Safer Typing of Complex API Usage through Java Generics

William Harrison
Software Structure Group
Dept. of Computer Science
Trinity College Dublin
bill.harrison@cs.tcd.ie

David Lievens
Software Structure Group
Dept. of Computer Science
Trinity College Dublin
david.lievens@cs.tcd.ie

Fabio Simeoni
Dept. of Computer and
Information Sciences
University of Strathclyde
fabio.simeoni@cis.strath.ac.uk

ABSTRACT

When several incompatible implementations of a single API are in use in a Java program, the danger exists that instances from different implementations may inadvertently be mixed, leading to errors. In this paper we show how to use generics to prevent such mixing. The core idea of the approach is to add a type parameter to the interfaces of the API, and tie the classes that make up an implementation to a unique choice of type parameter. In this way methods of the API can only be invoked with arguments that belong to the same implementation. We show that the presence of a type parameter in the interfaces does not violate the principle of interface-based programming: clients can still completely abstract over the choice of implementation. In addition, we demonstrate how code can be reused between different implementations, how implementations can be defined as extensions of other implementations, and how different implementations may be mixed in a controlled and safe manner. To explore the feasibility of the approach, gauge its usability, and identify any issues that may crop up in practical usage, we have refactored a fairly large existing API-based application suite, and we report on the experience gained in the process.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications—*Object-oriented Languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Polymorphism, Abstract Data Types*

General Terms

Programming languages, Theory, Verification

Keywords

Interface-based programming, Generics, Family Polymorphism, Programming patterns

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PPPJ '09, August 27–28, 2009, Calgary, Alberta, Canada.
Copyright 2009 ACM 978-1-60558-598-7 ...$10.00.

1. INTRODUCTION

Large systems are typically built from components that interact through application programming interfaces (APIs). The virtues of *interface-based programming* are well recognized. To name just a few: reduced complexity of individual components, looser coupling between components, improved testability, the potential to exploit multiple implementations in different domains, etc. In Java, an API is expressed as a set of interfaces, and implementations of the API are sets of classes implementing the interfaces. We call an API *complex* when there are interdependencies between the different interfaces. In other words, when interfaces of the API define methods with parameter or return types that are one of the other interfaces of the API.

Often APIs just offer a window on the underlying system, and classes that make up an implementation are not expected to solely rely on the interface for internal interaction. This mean that implementation classes typically have dependencies upon each other above and beyond those visible in the interfaces of the API. These *encapsulated dependencies* may be on functionality that is not exposed by the API, on hidden shared state, or even on semantic properties of the implementation of certain methods that can not be captured through an interface. As a consequence, different implementations of an API may not be compatible with each other, and objects from different implementations can —and should— not be mixed, or errors may ensue. The potential for such mixing is what we call the *complex API problem.*

It is important to point out that errors due to mixing are not normally caught at compile-time by the type system. Consider the example introduced in Fig. 1. The interfaces `IButton` and `IIcon` may have a number of different implementations. For example, one suitable for the Windows platform and another targeted at the Mac OS platform. A Windows-specific button may not behave correctly when used with a Mac OS-specific icon. However, all implementations of the icon interface are subtypes of `IIcon`, so it is always possible to invoke the `setIcon` method with any of them, irrespective of which implementation of buttons is in use.

```
IButton b = new WinButton();
IIcon i = new MacIcon();
b.setIcon(i);                    //run-time error
```

So, although interface-based programming enables client code to be oblivious to the choice of implementation, applications may depend on consistent usage of one such implementation, and this invariant needs to be maintained in

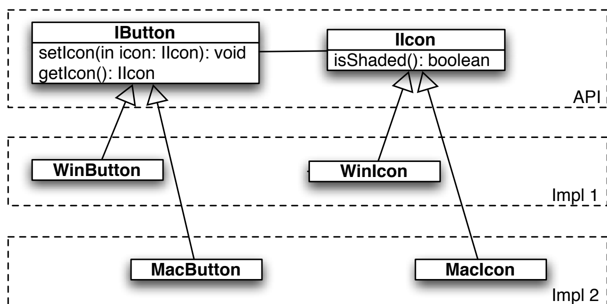

Figure 1: Example API and its Implementations

the face of evolution whereby new implementations are injected into an application over time. Furthermore, some applications may wish to use different implementations of a single API at the same time, yet need to keep incompatible data flows apart. An example of such an application is the *Concern Manipulation Environment*(CME) [8]. This environment is an open-ended suite of tools that makes heavy use of plug-ins implementing a number of complex APIs. One of those, the *Concern Information Tool* API (CIT), is similar to the Java Reflection API —except that Java Reflection is characterized by classes and abstract classes (rather than interfaces), and therefore unsuitable for use in defining a plug-in component. The CIT API has implementations to handle class files, Java source files, and UML diagrams; each with internal encapsulated dependencies. Other CME components that construct concern models use these CIT implementations to extract information from different kinds of artefacts. Some of these components use different kinds of artefacts at the same time, and hence run the danger of accidentally passing an instance of one implementation to a method of an object that belongs to a different implementation.

