On the Difficulties of Raising the Level of Abstraction and Facilitating Reuse in Software Modelling: The Case for Signature Extension

Matthias Schöttle and Jörg Kienzle
School of Computer Science
McGill University
Montréal, Canada
Email: matthias.schoettle@mail.mcgill.ca, joerg.kienzle@mcgill.ca

Abstract—**Reuse is central to improving the software development process, increasing software quality and decreasing time-to-market. Hence it is of paramount importance that modelling languages provide features that enable the specification and modularization of reusable artefacts, as well as their subsequent reuse. In this paper we outline several difficulties caused by the finality of method signatures that make it hard to specify and use reusable artefacts encapsulating several variants. The difficulties are illustrated with a running example. To evaluate whether these difficulties can be observed at the programming level, we report on an empirical study conducted on the *Java Platform API* as well as present workarounds used in various programming languages to deal with the rigid nature of signatures. Finally, we outline signature extension as an approach to overcome these problems at the modelling level.**

I. Introduction

Complex systems are rarely built from scratch. To improve productivity and achieve higher quality during software development, it is common practice to rely on the existence of reusable artefacts. Reuse of artefacts comes in different flavours [14]. *Planned reuse* [7] refers to the situation where: 1) a *recurring development issue* has been identified, 2) *one or several solutions* to this issue have been developed, and 3) the software artefacts (e.g., documentation, models (if any) and code) realizing the solutions are *packaged in a reusable unit* and made available for reuse. At the programming level, reusable frameworks and libraries are in widespread use.

The philosophy of model-driven engineering (MDE) is that during development high-level specification models of a system are refined or combined with other models to include more solution details, such as the chosen architecture, data structures, algorithms, and finally even platform and execution environment-specific properties. Reuse in MDE is achieved through (domain-specific) modelling languages, which capture the essential concepts relevant to the development of the software at a given level of abstraction, and through model transformations that assist developers in transitioning from one layer of abstraction to another towards a concrete solution and implementation. To be effective in this framework, models that represent the system at a given level of abstraction need to be generic enough to allow for (ideally many) possible solution-specific refinements of the system at lower levels. This is even more true for models that are meant to be reusable.

Interfaces have been effectively applied at the programming level—but more recently also at the modelling level—to enable reuse within and across abstraction levels during software development [12]. This paper reflects on the challenges that developers face when defining interfaces for higher levels of abstraction or for units encompassing multiple solution variants. In particular, we concentrate on the problems caused by the *finality of signature declarations*.

The remainder of the paper is structured as follows. Section II reviews the software development roles in the context of reuse, as well as interfaces and signatures. Section III elaborates on the difficulties caused by final signatures by means of examples. Section IV presents an investigation of the *Java Platform API* to highlight these difficulties in a state-of-the-art programming language library as well as a discussion about workarounds used in various programming languages. Finally, Section V advocates the need for augmenting programming and modelling languages with constructs that allow signatures to be extended, and outlines how such constructs could be defined for programming and modelling languages alike.

II. On Reuse, Interfaces and Signatures

In the context of reuse, at least two clearly distinct software development roles arise. The *designer* of the reusable unit is an expert of the domain of the development issue that the unit addresses. She has a deep understanding of the nature of the issue, is able to identify variations of the problem and can therefore potentially identify user-relevant variations or *features*. Because the designer elaborates and implements the solution artefacts, she knows the exact implementation details, their properties and qualities, and the trade-offs that she decided to make. However, the designer does not know in what contexts and how exactly the reusable unit may be used in the future. Therefore, in addition to realizing the solutions, the designer strives to make the reusable unit as versatile and generic as possible, so that the solutions can be applied in a wide variety of reuse contexts. This might again involve coming up with multiple, functionally equivalent, yet different variants of realizations in terms of qualities and non-functional properties, e.g., varying memory footprint or performance. Finally, the designer needs to modularize and package the reusable unit to make it available to others, e.g., in form of a library or framework. There is no doubt that building a reusable unit is a challenging, non-trivial, time consuming task for the *designer*.

