (Gamma et al., 1995), but with *dependency injection frameworks* it is easier to keep the desired consistency, and the programmer needs to write less code. The reverse generics mechanism is not related to object composition and delegation, i.e., the typical context of the *inversion of control* philosophy that dependency injection tries to deal with. With reverse generics the programmer does not have to design classes according the pattern of abstracting the actual behavior and then delegate it to factory meth-

[8]The repetition of "template" is not a mistake.

[9]Scala uses [] instead of <>.

ods; on the contrary the reverse generics mechanism allows generating new code (i.e., new classes) from existing one, without modifying the original code.

Package Template (Sørensen et al., 2010) is a mechanism for reusing and adapting packages by rebinding class references. A version has been proposed for Groovy (Axelsen and Krogdahl, 2009). Package Template offer sophisticated composition mechanisms, including class renaming and merging. The reverse generics mechanism is able to turn a non generic class into a generic one, while Package Template is not designed for this purpose.

Traits (Ducasse et al., 2006) were introduced in the dynamically-typed class-based language Squeak/Smalltalk to counter the problems of class-based inheritance with respect to code reuse. Although both traits and generic programming aim at code reuse, their main contexts are different: traits provide reuse by sharing methods across classes (in a much more reusable way than standard class-based inheritance), while generic programming (and also our generics) provides a mechanism to abstract from the type implementing specific behavior. Combining our generic mechanism with traits looks promising in that respect, also for the meta-programming features of traits themselves (Reppy and Turon, 2007).

## 8 CONCLUSIONS

The mechanisms presented in this paper provide features both to write generic code in a dynamically typed language and to extrapolate possible generic "template" code from existing one, and reuse it for generating new code. In our approach, class generalization and generic instantiation is based on class copying, similarly to C++ templates. Although this implies some code duplication in the generated code, this is consistent with the meta-level which is typical of *generative programming* mechanisms (Eisenecker and Czarnecki, 2000).

Since highly parametrized software is harder to understand (Gamma et al., 1995), we may think of a programming methodology where a specific class is developed and tested in a non-generic way, and then it is available to the users via its "reversed" generic version (in this case, we really need the non generic version for testing purposes, so the code must not be refactored). Therefore, reverse generics can be used as a development methodology, not only as a way to turn previous classes into generic: one can develop, debug and test a class with all the types instantiated, and then expose to the "external world" the generic version created through reverse generics.

A limitation of the implementation presented in this paper is that the generic parameters (like `T` in Section 2 and Section 4) are global subclasses, thus there can be only one such generic parameter (together with its requirements, Section 3 and Section 4.1). However, in this first prototype implementation of generics and reverse generics in Pharo, this did not prevent us from using these mechanisms to class hierarchies (like the case study of Section 6) and to study their applicability. Of course, in future versions, we will deal with this issue, and remove the "globality" of generic parameters.

At the best of our knowledge, no generic (and reverse generic) programming language construct is available in Smalltalk, Ruby and Python that achieve the same capabilities as we presented in this paper. It is subject of future work to further investigate whether our proposal can be applied to other dynamically typed languages.

## REFERENCES


Abrahams, D. and Gurtovoy, A. (2004). *C++ Template Metaprogramming: Concepts, Tools and Techniques from Boost and Beyond*. Addison-Wesley.

Allen, E., Bannet, J., and Cartwright, R. (2003). A First-Class Approach to Genericity. In *OOPSLA*, pages 96–114. ACM.

Allen, E. and Cartwright, R. (2002). The Case for Run-time Types in Generic Java. In *PPPJ*, pages 19–24. ACM.

Austern, M. H. (1998). *Generic Programming and the STL: using and extending the C++ Standard Template Library*. Addison-Wesley.

Axelsen, E. W. and Krogdahl, S. (2009). Groovy package templates: supporting reuse and runtime adaption of class hierarchies. In *DLS*, pages 15–26. ACM.