In this paper, we propose *a practical solution based on standard Java generics, that prevents mixing errors at compile-time*. The core idea is to add a type parameter to the interfaces of the API, and tie the classes making up an implementation to a type parameter that uniquely identifies the implementation. In this way, methods of an API can only be invoked with arguments that belong to the *same* implementation of the API, and attempts at mixing of objects from different implementations are flagged statically. In terms of the example of Fig. 1, this means that —subject to properly structured class implementations— we would be told at compile-time that we used Mac icons with Windows buttons, even when we type their denotations at the level of the interfaces `IButton` and `IIcon`.

The technique proposed can be readily applied, and the bulk of the discussion in this paper is concerned with investigating how the technique can be aligned with the practice of Java programming (Sec. 2–Sec. 4). Among others, we demonstrate how to write implementation-agnostic clients (Sec. 3.1), how to achieve controlled mixing of implementations (Sec. 3.2), how to reuse code between implementations (Sec. 4.1), and how to maintain substitutability for implementations defined in terms of existing implementations — which we call sub-implementations (Sec. 4.2). We also detail a case study that we have undertaken to explore the feasibility of the technique we propose (Sec. 5). We end the paper with a discussion of the relative merits of our approach (Sec. 6) and related work (Sec. 7), before summarizing our conclusions (Sec. 8).

2. ENFORCING CONSISTENT API USAGE

2.1 General Approach

As indicated above, our approach to prevent mixing objects from different implementations revolves around adding a type parameter to the interfaces of an API. When the API is complex, this type parameter shows up at all the interdependent parameter and return types of methods in the interfaces. The type system can then enforce that arguments to a method have the same type instantiation as the receiving object. When type instantiations uniquely identify implementations, objects from different implementations can not be mixed. Consider again the example in Fig. 1 of buttons and their icons. The parameterized interfaces would take the following form:

```
interface IButton<IMPL> {
  void setIcon(IIcon<IMPL> icon);
  IIcon<IMPL> getIcon();
}
interface IIcon<IMPL> {
  boolean isShaded();
}
```

Note in particular how the types of the `icon` parameter of the `setIcon` and return value of the `getIcon` method are bound by the type parameter of the `IButton` interface.

Implementations of an API need to instantiate the type parameter to a particular type. The main requirement is that all the classes that make up an implementation do so with respect to the *same* type. What type is being used for this purpose is of secondary importance. It can be any type, as long as it is used consistently. We call the instantiation type for an implementation its *implementation token*, as it shows up as the type parameter in client code. It is important to note that the association between implementation classes and implementation tokens is managed by the provider of an implementation, and that this happens without feedback of the type system. We will come back to this issue later in the paper. One possible choice of implementation token is an empty interface defined especially for the purpose. A simple implementation of the above API may then look, in skeleton, as follows:

```
interface Win {};     //implementation token
class WinButton implements IButton<Win> {
  IIcon<Win> getIcon() {...}
  void setIcon(IIcon<Win> icon) {...}
}
class WinIcon implements IIcon<Win> {
  boolean isShaded() {...}
}
```

Clients programming against the parameterized API are now prevented from invoking methods with objects from incompatible implementations: attempts to invoke the `setIcon` method on a Windows button with a Macintosh icon as argument yield a compile-time error:

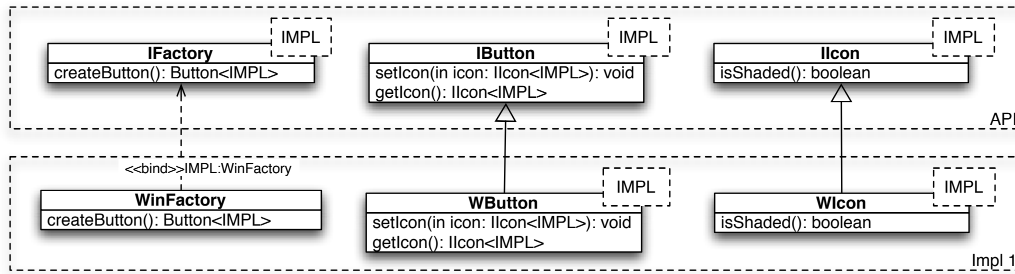

Figure 2: Implementation using abstract factory pattern

```
IButton<Win> b = new WinButton();
IIcon<Mac> i = new MacIcon();
b.setIcon(i);          //compile-time error
```

Note how we use type parameters in an atypical way to distinguish incompatible implementations of the same interface. Normally, type parameters are used to abstract away differences among types, to code uniformly against the widest possible set of values without giving up type safety. For example, collections are parameterized in the type of their elements so that one can write code that works against collections of different types of values. Here, in effect, we use type parameters to mark types, so we can keep their values apart.