A *user* of a reusable unit on the other hand is an expert of the application he is developing. He is aware of the specific requirements of the system he is working on. At some point, the user might become aware that the software needs to deal with a specific development issue for which an existing reusable unit is available. The user knows very little about the complexity of the recurring development issue, and even less about the implementation details of different solutions to the issue offered by the reusable unit. To make reuse possible and safe, the user needs to be able to determine whether the reusable unit is applicable to their system. He has to be able to determine which solution, in case the reusable unit offers more than one, is most appropriate for the specific application context. He needs to customize the reusable unit to his specific reuse context, and then must use the reusable unit correctly.

Experience has shown that reuse of artefacts with explicitly defined *interfaces* leads to high levels of reuse maturity [12]. Interfaces specify a contract that bridges the worlds of the designer of a reusable unit and the (hopefully many) users of the reusable unit. Furthermore, applying the information hiding principles [17], interfaces make it possible to hide solution complexity and properties within a reusable unit, and hence significantly reduce the complexity that the users of the reusable unit need to deal with.

A very common way of providing a static interface that allows the users to trigger functionality provided by a reusable unit is an *operation signature* (or *service signature*). A signature is made of the operation *name*, of a *set of parameters*, each one comprised of a formal *name* and *type*, as well as the *type of value* returned by the operation, if any. Finally, some modelling or programming languages also include in an operation signature the set of exception types that might be raised at runtime when the operation is invoked.

Most statically compiled modelling or programming languages require signatures to always be specified in their entirety. This forces the designer of a reusable unit to decide on the exact number[1] and type of every parameter of an operation before she can declare an interface or signature for it. Once declared, the signature is *set in stone*, i.e., the existing parameters are immutable, and no new parameters can be added. It is of course possible to *overload* methods, i.e., declare new methods with the same name and additional parameters, but then the API contains *several* methods.

While this might be sometimes appropriate, the *finality* of signature declarations poses difficulties to the designer and user of a reusable model in certain situations. These situations are summarized here and then elaborated further in section III:

1) When a reusable unit encapsulates several solutions, it is sometimes difficult for the designer to come up with a common final signature that works for all of the solutions. The situation is similar when a modeller needs to define a signature in a model at a level of abstraction that allows for different solution refinements.
2) With final signatures, it can be difficult for the designer to add a new feature to a reusable unit in a non-intrusive way. The situation is similar for modellers who want to add a new feature to a model of an existing product line.
3) When signatures are used to define callback interfaces that allow a reusable unit to trigger reuse-context-specific functionality, it can be difficult for the designer to define a final callback signature that is ideal for any reuse context. The situation is similar in reusable models where the reusable behaviour has to trigger reuse context specific behaviour.
4) When a programmer or modeller designs a reusable unit x and reuses reusable unit y with different variations, it is sometimes difficult to make a final decision, i.e., which variant from y to use, since the reuse context of x is unknown. As a result, it is difficult to define a final signature for the functionality offered by x, and it is difficult to specify behaviour in x that calls operations of y if the variants in y have different signatures.

III. Problematic Situations

A. Difficulties Defining a Common Interface for Alternative Implementations

When designing a reusable unit with several variations for a common purpose, a designer generally aims at providing a common interface to the user that is independent of the concrete variation being used by the user. This strategy is highly beneficial, because it allows the user to maintain a design that stays at a high level of abstraction without depending on a concrete variation. A common interface makes it possible for the user of a reusable unit encapsulating multiple solution variants to replace a chosen variation with another one exhibiting different qualities without significant effort. In model-driven design a similar situation occurs when the designer uses abstraction to delay deciding on solution-specific details. An interface at a given level of abstraction makes it possible to explore different solution-specific refinements during development. Unfortunately, when signatures are final once they are declared, it is difficult to provide a common interface in situations where different implementations of an operation achieving the same functionality require different parameters to execute.

For example, *collections* are reusable units that are used very frequently, and they come in many variants offering different functionality and exhibiting different non-functional properties. In programming languages, collections are typically grouped together so that they can be treated in a similar

[1]Some statically compiled languages support the declaration of signatures with an arbitrary number of parameters. For instance, in Java with the *varargs* feature [16], or in Go [25] and Python [19] with the *variadic function* feature.

