Softw Syst Model (2017) 16:997–1017
DOI 10.1007/s10270-015-0505-x

REGULAR PAPER

Promoting traits into model-driven development

Vahdat Abdelzad[1] · **Timothy C. Lethbridge**[1]

Received: 22 February 2015 / Revised: 15 September 2015 / Accepted: 21 September 2015 / Published online: 7 November 2015

Abstract Traits, as sets of behaviors, can provide a good mechanism for reusability. However, they are limited in important ways and are not present in widely used programming and modeling languages and hence are not readily available for use by mainstream developers. In this paper, we add UML associations and other modeling concepts to traits and apply them to Java and C++ through model-driven development. We also extend traits with required interfaces so dependencies at the semantics level become part of their usage, rather than simple syntactic capture. All this is accomplished in Umple, a textual modeling language based upon UML that allows adding programming constructs to the model. We applied the work to two case studies. The results show that we can promote traits to the modeling level along with the improvement in flexibility and reusability.

Keywords Reusability · Traits · Modeling · Umple · UML

1 Introduction

Reuse has long been an important objective in software engineering [31]. In this paper, we demonstrate enhanced mechanisms for reuse, building on the concept of traits [52]. We show how traits can be extended to incorporate deeper semantics and modeling abstractions such as UML associations. Furthermore, we demonstrate how this can be accomplished for popular programming languages.

Research in software reuse has brought higher-level languages, components, generative methods, new architectures, and domain engineering [8,23,39]. For the majority of these, abstraction and inheritance play particularly important roles. Inheritance shows itself in various forms such as single inheritance, multiple inheritance, and mixins. However, these variations suffer from conceptual and practical problems. For instance, there is a troublesome situation named the "diamond problem" or "fork-join inheritance" for multiple inheritance [17,22,49,53]. For mixins, there are also problems of linear composition, dispersal of glue code, and fragile inheritance [52]. Linear composition expresses that we may find a situation in which the total order of mixins does not exist. Disposal of glue code indicates that for conflict resolution, sometimes developers need to modify the mixins, introduce new mixins, or use the same mixin twice.

In order to overcome these problems, the concept of traits was introduced by Schärli et al. [21,52]. A trait, in its original form, is a group of pure methods that serves as a building block for classes; it is a simple but powerful unit of code reuse. Traits can be used to structure object-oriented programs in a compositional manner. They have been implemented natively in programming languages such as Squeak [28] and PHP [54].

Before proceeding, we want to clarify that in this paper we are *not* talking about the semantically distinct notion in C++ called traits classes [24,38]. These are as small objects which carry information used by other objects to determine implementation details. For example, they can be used to indicate whether or not a type is "void" inside a C++ program. The use of the term 'traits' in this paper follows the usage of the broader object-oriented literature.

Communicated by Prof. Hong Mei.

✉ Vahdat Abdelzad
v.abdelzad@uottawa.ca

Timothy C. Lethbridge
tcl@eecs.uottawa.ca

1 School of Electrical Engineering and Computer Science, University of Ottawa, Ottawa, Canada

Several research projects have extended the original concept of traits [9,11,18,20,35,41,44,45,60]. The majority of these projects have involved either applying traits in different languages or else showing new applications of traits. Meanwhile, model-driven technologies are making inroads into the development community, albeit slowly. Modeling abstractions such as state machines and associations provide new opportunities for reuse and can be combined with inheritance for even greater reusability. However, the issues with inheritance described earlier apply also when these new abstractions are inheritable units; this suggests that traits and models ought to be able to be synergistically combined.

The current situation is that most developers are unable to use traits in their designs and obtain the benefits of traits' flexibility because they are not readily available in mainstream programming languages such as C++ and Java, or modeling languages such as UML. As a result, software systems have a greater amount of duplication than they otherwise might. In order to tackle the indicated issues and provide traits with advanced features, we have extended classic traits in three ways:

(a) We allow them to include modeling elements, most particularly UML associations, state machines, and constraints. In this paper, we limit our discussion of modeling abstractions to associations.
(b) We allow developers to specify *required interfaces* that clients of traits must implement, extending the preexisting concept of required methods. Required interfaces force clients of traits to implement those interfaces in order to use a particular trait. Designers can either build hierarchical required interfaces for better consistency and more reusability or use already-existing interfaces to build needed traits.
(c) We enable traits to be used in languages such as Java and C++.

Taken together, these enhancements allow traits to be used in mainstream development by both modelers and programmers. Enhanced traits can be used to help build libraries of reusable models or code and can reduce duplication and accidentally duplicated defects.

We demonstrate these capabilities by applying them to Umple [4–7,33], which is a textual modeling language and permits embedding of programming concepts into models. In our research, we are *not* trying to introduce traits into the UML (which is a graphical language), in part because we can reach greater levels of expressiveness in a textual modeling notation. Umple follows UML semantics, but has many features such as mixins and aspects that leverage its textual form. Traits are another such text-oriented feature.

The novelty of our research can be summarized as (i) we brought the concept of traits to the modeling level and extended it with the modeling concepts in order to enable its use in the model-driven development context; (ii) we have created a textual syntax for traits in that modeling context; and (iii) we have a working model compiler and have demonstrated the use of traits in real systems created from model-generated code.

The rest of this paper is organized as follows. Section 2 expands on the motivation for our research. Section 3 describes the Umple technology we have used to implement our approach. The basic features of classic traits in addition to our extensions are described in Sect. 4. We focus on evaluation of the proposed features through two case studies in Sect. 5. Work related to our research is presented in Sect. 6. Section 7 depicts some of our research challenge. We discuss reusability arising from traits at the modeling level in Sect. 8. Finally, we present conclusions and future work.