2.2 Using Factories

Applications often use creational patterns [7] to abstract over more complex object instantiation scenarios. In particular, the *Abstract Factory* pattern can be used to establish an instantiation protocol that concrete factories may adopt to return objects of different implementations. By applying the technique to such factories, one can statically guarantee that they return a consistent set of objects (e.g. across different methods of a factory). Additionally, client code is prevented from mixing objects from different (incompatible) factories. With respect to the running example, the abstract factory would be expressed as an additional interface in the API:

```
interface IFactory<IMPL> {
  IButton<IMPL> createButton();
  IIcon<IMPL>   createIcon();
}
```

Concrete factories implement the above interface by instantiating the type parameter and returning instances of classes from corresponding implementations. Note the threading of the type parameter through the return types of creation methods, which makes returning an instance of an incompatible implementation a type error.

Fig. 2 shows how an implementation including factories could be structured. Note in particular, that in this case we have chosen to do the type instantiation at the point of object instantiation in the factory —rather than in the class definition; and that we have used the the factory class itself as the implementation token. Using the concrete factory as implementation token saves us introducing an empty interface. Due to its pragmatic nature, it is this pattern of implementation that we have used in the case study that we discuss in Sec. 5. Client code creates or obtains a factory once, and relies on it to generate objects that can be used together. In this way, the client code can avoid naming the implementation classes it relies upon:

```
IFactory<WinFactory> f = new WinFactory();
IButton<WinFactory> wb = f.createButton();
IIcon<Winfactory> wi = f.createIcon();
wb.setIcon(wi);       // guaranteed to be ok
```

2.3 Implementation-side Guarantees

The absence of mixing between different implementations may be exploited in implementation classes. In particular, it is clear that under the scheme, methods may make stronger assumptions on their inputs than the ones advertised in parameter types. For example, the inputs to the `setIcon` method of the `WinButton` class defined in Sec. 2.1 are guaranteed to be instances of the `WinIcon` class. The method may depend on this fact, either implicitly, for behavioural properties, or explicitly, to access features of the class that are hidden behind the interface, without fearing that its assumptions may be violated. For example, casts in this context are guaranteed to succeed, and hence need not be guarded with exception handling:

```
class WinButton implements IButton<Win> {
  void setIcon(IIcon<Win> i) {
    ...(WinIcon)i... // cast guaranteed to succeed
  } ...  }
```

3. GENERIC CLIENTS

In client code as shown in Sec. 2, implementation tokens appear manifest in the program text. For example, by looking at the types of the variables `b` and `i`, we see that `b` refers to an instance of the Windows-based implementation of button, while `i` refers to a Mac icon. This enables the type checker to spot when we attempt to mix implementations inappropriately. However, it may give the impression that our technique inherently violates the principle of interface-based programming that it sets out to support. This is not the case: we can effectively abstract over implementation tokens to write code that does not depend on the exact choice of implementation. The main tools to achieve this are *generic methods*, which use type parameters to make implementation tokens abstract, and *wildcards*, which offer a form of subtyping for parameterized types.

3.1 Implementation-Agnostic Clients

Generic methods may be made applicable to any implementation by taking the implementation token as a type parameter. For example, the method `m`, shown below, can be invoked with any implementation of the `IButton` interface, irrespective of its implementation token, and returns an icon of the same implementation. The *main* method shows a particular invocation using a Windows button.

```
class GenericClient {
  <IMPL> IIcon<IMPL> m(IButton<IMPL> b) {
    // code uses abstract implementation token
  }
  void main() {
    IButton<Win> wb = new WinButton();
    IIcon<Win> wi = m(wb);
  }
}
```

In the body of `m`, the implementation token is abstract, only known as `IMPL`. This makes the code in the body of `m` implementation-agnostic, and hence reusable against any implementation. However, the implementation token needs to be known to caller of the method. This may, of course, be a generic method itself, deferring the issue. But ultimately the call chain needs to be grounded in a context where the concrete implementation token is known. When we are aiming to maximize the amount of implementation-agnostic manipulation, this often is the point of creation of a value. Here, wildcards come to the rescue.