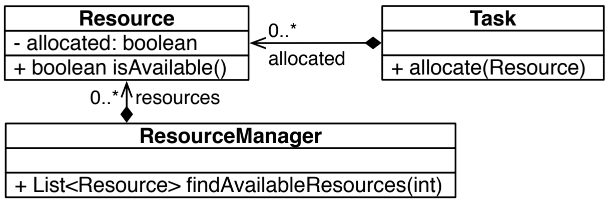

Figure 1. Resource Management Base Design

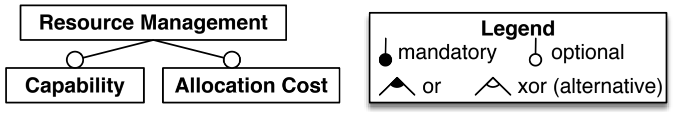

Figure 2. Resource Management Feature Model

way at a high level of abstraction. *Java* and *C#*, for instance, use inheritance to group different kinds of collections and algorithms to process them.

But defining a common interface for all kinds of collections is difficult. For example, the signature for adding an element to a plain collection is typically `add(Element)`, whereas adding an element to a map is provided by an operation with the signature `add(Key, Element)`. This can be problematic if a user wants to treat maps and collections in a uniform way, e.g., to check whether a collection/map contains a certain element.

B. Difficulties Adding New Functionality

Consider *Resource Management*, which is a recurring functionality required in many applications. At its core there are *Resources* which can be allocated to *Tasks* (signatures `Resource.isAvailable()` and `Task.allocate(Resource)`), and a *ResourceManager* class provides operations to find and allocate a number of resources to a given task (`findAvailableResources(int)`). A corresponding class diagram is shown in Figure 1.

Some applications might need additional functionality, which can be seen as additional features of the *Resource Management* unit. Figure 2 shows a feature model [11] with two optional variants. *Capability* provides the ability to differentiate resources according to their capabilities, and *Allocation Cost* augments the behaviour of resource allocation to consider individual resource allocation cost.

Generally, APIs are *set in stone* once published. The general recommendation is therefore to put a lot of effort into initial API design [3]. However, APIs need to evolve to accommodate changing requirements and integration of alternative solutions. In the case where the additional or optional functionality needs to execute together with existing functionality *and needs additional information from the user to process,* the situation is problematic. Since the signature of the operation providing the existing functionality is final, it is difficult to add additional parameters to it, and doing so would invalidate every place where the operation is being used.

For example, the optional feature *Capability* extends resources with capabilities. Such functionality might be required by a crisis management system (CMS) that needs to allocate workers to missions, and differentiate the workers according to their capabilities, such as *driver*, *fire fighter*, *first aid provider*, etc. In this case, a resource has an associated set of capabilities. Additionally, resources are allocated to a task because they fulfill a needed capability. Therefore, in order to look for available resources, additional information is required from the user: the desired capability for which resources are sought for has to be specified. Hence, some of the operation signatures of the reusable unit would need an additional `Capability` parameter: `findAvailableResources(int, Capability)` of `ResourceManager`, `allocate(Resource, Capability)` of `Task`, and `isAvailable(Capability)` of `Resource`.

When signatures are final, though, it is impossible to provide a clean common interface to the user of *Resource Management* that can be used at a high level of abstraction, i.e., with and without the optional *Capability* feature. In programming, optional functionality of a method is often triggered by parameters at the end of the signature. The behaviour of the method checks the value of the parameter, and if the user passes in a specific value, e.g., `false` or `null`, the optional functionality is not executed. If the language has support for default values, the designer can specify `false` or `null` as the default value, which relieves the user from needing to do so. However, the user still needs to consider these parameters, understand their intent and make a decision on whether to use them or not.

In our example, the designer of *Resource Management* is forced to provide a common `findAvailableResources(int)` method that provides the general ability to find available resources, *as well as* an additional overloaded `findAvailableResources(int, Capability)` method to support the optional feature. Alternatively, the designer can choose to define only one operation `findAvailableResources(int, Capability)`, and specify in the documentation of *Resource Management* that users who do not want to use the capability feature must pass `null` as an argument when invoking the operation at run time.

Neither of the workarounds are ideal, though. The former is error-prone, because users who have made the decision to use capabilities should only call the operations that have the *Capability* parameter. If by mistake they invoke one of the operations that does not handle capabilities, the consistency of resource management is jeopardized. The second workaround is at the least confusing, because users who do not want to use capabilities must upon every operation invocation pass `null` as a value.