2 Motivation

Modeling elements, e.g., state machines and associations, are reusable elements and can be inherited through the classes in which they are available. This means that they suffer from the same inheritance difficulties as other constructs found in programming languages such as methods. The concept of traits has shown in programming languages that they can overcome some of these difficulties. We wanted to explore their usefulness in modeling languages, to see if they could offer the same benefits—a way to better modularize and improve reuse, and a way to solve some of the problems surrounding avoidance of multiple inheritance. Furthermore, if traits are to be used in modeling, their syntax and semantics need to mesh well with other modeling elements. We wanted to explore the effect of incorporating model-level constructs such as associations into traits.

On the other hand, there is no support for traits in most popular languages such as Java and C++, so we wanted to investigate the ability of implementing model-based traits in these languages. In other words, we wanted to explore whether or not traits can be a part of platform-independent models and be available in platform-specific models as well without concerns about not having native support for traits in those languages.

3 Umple

In order to explore the notion of model-level traits and show feasibility of the features and extensions, we have implemented them in Umple [4,56]. Umple is an open-source modeling tool and programming language family which enables us to have what we call model-oriented programming. It brings abstractions such as associations, state machines, and constraints derived from UML [59] to object-oriented programming languages. It also allows developers to use aspect orientation, design patterns, tracing, and now traits at the modeling level without worries about their real implementation in different target languages. All concepts in Umple are defined textually, and the textual language follows the syntactic convention of C-family languages. Despite begin textual, Umple also provides a way to define and represent them visually. A cloud-based interface of Umple can be found at [57] which covers both textual and graphical representation. There is much published literature on Umple [5–7,33], so we provide the following limited overview to allow the reader to understand the notation in this paper.

Listings 1–10 show examples of Umple. The primary top-level entities are classes, interfaces, and traits. Each such entity is declared using a keyword (‘class’, ‘interface’, or ‘trait’) followed by the name of the entity and then matching braces surrounding a series of elements. The elements inside the top-level constructs can include attributes (declared in a manner similar to variables, but implying additional semantics), methods (declared as in other C-family languages), associations, constraints, isA (generalization) directives, stereotypes, and state machines. An example of a class with an association is:

```
class Club {0..1 -- * Person member;}
```

The 0..1 and * are multiplicities as in UML. The word ‘member’ is an optional role name given to the association. The ‘–’ means a bidirectional association; associations navigable in one direction only would be shown as ‘->’ or ‘<-’. A constraint is a Boolean condition surrounded by square brackets. An isA directive specifies generalization (single inheritance) or interface implementation. The word ‘isA’ is followed by the superclass and/or interface(s). Umple adopted this convention to allow designs to easily change from one form of inheritance to another. As we will see, we adopt the same notation for trait inclusion.

4 Details of the approach

In this section, we first explain (in Sect. 4.1) the already-existing features of traits that come from their base in programming languages. Afterward, we explain our contributions to traits in terms of associations and required interfaces (in Sects. 4.2, 4.3). We initially provide very simple abstract examples illustrating existing features of traits, and our own enhancements. Then, to fully explain the use of our enhanced traits in the context of a real system, we describe the design of a graphical system based on traits in Sect. 4.4 . Finally, Sect. 4.5 presents various automatic code generation mechanisms for traits and describes our own implementation.

As we mentioned previously, we are not purporting to add traits to UML. We are introducing them as a textual modeling concept in the Umple language that strongly aligns with UML but is not UML itself. That said, we have created a graphical representation to help visualize the traits. This is not intended to be a proposed UML extension, or even a formal contribution to the paper; it is merely intended to help explain the use of traits and help readers to understand them. In our approach, diagrams are generated and not intended to be drawn using a drawing tool. We do not make any claims about whether it would be usable or useful to edit the diagrams or use them to create traits in models.

4.1 Basic trait capabilities

As originally introduced, traits consist of sets of *provided methods* and *required methods* [51,52]. Clients of a trait declare they use the trait, in which case the contents of the trait’s provided methods become logically part of the client. The provided methods have access to functionality in these required methods of clients. Clients, which can be either other traits or classes, must have their own implementations of each of the required methods. Access to any of the clients’ attributes has to occur through accessor methods (i.e., set and get methods, which form a subset of the required methods).

If the client of a trait is a class, also called a *final client*, that class must satisfy the required methods either by itself or by inheritance from its superclasses. Final clients can use any number of traits and have to satisfy all required methods of all used traits.

When the client of a trait is another trait, also called a *composed* trait, there are two ways by which required methods of the composing trait can be satisfied. Either the composed trait will satisfy them by its provided methods or else the final client of composed trait will satisfy them. When a composed trait does not satisfy required methods of the composing traits, those required methods become part of its own required methods. Determination of whether a *required* method is satisfied in a client is performed by matching method name, return type, and parameters; visibility such as public and private is not relevant.

Listing 1 and Fig. 1 illustrate these cases. In Umple, traits are defined by the keyword “trait”. Trait attributes and provided methods are defined similarly to how attributes and methods are defined in classes (with visibility, name, return type, parameters, and body). Provided methods, however, cannot be abstract. If methods are defined as abstract, they are considered to be *required* methods.

There is support for automatically drawing diagrams incorporating traits in Umple. The diagram representation is similar to class diagrams in UML, but there are two sections for methods. The right section shows required methods, while the left section shows provided methods. Umple also

Listing 1. Satisfactions in traits

```
1  class Class1{
2   void method3(){/*details omitted*/ }
3  }
4  class Class2{
5   isA Class1;
6   isA Trait2;
7   void method2(){/*details omitted*/ }
8  }
9  trait Trait2{
10  isA Trait1;
11  abstract void method3();
12  void method1(){/*details omitted*/ }
13  void method2(){/*details omitted*/ }
14 }
15 trait Trait1{
16  abstract void method1();
17  abstract void method2();
18  void method4(){/*details omitted*/ }
19 }
```