Wildcards offer a form of subtyping for parameterized types [16]. In general, two different instantiations of a generic class C are not related through subtyping. However, the wildcarded type $C\langle ? \rangle$ is a supertype of any type instantiation of C. This means that variables typed with a wildcard can be bound to values of any implementation. In the following code, the variable `wb`, could hold an object of either Windows- or Mac-based implementations. Typing a button with a wildcard does not necessarily prevent it from being passed into a generic method for further implementation-agnostic processing. This is possible because of a process known as *wildcard capture* (cf. [16], Sec. 3.2). It means that we may still invoke `m` with `wb`, despite not knowing the implementation of the value bound to `wb`.

```
IButton<?> wb = new WinButton();
IIcon<?> wi = m(wb);  // wildcard capture
```

The problem with wildcards is that by hiding the implementation token, interaction between values of the same implementation may be hindered. Wildcard capture may mitigate this problem somewhat, but is subject to certain limitations. For example, a method that takes two parameters of the same implementation can not be invoked by relying on wildcard capture, as different wildcards, for reasons of soundness, are always assumed to be of incompatible types. Similarly, relations between input types and return types are lost. For example, on invoking `m` with `wb`, it is 'forgotten' that the result is of the same implementation as the input. This means that, with respect to the above assignments, the following, although sound, would not be allowed:

```
wb.setIcon(wi); // not allowed
```

A way of avoiding these limitations, is to bundle value creation into factories, typed by wildcards on their creation, and then, using wildcard capture, inject these factories into classes that are parameterized in their implementation tokens. Such a pattern can be used to achieve clients that are entirely agnostic to the choice of implementation, i.e. that contain no manifest occurrences of specific implementation tokens. The pattern, which we demonstrate by example below, is somewhat evocative of the kind of approach advocated in so-called *dependency injection frameworks* [9]. In essence, such frameworks separate application code from choice of implementation by injecting a reflectively instantiated factory (or any other configurable 'dependency') into an application.

The example code contains two classes: `GC`, a generic client that is entirely agnostic of the choice of implementation, and `DI`, which contains static methods to reflectively instantiate a concrete factory based on some input string `fname`, and inject it into a new instance of `GC`. The generic client class is parameterized in a single implementation token, and receives a factory bound to that implementation token on instantiation. The client creation process initiated in `DI` relies on the method `createGC`, which performs wildcard capture on the implementation token of the reflectively instantiated factory, before instantiating `GC` with the captured implementation token.

```
public class GC<IMPL> { ...
  GC(IFactory<IMPL> f) { this.f = f; }
   ...
  <IMPL> IIcon<IMPL> m(IButton<IMPL> b) { ... }
  public void main() { ... }
}
public class DI {
  static <T> GC<T> createGC(IFactory<T> f) {
    return new GC<T>(f);
  }
  static IFactory<?> createF(String fname) {
    return (IFactory<?>)
             Class.forName(fname).newInstance();
  }
  static GC<?> init(String fname) {
    return createGC(createF(fname));
  }
  static void run() {
    GC<?> c = init(...);
    c.main();  // start client
  }
}
```

3.2 Dynamic Clients

Up to this point, we have only been concerned with totally preventing any mixing of implementations. However, there are situations where one can be more permissive: some methods may not depend on encapsulated dependencies, and hence may safely mix inputs from different implementations. In these cases, we may use wildcards to type method parameters, and such methods may then be invoked with any combination of different implementations of the corresponding interfaces.

Considering that a similar effect can be obtained using generic methods with multiple type parameters, a more pertinent use of wildcards may be to type collections containing objects from different implementations:

```
List<IButton<?>> bs = ...
bs.add(new WinButton());
bs.add(new MacButton());
```

The process at work with wildcards is normal subtyping. Once the type parameter is abstracted over, some operations may not be applicable, and the value may not be passed into a context that requires a specific implementation. With subtyping between non-generic classes, if specific knowledge is needed after it has been abstracted a way, a cast may be used to dynamically reassert additional type information. However, Java implements generics using *erasure*, which does not give a run-time representation to type parameters. This means that casting from a wildcard to a specific implementation token, although allowed, makes little sense, as it will never fail.

```
IButton<?> mb = new MacButton();
IButton<Win> wb = (IButton<Win>)mb; // never fails
```

Of course, using reflection, objects can be interrogated for the class of which they are an instance. However, relying on this information instead of implementation tokens implies that clients would need intimate knowledge of the specific classes that make up a particular implementation of an API. It would be better if it were possible to offer a way to dynamically query what implementation an object belongs to in terms of implementation tokens. One reasonable way to achieve this would be to add a method to all APIs that supports such querying. Assuming that such a method is called `getToken`, one could use the following code to identify all the buttons in a heterogeneous list of buttons, which have a shaded icon, and which belong to the same implementation as the first button in the list:

```
for ( IButton<?> b : bs )
  if (b.getToken() == bs.get(0).getToken()
          && b.getIcon().isShaded())
    ...
```

Alternatively, one may use such a method as a semantically sound guard before a cast:

```
if ( mb.getToken() instanceof Win )
  wb = (IButton<Win>)mb;
```

3.3 Raw Clients

A final way to program uniformly against any implementation of an API is to ignore the parameterization altogether, i.e. use the API *raw* [10]. This removes the burden imposed on a client by a parameterized API, but, of course, also its benefit. Manipulation of the API is unconstrained again, like in the pre-parameterized version, and equally unsafe. The possibility of having raw clients makes that the technique becomes *pay-as-you-go* for client programs. Simple clients of an API, for which guaranteeing the absence of mixing is trivial, may ignore the parameterization, while more complex clients of the same API may instead rely on it to get the feedback of the type system on mixing invariants.