C. Difficulties Providing a Callback Interface that fits all Reuse Contexts

Many frameworks take over the flow of control of an application and use the *callback* technique to execute application code when needed. This requires the designer of the framework to specify *at design time* an interface that defines the operation signature of the callback method that the framework will call when it wants to hand control to the application. However, the designer does not know in which contexts his reusable unit is going to be reused, and therefore has difficulties defining a final signature for the callback that will work in all contexts.

For example, the *Allocation Cost* feature (see Figure 2) augments *Resource Management* with the functionality `estimateTotalAllocationCost(Set<Resources>)` that can determine the total cost of allocation given a set of resources. Since the allocation cost of an individual resource typically depends on the state of the resource and since the designer of *Resource Management* does not know anything about the state of the actual resources, the behaviour that calculates the individual allocation cost of a resource needs to be provided by the user of *Resource Management* and invoked though a callback. To this aim, the designer defines a parameterless callback signature `estimateAllocationCost()` that needs to be implemented for the `Resource` class by the user.

Unfortunately, this is problematic when *Resource Management* is reused in the context of the CMS where workers are allocated to missions. Missions typically are performed in a certain region or location. When allocating a specific worker to a mission, the allocation cost depends on the distance between the worker's current location and where the mission is taking place. The current location of the worker—the resource—is part of the state of the worker, and accessible in the callback method. Unfortunately, the location of the mission—the task—is not accessible. Possible steps the designer can take to anticipate this problem is to include a dummy `Object` parameter in the callback. The user then has to create a class that subclasses `Object` with attributes to hold the desired application-specific state. Additionally, the user then has to tell the reusable unit at runtime which concrete instance of this new class to pass to the callback. This does work, but is very cumbersome.

D. Difficulties Delaying Design Decisions

An additional difficult situation arises in the context of software product line development (SPL) [18] and reuse hierarchies. Software product line development is an approach that is beneficial when developing a collection of similar software systems—a family of products—that share some commonalities and differ in a well-defined set of features exposed by the SPL. To increase reuse, functionality that has been identified as common is encapsulated within shared software artefacts that are reused within multiple products. Unless the shared functionality is absolutely identical for all products, it is again difficult for the designer of the shared reusable unit to define an interface that satisfies the needs of each individual product. A specific product or feature might require a slightly different variant of the functionality encapsulated within the reusable unit, which in turn could require additional information to process. This resembles the difficult situations explained in sections III-A and III-B. In a way, the design decision of which concrete product to produce in an SPL is delayed until the SPL is configured with the desired set of features. It is only at that time that it is absolutely clear what information is needed to process some shared functionality.

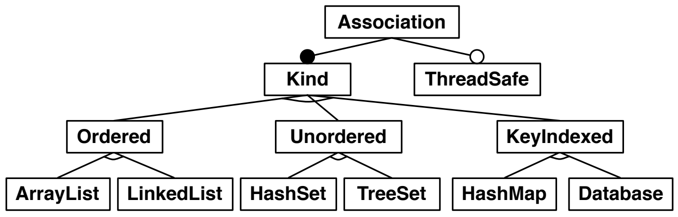

Figure 3. Association Feature Model

This situation is even more prominent when a designer of a reusable unit wants to reuse (within his own design) some other reusable unit that offers different variations. For example, in [2] the authors describe the design of a reusable unit *Association* encapsulating various association designs based on different collection types with different features: support for unique elements, multi-threading, enforcement of minimum and maximum amounts of elements, etc. A feature model of such a reusable unit is shown in Figure 3. Internally, it uses Java collection types to implement the different association features, as suggested by the names of the leaf features. However, it would be possible to use collection types of another object-oriented programming language instead.

The designer of *Resource Management* can use the *Association* reusable unit to keep track of one or more resources that are allocated to a task. The designer of *Resource Management* could reuse any of the variants provided by *Association* to realize this relationship. However, each variant exhibits different properties, in particular with respect to non-functional properties, e.g., performance or memory usage. Since *Resource Management* is a reusable unit itself, it is impossible for the designer of *Resource Management*, who now also is the user of *Association*, to determine the most appropriate alternative to reuse, since this decision depends on the context in which *Resource Management* will be reused in the future. Ideally, the designer would like to delay the decision [13] of which specific variant of *Association* to reuse and complete her design using a common, high-level interface.