Listing 2. Conflicts when using traits

```
1  class Class1{
2   isA Trait1, Trait2;
3  }
4  trait Trait3{
5   public void method1(){/*details omitted*/ }
6   public void method2(){/*details omitted*/ }
7  }
8  trait Trait1{
9   isA Trait3;
10 }
11 trait Trait2{
12  isA Trait3;
13  public void method2(){/*details omitted*/ }
14 }
```

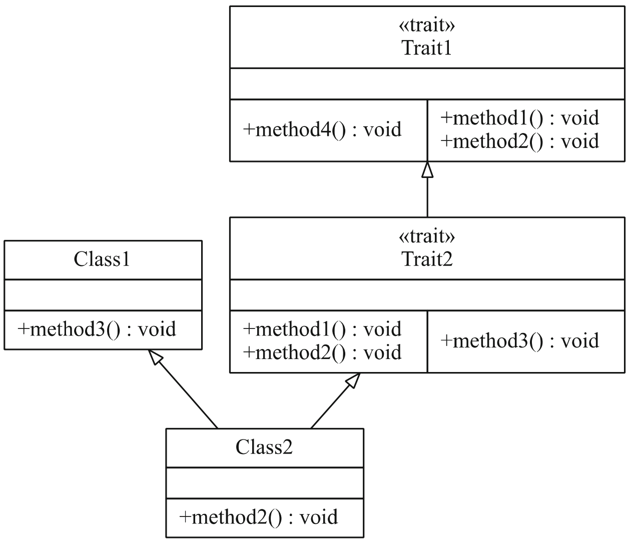

Fig. 1 The diagram corresponding to Listing 1

provides a mechanism to visualize the system as a normal class diagram after the traits have been applied or 'flattened'. Developers can easily switch between these two views to have better understanding of the system design.

In Listing 1, there are two classes Class1 (lines 1–3) and Class2 (lines 4–8), and two traits Trait2 (lines 9–14) and Trait1 (lines 15–19). Class Class2 is a subclass of class Class1 (line 5) and uses the composed trait Trait2 (line 6). Trait Trait2 uses the composing trait Trait1 (line 10) and has one required method method3() (line 11) and two provided methods method1() and method2() (lines 12–13). Trait Trait1, which is a composing trait in this example, has two required methods method1() and method2() (lines 16–17); one provided method method4() (line 18). Required methods of trait Trait1 are satisfied by provided methods of trait Trait2. In addition, the required method method3() of trait Trait2 is satisfied by the method in superclass Class1.

When a composed trait is used by a class, there is a situation in which certain required methods of composing traits can be satisfied by both the class and the composed trait. In this case, satisfactions (or implementations) that come from the class will take priority and be considered the valid ones for those required methods. In Listing 1, for instance, required method method2() of trait Trait1 is satisfied by the provided method of trait Trait2 and the method of class Class2. In this case, the provided method of trait Trait2 will be disregarded. Therefore, the result of the single inheritance and using a trait for class Class2 is four concrete methods method1(), method2(), method3(), and method4() in which concrete method method2() of trait Trait2 does not play any role.

Using traits is not always a straightforward mechanism, and sometimes there are conflicts. Provided methods are the main reasons for conflicts when they appear in clients, derived from different traits and hierarchy levels. If a method with the same signature comes to a client from two different traits, it is considered as a conflict and must be resolved. However, if the method comes from two different traits but with a common source (i.e., both different traits use a common third trait), it is not considered as a conflict. Conflicts are detected in our implementation automatically.

Listing 2 illustrates a symbolic example of two cases, in which one of them results in a conflict and another does not. Class Class1 uses two traits (line 2), and it, hence, will get two methods—method1() and method2() (lines 5–6)—twice, coming from two different traits Trait1 and Trait2. There is no conflict for method1() because Trait1 and Trait2 get the same method from a common source (Trait Trait3).

However, this is not the case for the method2() because this method has been overridden by trait Trait2 (line 13). In other words, the sources are now not the same. Therefore, in class Class1, there is a conflict. It is important to note that there is no graphical representation for this example because

Umple detects the conflict and does not allow generation of an inconsistent diagram.

There are two mechanisms to resolve this kind of conflict. The first one is aliasing (or renaming), and the second is removing. Aliasing allows renaming one of the conflicting methods, so the client can have access to both methods. Removing lets the developer remove one of the conflicting methods, so it will not be accessible anymore in the client and hence avoid the conflict. Choosing one of these two methods depends on the structure of the application and the developer's needs. In order to resolve the conflict in Listing 2 through renaming, the following is the Umple syntax that must replace the syntax in line 2 of class Class1:

```
isA Trait1, Trait2 < -method2() >;
```

The minus sign indicates 'remove'; this will, hence, result in not having the method method2 (line 13), which comes from trait Trait2, and just having the method with the same name coming from trait Trait1. The same conflict can also be resolved by using renaming in which a new name method3() will be given to one of the conflicting methods. In addition, it is also possible to change visibility of the renamed method in order to restrict access to it. The syntax, which must be used in class Class1 instead of the line 2, is as follows. This syntax also makes the renamed method private so as not to let other classes have access to it.

```
isA Trait1, Trait2 < method2() as private method2() >;
```

Traits can also be more general, which is achieved through template parameters. Template parameters are a technique to increase the genericity and hence flexibility and reusability of various elements. In C++ and Java, one can specify generic classes, interfaces, and operations [41]. One can also model UML elements with unbound formal parameters that can be used to define families of classifiers, packages, and operations. This feature is also applied to traits to substantially increase reusability of traits and broaden their appeal. Currently, template parameters are just supported by Scala, which is at the implementation level. Our approach,

Listing 3. Traits without template parameters

```
interface Interface1{
  void method1();
}
class Class1{
  isA Interface1, Trait1;
  void method1(){}
  Class1 method2(Class1 data){/*details omitted*/}
}
class Class2{
  isA Interface1, Trait2;
  void method1(){/*details omitted*/}
  Class2 method2(Class2 data){/*details omitted*/}
}
trait Trait1{
  abstract Class1 method2(Class1 data);
  String method3(Class1 data) {/*details omitted*/}
}
trait Trait2{
  abstract Class2 method2(Class2 data);
  String method3(Class2 data) {/*details omitted*/}
}
```

implemented in Umple, covers template parameters at the modeling level and combines these with other modeling elements such as associations. This combination increases modularity and reusability to an extent that is not achievable at the implementation level.