4. IMPLEMENTATION-SIDE CODE REUSE

4.1 Code Reuse across Implementations

Code from classes such as `WinButton` that are part of an implementation can not be reused in classes from a different implementation because their type parameter is already instantiated. Any subclass would inherit the type parameter too, making it, by definition, part of the original implementation, and hence mixing of instances would be allowed[1]. To achieve code reuse *across* implementations, one needs to separate the definition of implementation code from the association of that code to a particular implementation —its type instantiation. We have already shown an example of this in Fig. 2, where the type instantiation was done in the factory, and the implementation classes were left uninstantiated. As a consequence, these classes can be inherited from, without subclasses becoming part of the implementation of their superclasses.

To maximise code reuse, we suggest a pattern whereby code that is intended to be reused across implementations is factored out in uninstantiated classes. One can then achieve code reuse by inheriting from these classes. Type instantiation can either happen *on-the-fly*, e.g. in factories, or, when non-reusable implementation-specific code is desired, on derivation into a class that is statically associated with a particular implementation. Hierarchies may even contain nodes that are only there for code reuse, and are not part of a particular implementation. For example, one could easily imagine that the Windows-based and Mac-based implementations of buttons may share an (abstract) common superclass. Fig. 3 shows a hierarchy of implementation classes of the `IButton` interface that observes the pattern we describe. The classes `AButton`, `WButton` and `MButton` define reusable code, and presumably contain the bulk of the implementation code. The leaf node `WinButton` ties down its implementation token, and hence can only contain code specific to the implementation it has committed to.

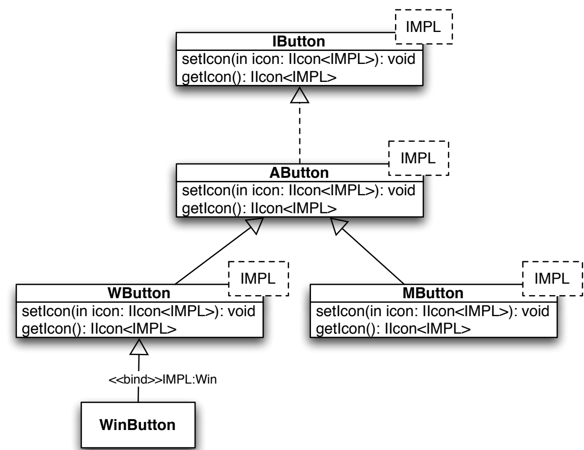

Figure 3: Class organization that facilitates code reuse across implementations

[1]Note that this in itself need not be a problem: one can imagine implementations of an API that have different classes of the same interface which are designed to be mixed.

4.2 Sub-implementations

Often implementations may be written as extensions of other implementations. In these cases, the extended implementation may still satisfy the encapsulated dependencies assumed by the base implementation. We would like to be able to pass objects from the extended implementation into code that expects objects from the base implementation. In other words, we want to introduce the ability to mix objects from related implementations—albeit one-way, from extended implementations to base implementations, and not the other way around. Note that, even though the sub-implementation may well be derived from the base implementation by inheritance, just passing objects from sub-implementations into methods that expect objects from base implementations does not work: different type instantiations of otherwise related classes are never subtypes of each other. In other words, $D\langle Y\rangle$ is never a subtype of $C\langle X\rangle$, even when D is a subtype of C and Y is a a subtype of X.

However, we can achieve the kind of substitutability that we are after by changing the signature of methods in the API to include bounds. For example, the `setIcon` method of the `IButton` interface may be made into a generic method whose parameter is bounded by the implementation token of the interface. In this way, the method can be invoked not only with objects from the same implementation, but also with objects from sub-implementations.

```
interface IButton<IMPL> {
  <I extends IMPL> void setIcon(IIcon<I> icon);
}
```

Of course, for this to work, implementation tokens of related implementations need to form a hierarchy. This makes it impossible to use of concrete factory classes as implementation tokens, because concrete factories tie down their implementation token, and hence are not subtypes of each other. Also, when using sub-implementations, classes needs to be careful when being specific in their dependency on other classes of their implementation. Consider the classes in the hierarchy of Fig. 4. We said before that methods of the class `WinButton` may depend on the fact that they will be invoked with objects coming from the same implementation. However, if the `setIcon` method in `WinButton` were to cast to `WinIcon` (assumed to be part of the same implementation, but not shown in Fig. 4), this cast may fail when the method is invoked with an instance of the class `ExtWinIcon` which is part of a sub-implementation. These classes, although possibly sharing most of their code, are not related to each other through subtyping or (direct) inheritance. To avoid this problem, dependencies should only exist between uninstantiated classes, where sub-implementations align with inheritance hierarchies. The problem is in part mitigated by the fact that the API advertises when sub-implementations may be provided and when not.