IV. Validating the Need for Flexible Signatures

A. Exploring the Java Platform API

This section presents an empirical study that we conducted to demonstrate the potential usefulness of signature extension. Java ships with an extensive runtime library of reusable classes providing different kinds of commonly needed functionality. We examined the Java 10.0.1 runtime, and focussed our attention on the *java.base* module, which contains 5746 classes, of which 3245 are in the `java` and `javax` root packages.

The Java programming language has evolved since version 1.0 (released in 1996) and contains evolution information in the source code. We extracted the following information from the source code of the *java.base* module (version 10.0.1) for each public method of public classes:

- the method's signature
- whether the method overloads another method
- whether the method is marked as *deprecated*, which is the Java way of saying that a method is outdated and should not be used anymore. If available, we also extracted since which version the method is deprecated[2], and from the Javadoc's `@deprecated` tag the comment explaining the deprecation, and whether the method was replaced with other alternative methods
- in which version the method was introduced (provided using Javadoc's `@since` tag)[3]
- whether the method implements/overrides another (from a superclass/interface)

To accomplish this task, we used the *AST Parser* of *Eclipse JDT* and parsed each source file contained in the corresponding source code file of the *java.base* module. We only considered the main type declaration of the java file and hence ignored any additionally defined inner classes. In total we found 1104 public classes and 11720 public methods. The following tables present the gathered information.

Table I
GATHERED DATA OF THE *java.base* MODULE

Module	**java.base**
Number of public classes	1104
Number of public methods	11720
Number of overloaded methods	4072
Number of methods overloaded in subclasses	41
Number of deprecated methods	138
Number of overloaded methods that are deprecated	36

As Table I shows, close to 35% of all the public methods offered by the *java.base* module are overloaded. This is a considerable number. 41 methods are overloaded in subclasses, which is an indication of the presence of alternate features that need additional or potentially a different set of parameters. 36 overloaded methods are deprecated, which means that potentially due to evolution they had to be replaced by other operations with different parameters.

We therefore examined the overloaded methods in more detail, looking at each group of overloaded methods. A group consists of all methods of the same name in the same class. Table II shows the results of our investigation.

There were a total of 1488 groups of overloaded methods. 403 of them, i.e., 27%, contained methods that were introduced over time in later versions of Java. Again, some of those methods could represent situations in which new features were introduced that required different parameters. 21 of those groups had a deprecated method in them, and for 10 of those new methods were introduced at the same time. These could represent situations in which new features were introduced, and as a result, methods required a new set of parameters to be able to continue to provide their functionality in the presence of the new feature.

Table II
GATHERED DATA OF ALL METHOD GROUPS OF THE *java.base* MODULE

Module	**java.base**
Number of method groups	1488
Minimum group size	2
Maximum group size	20
Average group size	2.74
Number of groups with a deprecated method	21
Number of groups where a method was introduced in a later version	403
Number of groups where a deprecated method exists and a method was introduced (potentially replacing the deprecated method)	10

We further manually searched the Java base module to find situations other than collections where signature extension could be helpful.

a) Adding Optional Functionality: With Java 1.4, support for different character encodings other than the platform's default encoding was added as a new feature through the additional class `java.nio.charset.Charset`. As a result, several methods across the API were added to take a `Charset` as an argument, or alternatively a *String* argument designating the name of the character set, e.g., *UTF-8*. For example, in addition to the method `java.lang.String.getBytes()`, `java.lang.String.getBytes(Charset)` as well as `java.lang.String.getBytes(String)` were added. In total, due to the introduction of the character encoding feature in Java 1.4, there are now 25 overloaded methods with an additional *Charset* argument.

Similarly, Java 1.4 introduced a new abstract class `java.net.SocketAddress` with one subclass `java.net.InetSocketAddress` which implements an IP socket address consisting of an IP address and port. This was added alongside the existing `java.net.InetAddress` (which only consists of the IP address). When dealing with sockets, methods requiring an `InetAddress` therefore also need an argument for the port. As a result, through the addition of `SocketAddress`, those methods (such as in the classes `DatagramPacket`, `DatagramSocket`, and `MulticastSocket` of the `java.net` package) had to be overloaded with the alternative functionality. Interestingly, in `java.net.Socket` and `java.net.ServerSocket`, instead of overloading the existing constructors with a `SocketAddress` argument, each class has a constructor to create an unbound socket and an additional method accepting a `SocketAddress`, which needs to be called after instantiation.