Template parameters can be referred to in required and provided methods and attributes. Traits can have template parameters with generic or primitive data types. In Umple, primitive types include Integer, Float, and String. Generic types include classes and interfaces. The difference is that it is possible to put restrictions on bound values of generically typed parameters. Such restrictions might include a declaration that the interfaces or classes must be extended or implemented by bound values. These restrictions are only available for generic-type parameters because primitive types cannot implement or extend any other types and so there is no way of imposing such constraints on them.

Listing 3 and Fig. 2 show an example in which template parameters have not been used. In Listing 3, there are two traits Trait1 and Trait2 each having a method called method2() but with different return and parameter types.

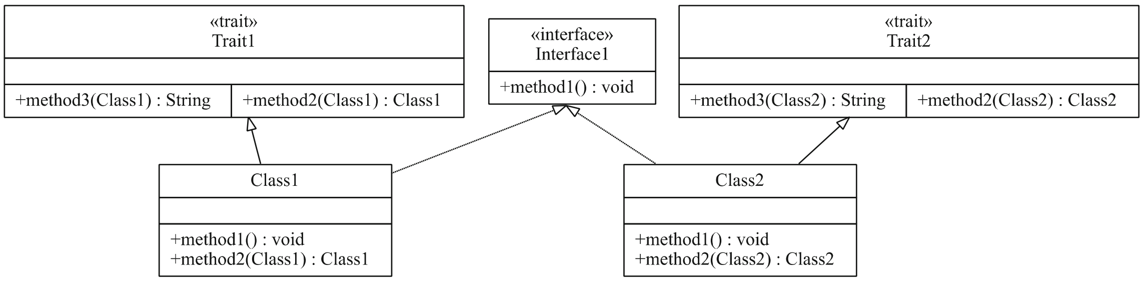
«trait»
Trait1
+method3(Class1) : String
+method2(Class1) : Class1
«interface»
Interface1
+method1() : void
«trait»
Trait2
+method3(Class2) : String
+method2(Class2) : Class2
Class1
+method1() : void
+method2(Class1) : Class1
Class2
+method1() : void
+method2(Class2) : Class2

Fig. 2 The diagram corresponding to Listing 3

Listing 4. Traits with template parameters

```
1   interface Interface1{
2     void method1();
3   }
4   class Class1{
5     isA Interface1, Trait1<Type = Class1>;
6     void method1(){}
7     Class1 method2(Class1 data){ /*details omitted*/}
8   }
9   class Class2{
10    isA Interface1, Trait1<Type = Class2>;
11    void method1(){}
12    Class2 method2(Class2 data){ /*details omitted*/}
13  }
14  trait Trait1< Type isA Interface1 > {
15    abstract Type method2( Type data);
16    String method3( Type data) {/*details omitted*/}
17  }
```

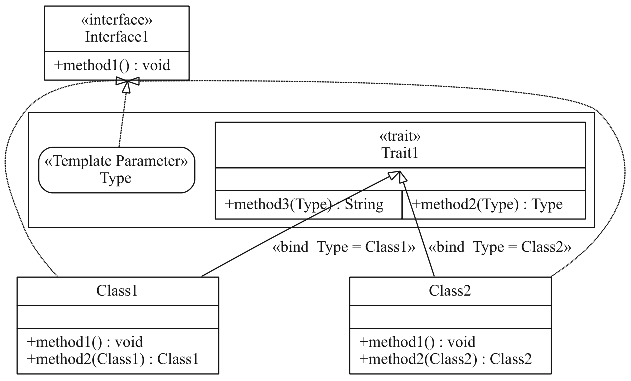

Fig. 3 The diagram corresponding to Listing 4

Their provided methods are also essentially identical, but operating on different implementations of interface Interface1 (Class1 and Class2) through the parameter "data". Because of not using template parameters, the design is required to have two traits in this specific case.

Listing 4 and Fig. 3 illustrate how using templates can allow us to create a single generic trait to replace the two traits in Listing 3. A template parameter Type defined in line 14 is restricted to bind to something implementing interface Interface1. The restriction is applied by the keyword "isA", which follows the name of parameter (in a case where there was more than one restriction, the restrictions would be conjoined by the symbol "&"). In trait Trait1, parameter Type is used for return value and parameter types of provided and required methods. When a generic trait is used in a client class or trait, it is necessary to bind values for all parameters. The binding for trait Trait1 occurs in class Class1 (line 5) and Class2 (line 10). In class Class1, value Class1 has been bound to parameter Type, while value Class2 has been bound to it in class Class2.

4.2 Required interfaces

Required functionality of classic traits is defined in terms of required methods. However, there are shortcomings with this approach. The first shortcoming is that there is no way to reuse a list of required methods. For example, consider a case in which there are traits that happen to have the same set of required methods but different provided methods. In this case, there is duplication due to repeated listing of the same methods. Also, if there are several traits that must always have the same list of required methods, inconsistency could be introduced in the design by changing just one of them and not all.