5. CASE STUDY

To establish the workability of the approach, and investigate issues that may arise in its application, we refactored a number of APIs and their implementations that form part of the *Concern Manipulation Environment*(CME) [8]. In particular, we have applied our approach to two APIs called CIT and CAT —the *Concern Information Tool* and *Concern Assembly Tool*. The CAT API is supported by a framework,

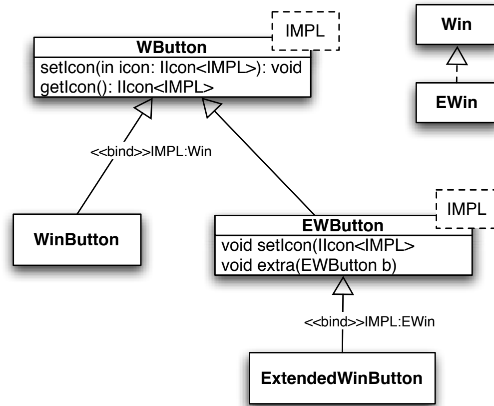

Figure 4: A Hierarchy with Sub-implementations

which was used in the implementation of several simple and several substantial CATs. The CIT API has some framework and utility support, and several substantial implementations. The CME components and API definitions that we selected for refactoring consist of 250 classes and interfaces, comprising about 44Kloc. An overview of them can be found in Table 1.

The refactoring process was performed on an API by API basis. First adding the type parameter to the interfaces, and then following the compiler diagnostics to identify the places in the implementations and clients where changes were required. We used many of the techniques outlined above, but avoided the ones that would have required to restructure code. Most client code has been made independent of concrete implementation tokens. This was helped by the fact that creational dependencies of the client on specific implementations had already been wrapped into abstract factories in the original application. Because of the presence of factories, we used them as implementation tokens, avoiding the introduction of new interfaces for this. There was typically only a need to deal with objects from one implementation at any one time, so mostly only a single type parameter was needed in generic methods or classes. However, one component that dealt with composition, potentially involving simultaneous creation of several output forms, was required to keep apart objects from up to five different CAT implementations. Another component, acting as a ‘coupler’ or ‘splitter’ for CAT, passes API calls to objects from several different implementations. The bulk of the code for the implementations of the CIT and CAT APIs is located in supporting frameworks. We did not use wildcards in the parameterization of the APIs, but some static helper methods in these supporting frameworks were found to be most appropriately typed using wildcards. We did not use sub-implementations or bounds on API methods in our application.

As an aside, we found the refactoring process itself relatively straightforward, and we believe it could be facilitated further through a mechanical refactoring tool that would only need guidance on how permissive methods ought to be in admitting different implementations (whether to use wildcards or bounds as type parameters).

Characterization	#classes/ interfaces
Concern Information Tool API	28
Concern Assembly Tool API	22
CIT framework and utilities	39
CAT framework	154
Four small CAT implementations: serializer, trace, mini, connectors	72
CIT/CAT implementation for Java class files	89

Table 1: CME Components that were Refactored

The case study also highlighted a number of general issues with the integration of generics into Java that may come up when parameterizing an API. For example, some of the APIs that we refactored declared exceptions as part of their protocol. However, exceptions in Java can not be parameterized. The reason for this is that exceptions need to derive from the abstract class `Throwable` which itself is not parameterized. This means that exceptions-based control flow may receive less help from the type checker regarding mixing than normal control flow. Another issue that cropped up was the fact that Java disallows the use of type parameterization for static variables. This prevents constants (*final static* variables) from holding objects related to a parameterized API (unless used raw). Finally, the use of arrays to hold objects related to an API may also sometimes be complicated due to the parameterization. A discussion of the reasons for this is out of scope; we refer the interested reader to [1], Sec.7.3. Of course, the aforementioned issues all relate to the general use of generics in Java, and are not specific to our proposed technique.

In conclusion: although the technique that we propose is invasive because of the proliferation of type parameters throughout application code, the case study seems to demonstrate that the technique is *feasible in practice*, and can *scale up to real application scenarios.*

6. DISCUSSION

In this section, we turn to an informal cost-benefit analysis of our technique with respect to the standard non-parameterized solution. The main benefit, in fact, the purpose of the technique is to statically prevent programs from mixing incompatible API implementations. Of course, when this aspect of a program is trivially assured, the help of the type checker is of little benefit. On the other hand, when multiple data streams need to be kept apart, or the creation process is complex and/or subject to frequent evolution, static feedback may yield significant advantages.