We located another additional optional functionality in classes that write to files. Both `java.io.FileWriter` and `java.io.FileOutputStream` allow optionally to

[2]Since the since attribute for `@Deprecated` was only introduced with Java 9, the earliest value retrieved is version 9.

[3]If a method does not have this information, we assumed that it was introduced in the same version as the class, which is provided with the `@since` tag in the Javadoc information of the class.

append to an existing file instead of writing it from scratch. This is specified using an additional `boolean` argument in the respective overloaded constructor. What is interesting to note is that `FileWriter` does not have an overloaded constructor allowing the character set to be specified (see above). Due to this lack, the Apache Commons IO library[4], which supplies input/output utilities, provides a class `org.apache.commons-.io.output.FileWriterWithEncoding` to accomplish this. I.e., it contains both optional functionalities to specify a custom character set and appending to a file.

b) Providing a Common Interface for Alternative Implementations: In Java, a design decision was made to use two disjoint class hierarchies for maps (`Map`) and collections (`Collection`). One of the reasons being that forcing maps to be collections or vice versa "[...] leads to an unnatural interface"[5]. As described in Section III-A, this is problematic when they should be treated in a uniform way. Furthermore, simply iterating over a map is not directly possible. Instead, the user needs to make the explicit choice to iterate over the *key*, *value* or *entry set*. The entries itself are key-value pairs that are exposed to the user. This can lead to inefficiencies when a user is not aware of this particularity, and writes code that iterates over the keys, only to then retrieve the corresponding value for each iteration step using the `get(key)` method.

In contrast, C# only uses one hierarchy, i.e., `Collection`, that also contains maps (`Dictionary`). However, because a collection is generic and typed to contain elements `E`, for maps the type `E` is a `KeyValuePair`, where each instance provides an entry of the map with a key and its value. Because the `add` operation is defined in the top-level class `Collection`, the user can call it also on a map, but needs to provide an instance of a `KeyValuePair` as a parameter. In order to help the user who does not need the additional layer of abstraction and hence does not mind to write dictionary-specific code, an additional, more convenient method (`add(K, V)`) is defined in `Dictionary` that transparently takes care of creating a `KeyValuePair` instance for the user.

B. Exploring Workarounds in Programming Languages

Looking at the Java Platform API confirmed the occurrence of the first two difficulties outlined in Section III. In order to overcome these at the programming language level, there exist certain workarounds.

a) Adding Optional Functionality: In the situation where a reusable unit is already being used and additional functionality is added, the goal is often to keep binary compatibility [8]. Guidelines for defensive interface evolution provide several ways to achieve minimal impact on the user in addition to the basic strategy outlined above [5]. Among them are *defender methods* (also known as *virtual extension methods*) introduced in Java 8 [6], abstract classes with default implementations, or Eclipse's way of specifying additional interfaces that extend the existing interface [9]. Other workarounds that are suggested for API evolution within Eclipse are marking the old method as *deprecated* and forwarding the call to the new method in its implementation.

b) Providing a Callback Interface that fits all Reuse Contexts: The third problem is a problem developers commonly face. The query "pass extra argument to callback function" on *Stack Overflow* results in 259 questions (or their answers) being matched. Various programming languages provide workarounds to overcome the difficulty of the user to require extra information within a callback. Here is a non-exhaustive list of techniques that we have observed at the code level to deal with the situation, some depending on specific implementation language features.

1) The user can define a custom interface that contains a method with the additional parameters. The user then implements the original interface, which when invoked determines the value for the additional parameters, and forwards the call to the custom interface with the additional arguments[6].
2) When a programming language supports anonymous inner classes, the user can declare an operation with parameters that hold the additional information. When called, the method creates an anonymous instance of the callback interface and returns it, which can then be registered with the framework. When the callback is received, the additional parameters of the operation are accessible from the anonymous inner class[7].
3) In Python, lambdas, partial functions [27], or function decorators [24] may be used to augment framework-defined callbacks with additional parameters.
4) In Javascript, it is possible to call a function with more arguments than defined in its signature. Inside the function, arguments can be accessed using the `arguments` object [26]. This could help the user to access additional arguments in a callback, but the designer would also have to pass additional values in the call. Alternatively, it is possible to bind additional parameters to a function using the `bind` function [15], [10].
5) Similarly, in C++, a *bind* method exists (provided by a separate library called *boost::bind* from the C++ collection of libraries called *boost*) [4], [20]. It allows to create a new function pointer to be created with the original function's arguments bound or rearranged, and also to add additional parameters, if needed.