Listing 5 and Fig. 4 describe an example of this shortcoming. As can be seen, all traits have two common required methods method1() and method2() (lines 13–14, 18–19,

Listing 5. Duplication and potential inconsistency in required methods

```
1   class Class3{
2     void method1(){/*details omitted*/ }
3     double method2(){/*details omitted*/ }
4   }
5   class Class1{
6     isA Class3, Trait1;
7   }
8   class Class2 {
9     isA Class3, Trait3;
10    boolean method3(){/*details omitted*/ }
11  }
12  trait Trait1{
13    abstract void method1();
14    abstract double method2();
15    float method3(){/*details omitted*/ }
16  }
17  trait Trait2{
18    abstract void method1();
19    abstract double method2();
20    float method4(){/*details omitted*/ }
21  }
22  trait Trait3{
23    isA Trait1,Trait2;
24  }
25  trait Trait4{
26    abstract void method1();
27    abstract double method2();
28    abstract boolean method3();
29    float method5(){/*details omitted*/ }
30  }
```

and 26–27). They also have different provided methods method3() (line 15), method4() (line 20), and method5() (line 29). The required methods of traits Trait1 and Trait2 must always be kept the same because we want to have composed trait Trait3 (line 22), which brings together provided methods of two traits (line 23). These provided methods must always achieve their functionality from the same required methods. Since traits Trait1 and Trait2 are two separate traits and can be extended separately, there is no way guarantee those required methods will be kept the same during software maintenance. In other words, there is the possibility one of those traits

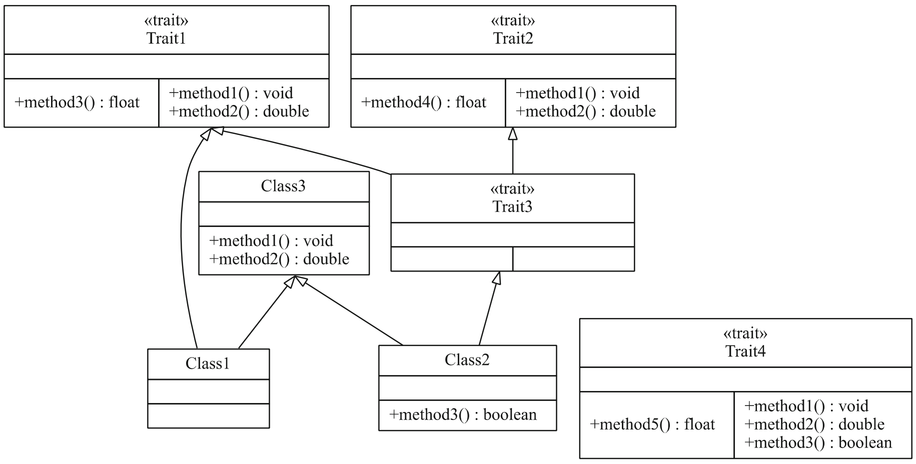

Fig. 4 The diagram corresponding to Listing 5

Listing 6. Traits with incomplete set of required methods

```
1   interface Interface1{
2     void method1();
3     double method2();
4   }
5   class Class1{
6     isA Interface1, Trait1;
7     void method1(){/*details omitted*/ }
8     double method2(){/*details omitted*/ }
9     void method3(){/*details omitted*/ }
10    float method4(){/*details omitted*/ }
11  }
12  class Class2{
13    isA Trait1;
14    void method3(){/*details omitted*/ }
15    float method4(){/*details omitted*/ }
16  }
17  trait Trait1{
18    abstract void method3();
19    abstract float method4();
20    void method5(){/*details omitted*/ }
21  }
```

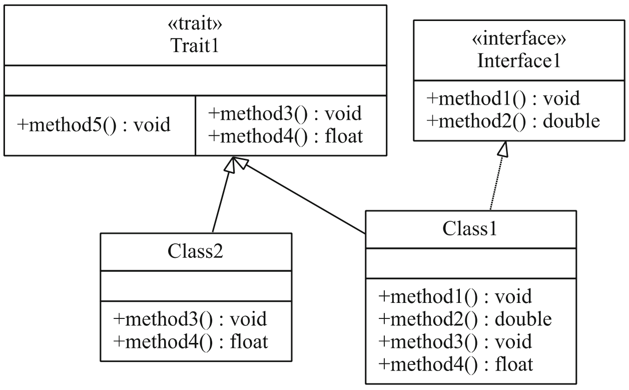

Fig. 5 The diagram corresponding to Listing 6

might be modified without applying the change to the other one.

The second shortcoming appears when we know the clients of traits and that they must implement certain interfaces in order to have a correct implementation for the required methods. This suggests that it might be a good idea to put a restriction on clients that specify the interfaces they must implement. Such a restriction would ensure that traits are not used in clients that just happen to have methods with the same signature. This is important because having reusable elements which work correctly with minimum errors should be the designers' responsibility and not the user's [37]. Later, we will show a solution to this.

Listing 6 and Fig. 5 depict this shortcoming, in which two classes Class1 (lines 5–11) and Class2 (lines 12–16) have the ability to satisfy required methods of the trait Trait1. In addition, imagine that the correct satisfaction of required methods depends internally on clients which implement interface Interface1 (lines 1–4). According to this, trait Trait1 should not be used by class Class2, while it has been used. This gives rise the idea that there should be a mechanism to let traits specify more precisely what they want.

In traditional traits, there is no straightforward way to apply a mechanism to avoid this issue. Of course, it can be done implicitly by defining all abstract methods of interfaces as required methods, but again in that case there will be duplication and the issue of inconsistency. Furthermore, this makes traits have lots of required methods, which results in less readable and understandable traits.

In order to address these issues, we extend traits with *required interfaces*. Using these, traits can either put extra restrictions on clients or just manage their required methods in a more modular and reusable way. Traits may use already-existing interfaces or new interfaces may be written to accomplish the desired modularization. Furthermore, developers will be able to create a hierarchy of interfaces to optimize the reusability.

