DOI: 10.15439/2024F3128
ISSN 2300-5963 ACSIS, Vol. 39

Evolving the Enterprise Software Systems Landscape: Towards Anti-Patterns in Smalltalk-to-Java Code Transformation

Marek Bělohoubek and Robert Pergl
Faculty of Information Technology, Czech Technical University in Prague, Prague, Czech Republic
Email: {marek.belohoubek,robert.pergl}@fit.cvut.cz
http://ccmi.fit.cvut.cz

***Abstract*—In the rapidly evolving landscape of enterprise software systems, there is a marked escalation in the proliferation of new technologies, tools, languages, and methodologies daily. These innovations are pivotal not only for the development of new systems but also for the maintenance and augmentation of existing infrastructures. Consequently, it is imperative to devise systems that are responsive to these advancements, fostering the integration of novel tools and methodologies into the current systems. This integration often necessitates mechanisms for transforming source code across diverse programming languages. In the course of developing a transformation tool from Smalltalk to Java, we encountered several code patterns that significantly impede the transformation process. This paper aims to elucidate one such transformation anti-pattern. We provide a comprehensive overview, a formal delineation, illustrations derived from actual code, and propose refactoring strategies for both Smalltalk and Java environments.**

I. Introduction

The EVOLUTION of software systems within enterprises is a perpetual and intricate process. Each day ushers in a profusion of fresh technologies, tools, programming languages, and methodologies aimed at facilitating the development of novel systems while also helping in the maintenance and expansion of existing ones [1].

The significance of maintenance and expansion cannot be overstated, primarily because as these systems mature, there emerges a necessity to adapt to updated requirements, thereby compelling the integration of new libraries, the adoption of the latest technologies, and, in extreme cases, the complete restructuring of the underlying architecture.

As current research suggests, the frequency of such modifications is increasing [2] it becomes essential to develop new systems that take into account these modifications [3] and create tools and methodologies to help incorporate this design philosophy into existing systems, for example by helping to transform source code between different programming languages.

Over a year ago, we began researching the transformation of Smalltalk code to Java, based on previous research by Engelbrecht [4], with the aim of creating tools and methodologies following these three principles:

- Transformation process should require as little manual input as possible.
- The resulting code should be easy to read and edit manually, to make future expansion and maintenance as simple as possible.
- Tools need to provide option to manually override (sub)result of any step in the process, and allow inclusion of these changes during repeated processing of the input code.

Throughout the process of implementation and subsequent testing on real-world code, we encountered numerous instances where our methods failed to effectively transform certain code snippets.

Upon closer examination of these snippets, we identified recurring code patterns that, while perfectly valid in Smalltalk, lacked direct equivalents in Java suitable for automated translation without compromising the original functionality or excessively cluttering the resulting code.

Internally, we have coined the term "Transformation anti-patterns" to describe these phenomena. In this paper, we aim to elucidate one such anti-pattern by furnishing a comprehensive description, formal definition, real code-based example, and refactoring solutions applicable to both the Smalltalk and Java contexts.

II. Languages

First, we need to provide the reader with additional information about the languages and design of our tool.

A. Smalltalk

Smalltalk, which emerged in the 1970s as Smalltalk-72, is a dynamically typed programming language that is purely object-oriented. It was first made available to the public with the release of Smalltalk-80 in the 1980s [5].

Smalltalk is distributed in the form of an image along with its own development environment and virtual machine. Consequently, every application developed in Smalltalk necessitates the execution of the associated image.

Various versions of Smalltalk (called dialects) exist, each with its own interpretation of the Smalltalk virtual machine. These include proprietary systems such as Visual Works [6], as well as open-source initiatives like Squeak [7] and Pharo [8].

B. Java

Java, created in the 1990s and launched in 1996 as Java 1.0, is a statically typed, object-oriented, high-level programming language [9].

A crucial component of Java is the Java Virtual Machine (JVM) [10], which facilitates the running of Java applications on various computer architectures, regardless of the platform used. This abstraction shields programmers from platform-specific intricacies.

C. Language differences

There are three main distinctions between Smalltalk and Java that complicate the conversion process to the extent that we refer to it as transformation.