Batov, V. (2004). Java generics and C++ templates. *C/C++ Users Journal*, 22(7):16–21.

Bergel, A. and Bettini, L. (2011). Reverse Generics: Parametrization after the Fact. In *Software and Data Technologies*, volume 50 of *CCIS*, pages 107–123. Springer.

Bergel, A., Ducasse, S., and Nierstrasz, O. (2005). Classbox/J: Controlling the Scope of Change in Java. In *OOPSLA*, pages 177–189. ACM.

Bracha, G., Odersky, M., Stoutamire, D., and Wadler, P. (1998). Making the future safe for the past: adding genericity to the Java programming language. In *OOPSLA*, pages 183–200. ACM.

Bracha, G., von der Ahé, P., Bykov, V., Kashai, Y., Maddox, W., and Miranda, E. (2010). Modules as Objects in Newspeak. In *ECOOP*, pages 405–428. Springer.

Cohen, T. and Gil, J. (2007). Better Construction with Factories. *JOT*, 6(6):103–123.

Dincklage, D. V. and Diwan, A. (2004). Converting Java classes to use generics. In *OOPSLA*, pages 1–14. ACM.

Ducasse, S., Nierstrasz, O., Schärli, N., Wuyts, R., and Black, A. P. (2006). Traits: A Mechanism for fine-grained Reuse. *ACM TOPLAS*, 28(2):331–388.

Ducasse, S., Pollet, D., Bergel, A., and Cassou, D. (2009). Reusing and Composing Tests with Traits. In *TOOLS*, volume 33 of *LNBIP*, pages 252–271. Springer.

Duggan, D. (1999). Modular type-based reverse engineering of parameterized types in Java code. In *OOPSLA*, pages 97–113. ACM.

Eisenecker, U. W. and Czarnecki, K. (2000). *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley.

Flatt, M. and Felleisen, M. (1998). Units: Cool Modules for HOT Languages. In *PLDI*, pages 236–248. ACM.

Fowler, M. (2004). Inversion of Control Containers and the Dependency Injection pattern. http://www.martinfowler.com/articles/injection.html.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Ghosh, D. (2004). Generics in Java and C++: a comparative model. *ACM SIGPLAN Notices*, 39(5):40–47.

Goldberg, A. and Robson, D. (1983). *Smalltalk 80: the Language and its Implementation*. Addison-Wesley.

Kapur, D., Musser, D. R., and Stepanov, A. A. (1981). Tecton: A Language for Manipulating Generic Objects. In *Program Specification*, volume 134 of *LNCS*, pages 402–414. Springer.

Kiezun, A., Ernst, M. D., Tip, F., and Fuhrer, R. M. (2007). Refactoring for Parameterizing Java Classes. In *ICSE*, pages 437–446. IEEE.

Lienhard, A. (2004). Bootstrapping Traits. Master's thesis, University of Bern.

Moors, A., Piessens, F., and Odersky, M. (2008). Generics of a higher kind. In *OOPSLA*, pages 423–438. ACM.

Odersky, M., Spoon, L., and Venners, B. (2008). *Programming in Scala*. Artima.

Odersky, M. and Wadler, P. (1997). Pizza into Java: Translating theory into practice. In *POPL*, pages 146–159. ACM.

Reppy, J. and Turon, A. (2007). Metaprogramming with Traits. In *ECOOP*, volume 4609 of *LNCS*, pages 373–398. Springer.

Siek, J. and Taha, W. (2006). A semantic analysis of C++ templates. In *ECOOP*, volume 4067 of *LNCS*, pages 304–327. Springer.

Sørensen, F., Axelsen, E. W., and Krogdahl, S. (2010). Reuse and combination with package templates. In *MASPEGHI*, pages 1–5. ACM.

Weiss, R. and Simonis, V. (2001). Exploring template template parameters. In *Perspectives of System Informatics*, volume 2244 of *LNCS*, pages 500–510. Springer.