The new design for the example in Listing 5 is shown in Listing 7 and Fig. 6. There is now a hierarchical design for required methods in terms of interfaces, making it reusable and consistent. Traits T1 and T2 now have the same required

Listing 7. Introduction of required interfaces to reduce duplication and inconsistency

```
 1  interface Interface1{
 2    void method1();
 3    double method2();
 4  }
 5  interface Interface2 {
 6    isA Interface1;
 7    boolean method3();
 8  }
 9  class Class3{
10    void method1(){/*details omitted*/ }
11    double method2(){/*details omitted*/ }
12  }
13  class Class1{
14    isA Class3, Interface1, Trait1;
15  }
16  class Class2 {
17    isA Class3, Interface2, Trait3;
18    boolean method3(){/*details omitted*/ }
19  }
20  trait Trait1{
21    isA Interface1;
22    float method3(){/*details omitted*/ }
23  }
24  trait Trait2{
25    isA Interface1;
26    float method4(){/*details omitted*/ }
27  }
28  trait Trait3{
29    isA Trait1,Trait2;
30  }
31  trait Trait4{
32    isA Interface2;
33    float method5(){/*details omitted*/ }
34  }
```

interface (lines 21 and 25), and if there is a modification in the required interface, it will be applied to both. Classes C1 and C2 have to implement interfaces I1 (line 14) and I2 (line 17) to be able to use traits T1 and T2.

Meanwhile, the issue described in Listing 6 can be easily resolved by the extension through adding required interface Interface1 to trait Trait1. In this case, class Class2, which does not implement the interface, will not be able to use trait Trait1 and this is exactly what the designer of the trait wanted. The whole process of detection and satisfaction is done automatically in our implementation.

4.3 Associations

An association is a useful mechanism at the modeling level that specifies relationships among instances of classifiers. An association implies the presence of certain variables and provided methods in both associated classifiers. Other methods as well as traits can hence refer to the implied methods. Since an association can be seen as a set of implied provided methods, it would be logically possible to extend the concept of traits to incorporate associations. However, prior to our work, there was no such a mechanism in traits.

An association can only be between a class and another class or between a class and an interface; in other words, defining an association between two interfaces is not allowed. This must also be accounted for in the definition of traits. The reason for this is that associations imply that at least one end

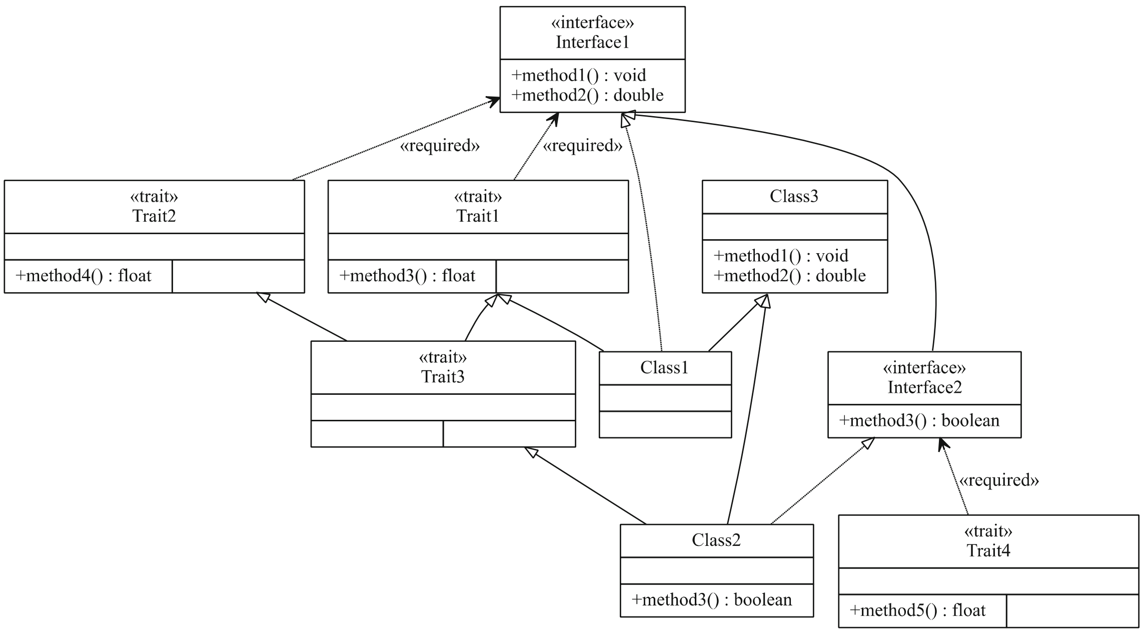

Fig. 6 The diagram corresponding to Listing 7

Listing 8. An issue with associations among classes

```
1   class Class1{
2    0..1 -- * Class2;
3    //details omitted.
4   }
5   interface Interface1 {
6    //details omited.
7   }
8   class Class2{
9    isA Interface1;
10   //details omitted.
11  }
```

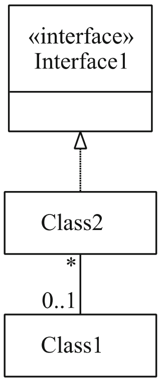

Fig. 7 The diagram corresponding to Listing 8

must maintain the state of the links between instances. For unidirectional associations (navigable in one direction only), only one end maintains this state, so the other end can be an interface. For bidirectional associations, both ends must maintain the state, so both ends must be classes.

Having associations in classes is considered as a kind of limitation on fine-grained reusability because such classes cannot then be used alone in other systems. This happens because the other systems also need the associated classes or interfaces. Furthermore, the nature of associations is defined based on exact names. When class A, for example, with an association with another class or interface B is to be used in a different system, then that system must have a class or interface with the exact name B. In order to avoid this issue, we extend traits to have associations with template parameters.