Firstly, they vary in their handling of typing and types. Java follows a static typing approach, requiring all variable types, method arguments, and return values to be defined before compilation. In contrast, Smalltalk is dynamically typed, meaning type checks are performed only during run-time.

The second differentiation lies in the contrast between Smalltalk's meta-classes and Java's class features. In Smalltalk, each object holds a reference to its metaclass, which stores class-specific methods and variables. Conversely, Java uses the static keyword to define class features, which do not pertain to an underlying object and thus cannot be accessed through reference.

Finally, whereas all types in Smalltalk are subclasses of Object, Java includes eight primitive types that solely contain values without any Object-like characteristics (such as methods, initialization, etc.).

These disparities necessitate transformation tools to either deduce all types used in the code for integration into statically typed Java, and/or implement an additional framework to mimic Smalltalk's dynamic behavior.

III. SMALLTALK TO JAVA TRANSFORMATION TOOL

This section consists of two parts: general overview of the transformation process and a more in-depth look at the translation step.

A. Transformation process

Our vision for the tool primarily revolves around facilitating the transition of the business logic from legacy Smalltalk systems into contemporary object-oriented languages.

These systems often grapple with challenges related to maintaining comprehensive and accurate documentation due to extensive years of ongoing maintenance and subsequent development. Given that the tool necessitates parsing the source code of the system for transformation purposes, it can concurrently generate a class model of the system as an ancillary outcome.

Considering that the transformation process will likely entail substantial modifications to the underlying architecture and technologies, particularly concerning the user interface (UI), it becomes imperative to support only partial transformation of the system.

In light of these requirements, our approach involves the development of a suite of tools designed to execute the transformation through the following sequential steps:

- **Scope definition** User defines the transformation scope by specifying which parts (bundles, packages, classes) of the system should be transformed.
- **Type inferring** Transformation tool then uses a real-time type inferrer [11] to obtain all types used in the selected scope.
- **Translation** The scope is then transformed class-by-class, method-by-method, using previously inferred types. The source code is translated from Smalltalk to Java, so the result not only works like the original but also looks as close as possible to the original.
- **Model generation** The transformation tool then produces a model of the transformed scope, in the form of a UML with a transformation profile applied to it.
- **Java generation** Finally, the generation tool uses the transformation model to generate the package structure and Java classes out of it.

B. Translation step

As mentioned previously, our goal is to make a set of tools and methodologies to help programmers migrate applications and systems from Smalltalk to Java with the expectation of further maintenance and development performed within the transformed code.

Therefore, such tools have to be capable of producing human-readable and editable code, which limits usage of certain methods that deal with the discrepancy between dynamically typed Smalltalk and statically typed Java in elegant ways, but produce code that is very hard to maintain and expand.

One such method described in Mr. Engelbrecht's work [4] suggests using a single class named `SmalltalkObject` as an ancestor to all classes in the transformed system.

This class then defines a dummy definition for every method in the original system with all inputs and outputs replaced by `SmalltalkObject` itself. Concrete classes override their own methods with specific implementations.

With this clever use of class-based reflection, this method effectively simulates Smalltalk behaviour in Java, but at the cost of extreme inconvenience for the future expansions of the transformed code, since every class implements or inherits every single method defined in the transformed code.

C. Translation tool

Our implementation of the tool responsible for the translation step treats each method / class definition as a separate code snippet, with the only connection to the rest of the transformed scope being inferred types.

We use the abstract syntax tree generated by Smalltalk compiler to transform methods node-by-node, in essence doing word-by-word literal translation.This is combined with usage of the inferred types to produce being human-like strongly typed code in Java.

To better demonstrate here is an example of simple Smalltalk method:

```
sum: aFirst and: aSecond
  ^ aFirst + aSecond.
```

Assuming that we have previously inferred that `aFirst`, `aSecond` and their sum are all typed `Integer`, we will get the following translation:

```
public Integer sumAnd(Integer aFirst,
  Integer aSecond){
    return aFirst + aSecond;
}
```

As demonstrated in the example above, the translated code uses strong typing and for simpler methods looks like it was written by a human.

Unfortunatelly during our testing on real-world application we have discovered several code patterns that prevent fully automated translation, either because there is no direct strongly typed equivalent to the original weakly typed code, or because the automatic solution would introduce severe complications to readability and extensibility of the code.