We must note that for our technique to yield proper static feedback, implementation tokens must uniquely identify implementations. Classes are associated with implementation tokens through type instantiation, but such instantiation bears no relationship to, or is not constrained by the encapsulated dependencies. This means that we may get it wrong, and when we do get it wrong, the type system does not warn us. For example, we may mistakenly associate the `Win` implementation token to a class providing an implementation for Macintosh buttons. If we do so, our technique will, in fact, ensure that we will *always* mix it with with instances that it can not handle, rather than prevent any such mixing.

Of course, getting it wrong means an implementer got confused on what classes make up its implementation. And when this happens, there are likely to be more errors in the implementation, such as casts to incompatible classes, or functional dependencies that are not being met. The problem may be mitigated by adopting design practices that aim to make it as clear as possible what makes up an implementation. For example, one may simply separate different implementations into different packages (making inter-implementation references show up as import statements). Additionally, one may move type instantiation outside the context of ordinary computation by doing it solely in class definitions, and making all parameterized classes abstract (discouraging instantiation inside factories).

Ultimately, by using implementation tokens to abstractly denote implementations, our technique provides a compromise between safety —preventing mixing for consistently defined implementations— and usability —the use of a single type parameter independent of the size or nature of the implementation. We believe this compromise to be pragmatically useful.

On the cost side, we need to mention the proliferation of type parameters showing up in code that uses APIs. As detailed above, we have undertaken a fairly large case study to gauge the practical impact of this issue. From it, we conclude that the syntactic overhead does not impede its application. In fact, tagging code with implementation tokens may make it clearer where in a program different implementations may interact.

It is also worth mentioning that our technique does not impose a run-time overhead, as implementation tokens exist only at compile-time due to the way generics is implemented in Java (by erasure). Of course, interface-based programming itself comes at a cost for method invocations compared to invocations of static methods or class methods.

Finally, as we have explained at length, a number of techniques can be used to minimize the loss of flexibility induced by parameterizing interfaces. For example, methods that do not depend on encapsulated dependencies can type their parameters with wildcards. Similarly, the ability to use sub-implementations relies on methods using type bounds. However, such changes to method signatures must be made in the API, and can not be adopted by individual implementations. This means that the 'permissiveness' of method inputs needs to be decided up-front and adopted uniformly by all implementations.

7. RELATED WORK

How to use complex APIs without mixing their implementations is a pragmatically important take on a well-known problem in the study of object-oriented languages. As usual, the tension is between static safety —which requires as detailed type information as possible— and expressiveness —which requires as abstract type information as possible. Subtyping is one of the mechanisms normally

employed to mitigate this tension, as it can hide type information from the typechecker. Unfortunately, when applied naïvely to interface-based programming, it compromises type safety by hiding the co-occurrence constraints of implementations (i.e. thei encapsulated dependencies).

One does not need a *complex* API to witness the problem; a single type that refers to itself, i.e. has some binary method, is sufficient to bring it out [4, 2]. However, complex APIs require more general solutions, and the literature adopts the term *family polymorphism*[2] for the problem that deals with interdependent types. In this line of work, the term *family* corresponds to what we called an *API implementation*, and top-level families define APIs.

A plethora of different approaches have been proposed to tackle the problem of family polymorphism. All restrict the use of subtyping between individual members of a family. There are approaches based on virtual types [3, 5, 13, 6], path-dependent types [12, 15], and various forms of extended inheritance [11, 14], to mention just a few. To the best of our knowledge, none of these approaches is directly applicable to Java —or any other mainstream class-based object-oriented language— without language extensions. In contrast, we propose an approach that is directly deployable, and aligns well with current software engineering practice.

One way to constrain subtyping, and restore the safety of interface-based programming, *without* resorting to a language extension, is to use generics. The overall idea is to use type parameters to abstract over the types of an API implementation, rather than partially hiding them from the typechecker. In Java, generics can be reconciled with subtyping —using wildcard— to overcome some of the limitations that have been attributed to such a solution [5]. In our approach, we denote implementations abstractly through an implementation token, yielding a single type parameter for any size of family. A more straightforward application of generics in this context, would yield classes that are parameterized recursively on *all* the other classes of their family. However, this proliferation of type parameters would mark a significant increase in complexity for both clients and implementers, making it effectively impossible to apply to families comprising of more than a few classes. Additionally, it would prevent modular extension of the APIs, as such extension would imply that all type definitions be extended with extra parameters. It is here that our unconventional usage of generics shines, as for trading in a limited amount of safety (cf. Sect. 6), we bring the technique within the realm of practical applicability.