V. Discussion and Outlook

The *Collection* and *Resource Management* examples in Section III, as well as the empirical observations at the programming language level presented in Section IV provide

[4]See https://commons.apache.org/proper/commons-io/.

[5]http://docs.oracle.com/javase/8/docs/technotes/guides/collections/designfaq.html#a14

[6]This technique is illustrated in the Android example *XYZTouristAttractions* in the `AttractionListFragment` Java class using the custom `ItemClickListener` interface [22], for example.

[7]This technique is illustrated in the Android example *XYZTouristAttractions* in the `AttractionsGridPagerAdapter` Java class [23], for example.

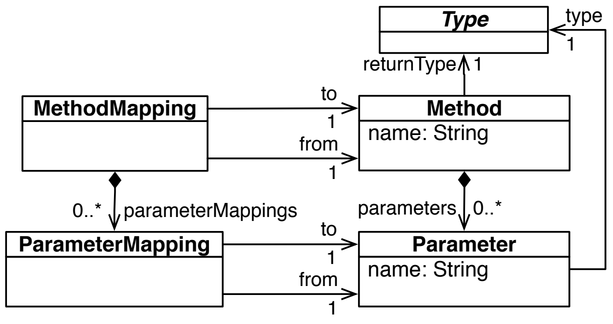

Figure 4. Signature Extension Metamodel Excerpt

evidence that the finality of signature declarations can cause difficulties to modellers and developers of reusable artefacts. The problem is more pronounced in the context of SPL development, where the number and type of parameters of methods might vary depending on the selected features for a product. To the best of our knowledge none of the existing modelling or programming language techniques or workarounds can deal with these difficulties in a satisfying way.

We therefore propose a mechanism for *signature extension* as a solution to overcome the aforementioned difficulties. Figure 4 shows an excerpt of a metamodel that provides mappings between methods and between parameters. Using these mappings, a method signature with its parameters can be defined in one place, and then re-defined again in another place, potentially with additional parameters. *MethodMappings* are established between the two definitions of the method, and *ParameterMappings* are created for the common parameters. This makes it possible to incrementally build a signature, e.g., when moving down abstraction levels, or when adding support for additional features. For example, for the *Collection* example outlined in Section III-D, signature extension allows to define a common interface in *Kind* for adding an element. *Ordered* then extends the signature with an additional index argument, whereas *KeyIndexed* extends it with an additional key argument. In addition, it allows the common interface of *Kind* to be used before deciding on any concrete collection.

The signature extension approach should be applicable to any software development language that uses signatures to define interfaces. The approach allows to incrementally define signatures, as well as use partially defined signatures before all extensions are known. We are currently in the process of adding support for signature extension to Concern-Oriented Reuse (CORE) [1], [21], a compositional, model-driven approach to software development inspired by advanced separation of concerns, software product lines and aspect-oriented software development. CORE supports design modelling using class, sequence and state diagrams. We are using the signature extension mechanism outlined above to make it possible for the designer or the user of reusable class diagram models to specify structural extensions to signatures already defined in other class diagrams. We are planning to use aspect-oriented modelling techniques to specify how to deal with the additional parameters behaviourally in the sequence diagrams and state diagrams, e.g., by specifying additional behaviour for existing sequence diagrams of signatures that were extended to deal with the additional formal parameters, or by adding additional values to method calls of extended signatures.

References

[1] O. Alam, J. Kienzle, and G. Mussbacher, "Concern-Oriented Software Design," in *MODELS 2013*. Springer, 2013, pp. 604–621.

[2] C. Bensoussan, M. Schöttle, and J. Kienzle, "Associations in MDE: A Concern-Oriented, Reusable Solution," in *Modelling Foundations and Applications - 12th European Conference, ECMFA 2016, Held as Part of STAF 2016, Vienna, Austria, July 6-7, 2016, Proceedings*. Springer International Publishing, 2016, pp. 121–137.