We have named such code structures "Transformation anti-patterns".

IV. Transformation anti-patterns

We define transformation anti-patterns as code structures that satisfy all of the following conditions:

- The code structure is syntactically and semantically valid in Smalltalk, capable of being compiled and executed without any errors.
- Translating the code structure directly from Smalltalk to Java would yield Java code that either fails to compile or exhibits considerable divergence in runtime behavior when compared to the original Smalltalk implementation.

Virtually all anti-patterns identified thus far originate from the semantic discrepancies between the two languages, frequently stemming from the contrasting principles underlying dynamic and static typing.

This phenomenon can be likened to the challenges inherent in translating between natural human languages. For example, the literal translation of a joke that relies on clever wordplay and phonetic similarity in one language will almost invariably fail to convey its original humour in another language.

Therefore, it is imperative to detect such occurrences during the translation process and notify the user, advising them to review the outcome, and, if possible, offering guidance on resolving any issues that may arise.

Our investigation into the transformation from Smalltalk to Java necessitates the identification of suspicious code structures, their categorisation into anti-patterns, and the provision of multiple strategies for addressing them.

The following section presents an example of one such anti-pattern, identified during experiments on a real-world Smalltalk system (regrettably, direct examples of the code cannot be provided due to an existing non-disclosure agreement).

While the example anti-pattern isn't the most complicated one, it demonstrates the basic principles behind anti-patterns very well and we have found it occurring quite frequently in the analysed code.

V. Ancestor defined variable

The employment of inheritance and class hierarchy is primarily motivated by the opportunity to establish methods and variables within a high-level class and subsequently allow all its subclasses to inherit these attributes, thereby adhering to the "Don't Repeat Yourself" (DRY) principle.

This practice is prevalent in both Smalltalk and Java, typically manifesting in the form of abstract classes or Java interfaces. However, disparities arise in its utilization between the two languages due to the distinction between dynamic and static typing.

In dynamically typed languages like Smalltalk, programmers can define variables and methods in ancestor classes with a general type and then utilize specialized types in subclasses instead.

While this approach is also feasible in strongly typed languages like Java, it necessitates explicit casting each time the programmer accesses the actual value or return value of a method from the ancestor class, to align with the specialized type in the subclass.

This casting can be implemented either by incorporating casts wherever the variables/methods from the ancestor class are utilized, or by overriding pertinent methods with new implementations that invoke their parent counterparts and then cast the result to the appropriate type.

However, both of these practices are suboptimal: the former clutters the code with type casts, while the latter violates the DRY principle by necessitating the reimplementation of methods.

Alternatively, programmers can leverage Java generics to define problematic types in the ancestor class and defer the specification of concrete types to its descendants. While this approach typically functions effectively, it encounters two limitations.

Firstly, descendants are only considered polymorphic if they all employ identical concrete types. Secondly, the concrete types must share at least one common ancestor or implement the same interface to be usable in both the ancestor class and the concrete classes, assuming the ancestor class doesn't solely define simplistic get/set methods.

While employing disparate types in ancestor and descendant classes is generally discouraged, even in dynamically typed languages, ancestor classes following this pattern are typically abstract, serving as "interfaces with partial implementation," either explicitly designated as such or never instantiated within the program itself.

VI. Example

Let's suppose following scenario. We are translating three classes:

- `Ancestor` - which defines an instance variable called `property` and implements its accessor.
- `DescendantString` - subclass of `Ancestor`, initializes it's instance variable with empty `String`.
- `DescendantInt` - subclass of `Ancestor`, initializes it's instance variable with `int`.

Here is code snipped with their Smalltalk implementations:

```
Object subclass: #Ancestor
instanceVariableNames: 'property.'

Ancestor>>property
    ^property.

Ancestor subclass: #DescendantString

DescendantString>>initialize
    property := ''.

Ancestor subclass: #DescendantInt.

DescendantInt>>initialize
    property := 0.
```

Transforming the code above directly is certainly possible (see the end of this subsection), but since each of the `Ancestors` subclasses initializes `property` with a completely different type, it will be defined as `Object`.