8. CONCLUSIONS

In this paper we have presented a simple technique to prevent mixing of incompatible implementations of a single API. The approach is based on an idiomatic use of generics. The core idea is to add a type parameter to the interfaces of an API, and tie the classes that make up an implementation to the a particular choice of type parameter, which we call an implementation token. In this way methods of the API can only be invoked with arguments that belong to the same implementation, so that mixing of objects from different implementations is statically flagged.

We have explained how to apply the technique and demonstrated it through a simple example. We have addressed the concern that the type parameter used in our approach may prevent interface-based programming: most client code can abstract over the choice of implementation by becoming generic, while creational dependencies can be wrapped inside abstract factories and typed with wildcards. We have also shown how it is possible to provide for code reuse between implementations, and discussed different techniques for controlled and safe mixing of implementations by using wildcards and bounds.

Finally, we have discussed a case study that we have undertaken in order to investigate the feasibility of our technique and gauge its usability in practice. Based on this, we believe that our approach does not pose too much effort to be useful in practice, and that is not too complex to be understood by an average programmer. In fact, we believe for some limited syntactic overhead in the form of type parameters, the technique provides a good degree of early feedback on errors that are potentially hard to find, and makes it easier to spot how and where different implementations may interact.

Acknowledgements

Bill Harrison and David Lievens are being supported by a grant from Science Foundation Ireland. Fabio Simeoni is partly funded by the European Commission in the context of the D4Science project of the FP7 IST priority. We would like to thank the anonymous reviewers for their suggestions that helped improve this paper.

9. REFERENCES

[1] G. Bracha. Generics in the java programming language. http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf.

[2] K. B. Bruce. Some challenging typing issues in object-oriented languages. *Electr. Notes Theor. Comput. Sci*, 82(7), 2003.

[3] K. B. Bruce, M. Odersky, and P. Wadler. A statically safe alternative to virtual types. In E. Jul, editor, *Proceedings of 12th European Conference on Object-Oriented Programming (ECOOP'98)*, volume 1445 of *Lecture Notes in Computer Science*, pages 523–549. Springer Verlag, 1998.

[4] K. B. Bruce, A. Schuett, and R. van Gent. PolyTOIL: A type-safe polymorphic object-oriented language. In W. Olthoff, editor, *Proceedings of 9th European Conference on Object- Oriented Programming (ECOOP'95)*, volume 952 of *LNCS*, pages 27–51, Aarhus, Denmark, Aug. 1995. Springer-Verlag.

[5] E. Ernst. Family polymorphism. In J. L. Knudsen, editor, *Proceedings of 15th European Conference on Object- Oriented Programming (ECOOP 2001)*, number 2072 in LNCS, pages 303–326. Springer Verlag, 2001.

[6] E. Ernst, K. Ostermann, and W. R. Cook. A virtual class calculus. In J. G. Morrisett and S. L. P. Jones, editors, *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, pages 270–282. ACM, 2006.

[2]The term was first used in [5], where did not denote the problem, but the proposed solution.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[8] W. Harrison, H. Ossher, S. Sutton, and P. Tarr. Supporting aspect-oriented software development with the Concern Manipulation Environment. *IBM Systems Journal*, 44(2):309–318, 2005.

[9] R. Johnson, J. Hoeller, A. Arendsen, T. Risberg, and D. Kopylenko. *Professional Java Development with the Spring Framework*. Wrox Press Ltd., 2005.

[10] M. Naftalin and P. Wadler. *Java Generics and Collections*. O'Reilly, 2006.

[11] N. Nystrom, S. Chong, and A. C. Myers. Scalable extensibility via nested inheritance. In J. M. Vlissides and D. C. Schmidt, editors, *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2004, October 24-28, 2004, Vancouver, BC, Canada*, pages 99–115. ACM, 2004.

[12] M. Odersky, V. Cremet, C. Röckl, and M. Zenger. A nominal theory of objects with dependent types. In F. Buschmann, A. P. Buchmann, and M. Cilia, editors, *ECOOP 2003*, number 3013 in LNCS, pages 201–224. Springer Verlag, 2003.

[13] K. Ostermann. Dynamically composable collaborations with delegation layers. In B. Magnusson, editor, *ECOOP 2002 - Object-Oriented Programming, 16th European Conference, Malaga, Spain, June 10-14, 2002, Proceedings*, volume 2374 of *Lecture Notes in Computer Science*, pages 89–110. Springer, 2002.

[14] C. Saito, A. Igarashi, and M. Viroli. Lightweight family polymorphism. *Journal of Functional Programming*, 18(3):285–331, 2008.

[15] The scala programming language. http://lamp.epfl.ch/scala/.

[16] M. Torgersen, C. P. Hansen, E. Ernst, P. von der Ahé, G. Bracha, and N. Gafter. Adding wildcards to the Java programming language. In *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, pages 1289–1296. ACM, 2004.