Listing 8 and Fig. 7 show an example explaining the case in which a class cannot be reused alone. In class Class1 (lines 1–4), there is a bidirectional association between Class1 and Class2. Class Class2 (lines 8–11) implements interface I (lines 5–7). If we would like to use class Class1 in another system, we have to transfer class Class2 and interface I as well. There might be a class in the new system that can satisfy all features of class Class2, but we cannot use it because it is forced to have exactly the same name in the new system. It is also impossible to change the name of compatible class to Class2 because there will be inconsistency among elements

Listing 9. Reusable associations in traits

```
1   class Class1{
2    isA Trait1<RelatedClass=Class2>;
3    //details omited.
4   }
5   interface Interface1 {
6    //details omited.
7   }
8   class Class2{
9    isA Interface1;
10   //details omitted.
11  }
12  trait Trait1 <RelatedClass isA Interface1> {
13   0..1 -- * RelatedClass;
14   //details omitted.
15  }
```

of the new system. In addition, if we change the name and apply it to all other parts of the system, the new name may be out of the domain of the system and so create understanding challenges. The same issues related to class Class1 can happen to class Class2 because it depends on class Class1.

In order to avoid the limitations in Listing 8, we redesign it with a trait depicted in Listing 9 and Fig. 8. There are again two classes Class1 and Class2 and an interface Interface1. We added a trait (lines 12–13) with a template parameter RelatedClass restricted to implement interface I. Furthermore, we added the same association which was in class Class1 in Listing 8 to the trait, but parameter RelatedClass was substituted for concrete name Class2. This association applied to class Class1 in line 2, in which class Class2 has been bound to parameter RelatedClass. As a result, the association is available for both classes Class1 and Class2. In this case, if we want to use class Class1 in another system, we do not need to have exactly a class named Class2. We need a class that implements interface Interface1, and we simply need to bind it to parameter RelatedClass. The name of the class is not fixed anymore in this approach, and the proper candidate from the target system can be used with class Class1. Class Class2 can be reused independently too because there is no concrete relationship between it and other elements of the system.

Another solution that could partially resolve the issue in Listing 8 would be to establish an association with interface Interface1. Then, in the new system, the candidate would be forced to implement interface Interface1. However, in that case, it would be necessary to change the type of the association from bidirectional to directional, because having a bidirectional association between a class and interface is not allowed.

Adding genericity ensures there is no concrete associated element that needs to be available when traits are used in different systems. The associated elements will be replaced with ones that are accessible in the new systems through binding. Of course, the restriction on types of binding values

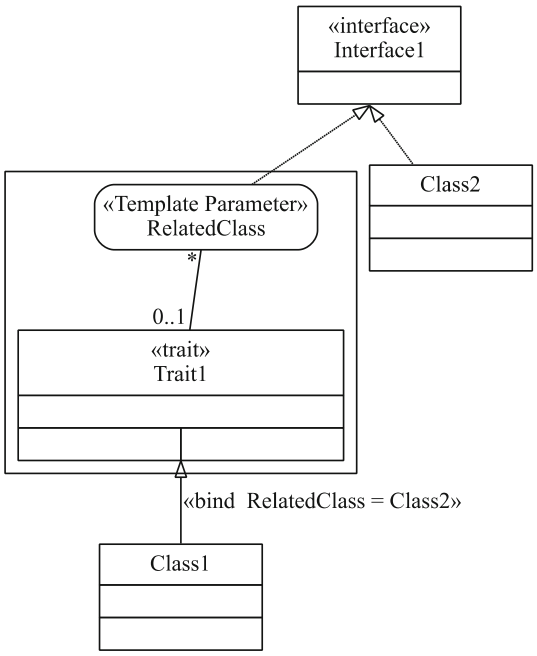

Fig. 8 The diagram corresponding to Listing 9

for the association is achieved by having restriction on parameters. There is now no need for exact names when traits are used. In fact, traits encapsulate associations and when they are used they apply associations to suitable elements through parameters.

When an association is defined, an API (set of methods) is generated in the class to allow such actions as adding, deleting, and querying links of associations. Traits may use this API inside provided methods to achieve their needed implementation. With this approach, we increase cohesion because we will have more related provided methods inside traits and reduce coupling at design time because of having template parameters. Furthermore, the abstraction level of traits will be increased because we will not use attributes to establish relationships and instead utilize the more abstract notion of associations.

Listing 10 and Fig. 9 depict a simple version of the observer pattern [25] implemented based on traits. As can be seen in line 7, the concept of subject in the observer pattern has been implemented as trait Subject, which gets its observer as a template parameter. A direct association has been defined in trait Subject (line 8) which has multiplicity of zero or one on the Subject side and zero or many on the Observer side. This association lets each subject have many observers, and it also applies the case in which observers do not need to know the subject. The trait has encapsulated the association between the subject and observers and then applies it to proper elements when it is used by a client.

As each subject must have a notification mechanism to let observers know about changes, there is a provided method notifyObservers() for this. This method obtains access to all observers through the association. Two classes Dashboard and Sensor play the roles of observer and subject. Class Dashboard has a method named update(Sensor) (line 2) used by the future subject to update it. Class Sensor obtains the feature of being a subject through using trait Subject and binding Dashboard to parameter Observer.

Listing 10. Observable pattern with traits and their associations

```
1   class Dashboard{
2    void update (Sensor sensor){ /*implementation*/ }
3   }
4   class Sensor{
5    isA Subject< Observer = Dashboard >;
6   }
7   trait Subject <Observer>{
8    0..1 -> * Observer;
9    void notifyObservers() { /*implementation*/ }
10  }
```

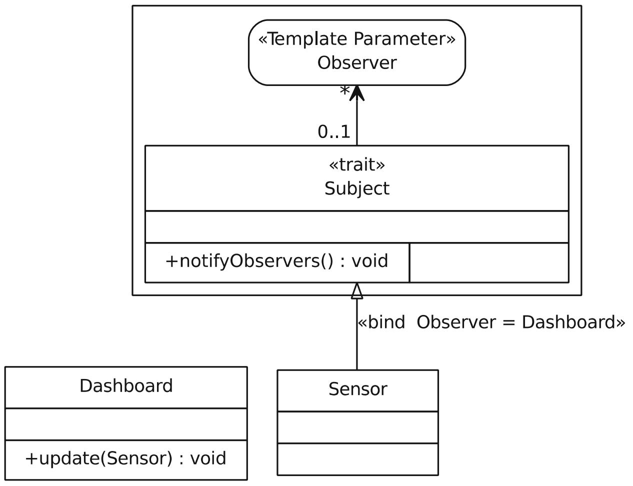

Fig. 9 The diagram corresponding to Listing 10

4.4 Case study: a geometric system

In this subsection, we present a case study developed to help illustrate the use of our enhanced traits. The main goals of the system are to show how our proposed features can be used and to help the reader appreciate their usefulness. It is important to note that we believe traits should be used when they bring benefits regarding flexibility, reusability, and avoiding multiple inheritance. It is possible to make more extensive use of traits, to the point of using them to introduce for every single method. We have not chosen that style, instead using traits when they are useful, and using classic object-oriented design when it already works well.