Therefore, even if the user wants to work directly with one of the subclasses, he will be forced to use type casting for all but the simplest operations with `property`.

```
public class Ancestor {

  protected Object property;

  public Object getPropertyValue(){
    return property;
  }
}

public class DescendantString
    extends Ancestor{

  public DescendantString(){
    property = "";
  }
}

public class DescendantInt
    extends Ancestor{

  public DescendantInt(){
    property = 0;
  }
}
```

VII. Refactoring solution

There are two main solutions for refactoring that coincide with the place where they occur: Ancestor abstraction (in Smalltalk) and Ancestor parameterisation (in Java).

A. *Ancestor abstraction*

Move all problematic variables and method implementations from the ancestor to its direct subclasses, making both the ancestor and it's problematic methods abstract.

```
Ancestor subclass: #DescendantString
    instanceVariableNames: 'property.'

DescendantString>>property
    ^property.

DescendantString>>initialize
    property := ''.

Ancestor subclass: #DescendantInt
   instanceVariableNames: 'property'.

DescendantInt>>property
    ^property.

DescendantInt>>initialize
    property := 0.
```

The code above shows the changes in Smalltalk code and the translated result is bellow.

```
public abstract class Ancestor {
    public abstract Object getPropertyValue();
}

public classDescendantString
    extends Ancestor{

  protected String property;

  public DescendantString(){
    property = new String();
  }

  public String getPropertyValue(){
    return property;
  }
}

public class DescendantInt
    extends Ancestor{

  protected Integer property;

  public DescendantInt(){
    property = new Integer();
  }

  public Integer getPropertyValue(){
    return property;
  }
}
```

This approach breaks the DRY principle, but the refactoring can be done in the source system (before the transformation to the statically typed language). It also keeps the polymorphism intact.

B. Ancestor parameterisation

Change the problematic ancestor into parametric class (for example, in Java: `Ancestor <T> class`) and define the correct concrete type in the generalisation itself (to continue with our previous Java example: `Descendant class extends Ancestor<concreteType>`).

```
public abstract class Ancestor <T>{

  protected T property;

  public T getPropertyValue(){
    return property;
  }
}

public class DescendantString
    extends Ancestor<String>{

  public DescendantString(){
    property = "";
  }
}

public class DescendantInt
    extends Ancestor<Integer>{

  public DescendantInt(){
    property = 0;
  }
}
```

This approach follows the DRY principle, but moves all of the refactoring to the target side. Worse yet, this approach can break the principle of substitution depending on the target system.

For example, in Java only descendants that have their concrete type matching the currently used ancestor will be allowed for substitution (this is due to the Java generics implementation) [12].

Lastly, this solution can be applied only to instance side methods, because static methods cannot be parameterized.

C. Comparison

The main differences between the two proposed solutions lie in the side of refactoring (source or target) and if the solution breaks or follows the DRY principle.

During our experiments, we have found that *Ancestor abstraction* is the most commonly used, as it provides consistency during repeated transformations (by making all the changes in the source) and keeps the principle of substitution intact, both of which greatly outweigh breaking the DRY principle.

However, it is necessary to always keep the context in mind as there are many cases where *Ancestor parametrisation* leads to better results.

VIII. Related work

The foundational principles behind transformation anti-patterns are not unique to the transition from Smalltalk to Java [13]. Instead, they are commonly observed across transformations from weakly-typed to strongly-typed languages.

Historically, programmers have grappled with challenges arising from the fundamental differences between the source and target programming languages. Examples include the translation of nested routines from Pascal to C [14], addressing structured programming constraints in the translation from Fortran to C [15], and, most critically, issues related to differences in typing systems, such as those encountered in the transition from Python to Java [16].

Furthermore, the development of transpliers—translating compilers—is an essential area of exploration. Some transpliers are specialized for specific languages, like those for C-to-Rust [17]. However, there is also burgeoning research in the realm of multilingual transpliers [18] [19].

This discussion would be incomplete without acknowledging advancements in AI translation tools. ORIGIN-Transcoder [20] employs a neural-based algorithm to reduce the need for manual input in the translation process. Although its applicability to translations from Smalltalk remains unproven, there is significant interest in the potential role of generative AI in the translation process [21].

IX. Conclusion

In this article, we have introduced the concept of transformation anti-patterns and shown an example of one of them.