[3] J. Bloch, "How to Design a Good API and Why It Matters," in *OOPSLA '06*. New York, NY, USA: ACM, 2006, pp. 506–507.

[4] P. Dimov, "Boost.Bind Documentation: Chapter 1. Boost.Bind," https://www.boost.org/doc/libs/1_69_0/libs/bind/doc/html/bind.html, 2018.

[5] L. Eder, "Defensive API Evolution With Java Interfaces," https://dzone.com/articles/defensive-api-evolution-java, 2013.

[6] B. Goetz, "Interface evolution via virtual extension methods," https://cr.openjdk.java.net/~briangoetz/lambda/Defender%20Methods%20v4.pdf, June 2011.

[7] E. Henry and B. Faller, "Large-scale Industrial Reuse to Reduce Cost and CycleTime," *IEEE Software*, vol. 12, no. 5, pp. 47–53, Sep. 1995.

[8] IBM Corporation, "Evolving Java-based APIs 2," https://wiki.eclipse.org/Evolving_Java-based_APIs_2, 2017.

[9] ——, "Evolving Java-based APIs 3," https://wiki.eclipse.org/Evolving_Java-based_APIs_3, 2017.

[10] JotaBe, "Pass extra parameters to jquery ajax promise callback," https://stackoverflow.com/a/21985202, Feb. 2014.

[11] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson, "Feature-oriented domain analysis (FODA) feasibility study," SEI, CMU, Tech. Rep. CMU/SEI-90-TR-21, November 1990.

[12] J. Kienzle, G. Mussbacher, O. Alam, M. Schöttle, N. Belloir, P. Collet, B. Combemale, J. DeAntoni, J. Klein, and B. Rumpe, "VCU: The Three Dimensions of Reuse," in *International Conference on Software Reuse, ICSR 2016*, ser. LNCS. Springer, 2016, no. 9679, pp. 122–137.

[13] J. Kienzle, G. Mussbacher, P. Collet, and O. Alam, "Delaying Decisions in Variable Concern Hierarchies," in *GPCE*. ACM, 2016, pp. 93–103.

[14] Krueger, "Software reuse," *CSURV: Computing Surveys*, vol. 24, 1992.

[15] Mozilla MDN web docs, "Function.prototype.bind()," https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_objects/Function/bind, 2018.

[16] Oracle, "Varargs," https://docs.oracle.com/javase/1.5.0/docs/guide/language/varargs.html, 2010.

[17] D. L. Parnas, "On the Criteria To Be Used in Decomposing Systems into Modules," *Communications of the ACM*, vol. 15, no. 12, pp. 1053–1058, Dec. 1972.

[18] K. Pohl, G. Böckle, and F. J. van der Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.

[19] Python Software Foundation, "The Python Tutorial: Arbitrary Argument Lists," https://docs.python.org/3/tutorial/controlflow.html#arbitrary-argument-lists, 2019.

[20] Radman Games, "How to use Boost.Bind," http://www.radmangames.com/programming/how-to-use-boost-bind, 2011.

[21] M. Schöttle, O. Alam, J. Kienzle, and G. Mussbacher, "On the Modularization Provided by Concern-Oriented Reuse," in *MODULARITY Companion 2016*. ACM, 2016, pp. 184–189.

[22] The Android Open Source Project, Inc., "Android XYZ-TouristAttractions Sample," https://github.com/googlesamples/android-XYZTouristAttractions/, Apr. 2018.

[23] ——, "Android XYZTouristAttractions Sample," https://github.com/googlesamples/android-XYZTouristAttractions/, Apr. 2018.

[24] The Code Ship, "A guide to Python's function decorators," https://www.thecodeship.com/patterns/guide-to-python-function-decorators/, 2014.

[25] The Go Authors, "The Go Programming Language Specification: Function Types," https://golang.org/ref/spec#Function_types, 2018.

[26] w3schools.com, "JavaScript Function Parameters," https://www.w3schools.com/js/js_function_parameters.asp, 2019.

[27] wxPyWiki, "Passing Arguments to Callbacks," https://wiki.wxpython.org/Passing%20Arguments%20to%20Callbacks, 2011.