Anti-patterns themselves will not always cause problems, but they represent severe complications for maintenance and future expansions of the translated code.

Based on this analysis, we believe that there is a space for further research into this topic, with many more anti-patterns yet to be discovered.

Acknowledgment

This research was supported by the grant of Czech Technical University in Prague No. SGS23/206/OHK3/3T/18.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

Statement on the use of AI AI technologies (Writefull and ChatGPT) were used solely to improve the language of the paper.

References

[1] M. Hilbert and P. López, "The world's technological capacity to store, communicate, and compute information," *science*, vol. 332, no. 6025, pp. 60–65, 2011.
[2] R. Kurzweil, *The Law of Accelerating Returns*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 381–416.

[3] O. Dvořák and R. Pergl, "Tackling rapid technology changes by applying enterprise engineering theories," *Science of Computer Programming*, vol. 215, p. 102747, 2022.

[4] R. L. Engelbrecht *et al.*, "Implementing a smalltalk to java translator," Ph.D. dissertation, University of Pretoria, 2006.

[5] A. C. Kay, *The Early History of Smalltalk*. New York, NY, USA: Association for Computing Machinery, 1996, p. 511–598.

[6] "Custom Software Application Development Services - Cincom VisualWorks® | Cincom Smalltalk®," 9 2023. [Online]. Available: https://www.cincomsmalltalk.com/main/products/visualworks/

[7] Squeak.org, "Squeak/Smalltalk." [Online]. Available: https://squeak.org/

[8] "Pharo - Welcome to Pharo!" [Online]. Available: https://pharo.org/

[9] I. Cosmina, *An Introduction to Java and Its History*. Berkeley, CA: Apress, 2022, pp. 1–31.

[10] F. Yellin and T. Lindholm, "The java virtual machine specification," 1996.

[11] J. Blizničenko and R. Pergl, "Generating UML Models with Inferred Types from Pharo Code," in *International Workshop on Smalltalk Technologies*, Koln, Germany, Aug. 2019. [Online]. Available: https://hal.science/hal-04053497

[12] M. Naftalin and P. Wadler, *Java Generics and Collections: Speed Up the Java Development Process*. O'Reilly Media, 2006. [Online]. Available: https://books.google.cz/books?id=zaoK0Z2STlkC

[13] M. Bělohoubek and R. Pergl, "The state of smalltalk to java transformation: Approaches review," in *World Conference on Information Systems and Technologies*. Springer, 2024, pp. 235–241.

[14] N. Sundaresan, "Translation of nested pascal routines to c," *ACM Sigplan Notices*, vol. 25, no. 5, pp. 69–81, 1990.

[15] D. S. Higgins, "A structured fortran translator," *ACM SIGPLAN Notices*, vol. 10, no. 2, pp. 42–48, 1975.

[16] E. Jin and Y. Sun, "An algorithm-adaptive source code converter to automate the translation from python to java," *JLPEA*, 2020.

[17] L. Xia, B. Hua, and Z. Peng, "An empirical study of c to rust transpilers," *School of Software Engineering, University of Science and Technology of China, and Suzhou Institute for Advanced Research, University of Science and Technology of China-04/27*, 2023.

[18] F. Bertolotti, W. Cazzola, and L. Favalli, "$*$ piler: Compilers in search of compilations," *Journal of Systems and Software*, vol. 212, p. 112006, 2024.

[19] F. Bertolotti, W. Cazzola, D. Ostuni, and C. Castoldi, "When the dragons defeat the knight: Basilisk an architectural pattern for platform and language independent development," *Journal of Systems and Software*, vol. 215, p. 112088, 2024.

[20] V. Rajathi, M. Harishankar, J. S. DS *et al.*, "Origin-the transcoder," in *2022 1st International Conference on Computational Science and Technology (ICCST)*. IEEE, 2022, pp. 179–182.

[21] J. D. Weisz, M. Muller, S. I. Ross, F. Martinez, S. Houde, M. Agarwal, K. Talamadupula, and J. T. Richards, "Better together? an evaluation of ai-supported code translation," in *Proceedings of the 27th International Conference on Intelligent User Interfaces*, 2022, pp. 369–391.