In Fig. 10, a part of this system's hierarchy is depicted. We have simplified the case study for this paper, hiding classes, leaves, attributes, and methods that are not necessary for the discussion about traits we provide.

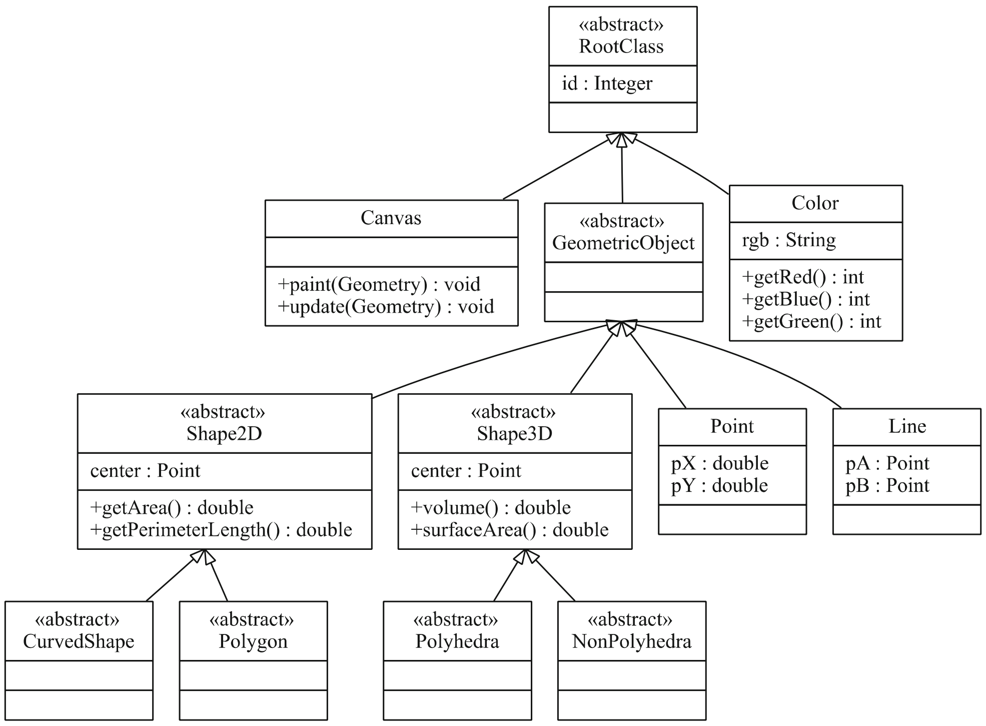

Fig. 10 The hierarchy of the graphical system

Full code and diagrams of the complete case study can be viewed in UmpleOnline [57]. In the Load menu, select the Geometric System. By default a class diagram is shown, without methods and/or traits. To show methods and traits, go to the Options menu and click on the appropriate checkboxes.

As shown In Fig. 10, there is a superclass named RootClass with three subclasses including Canvas, GeometricObject, and Color. The class Canvas is responsible for drawing geometric objects, and the class Color keeps the necessary features related to color. The class GeometricObject is an abstract class for all geometric objects and has four subclasses including Shape2D, Shape3D, Point, and Line. The class Shape2D is a superclass for two classes, CurvedShape and Polygon. The class Shape3D is also a superclass for two classes, Polyhedra and NonPolyhedra. These abstract classes have their own subclasses (leaves). For example, the classes Circle, Rectangle, Sphere, and Cube are subclasses of the classes CurvedShape, Polygon, NonPolyhedra, and Polyhedral, respectively. The class Line is straight, has no thickness, and extends in both directions without end.

One of the features that the system must have is to allow comparing two objects (e.g., shapes and color) regarding being equal or not. We call this the equality feature. For example, whether or not two points are equal. Furthermore, the system must also allow comparing objects regarding being bigger or smaller. We call this the comparability feature. For instance, whether or not a cube is smaller than another cube. There are cases in which we cannot have the comparability feature for classes and they must have just the equality feature. However, all classes that need the comparability feature must also have the equality feature. For example, points and lines can merely have the equality feature while a circle must have both features. In this system, there are also some other classes that do not need these features (e.g., Canvas).

Listing 11. The traits related to the equality and comparability features

```
trait TEquality<TP1>{
  abstract boolean isEqual(TP1 object);
  boolean isNotEqual(TP1 object){/* impl… */}
}
trait TComparable<TP2>{
  isA TEquality<TP1=TP2>;
  abstract boolean isLessThan(TP2 object);
  boolean isLessAndEqual(TP2 object) {/*impl… */}
  boolean isBiggerThan (TP2 object){/* impl… */}
  boolean isBiggerAndEqual(TP2 object){/* impl… */}
}
```

In order to implement these features, we have designed two traits named TEquality and TComparable. Their Umple code is depicted in Listing 11. The trait TEquality provides a provided method named isNotEqual() and requires a required method named isEqual(). It also has a template parameter named TP1 used in the arguments of the required and provided methods. This feature allows us to have the same or suitable type for the arguments of methods which are going to be used in clients. This removes casting of types in the

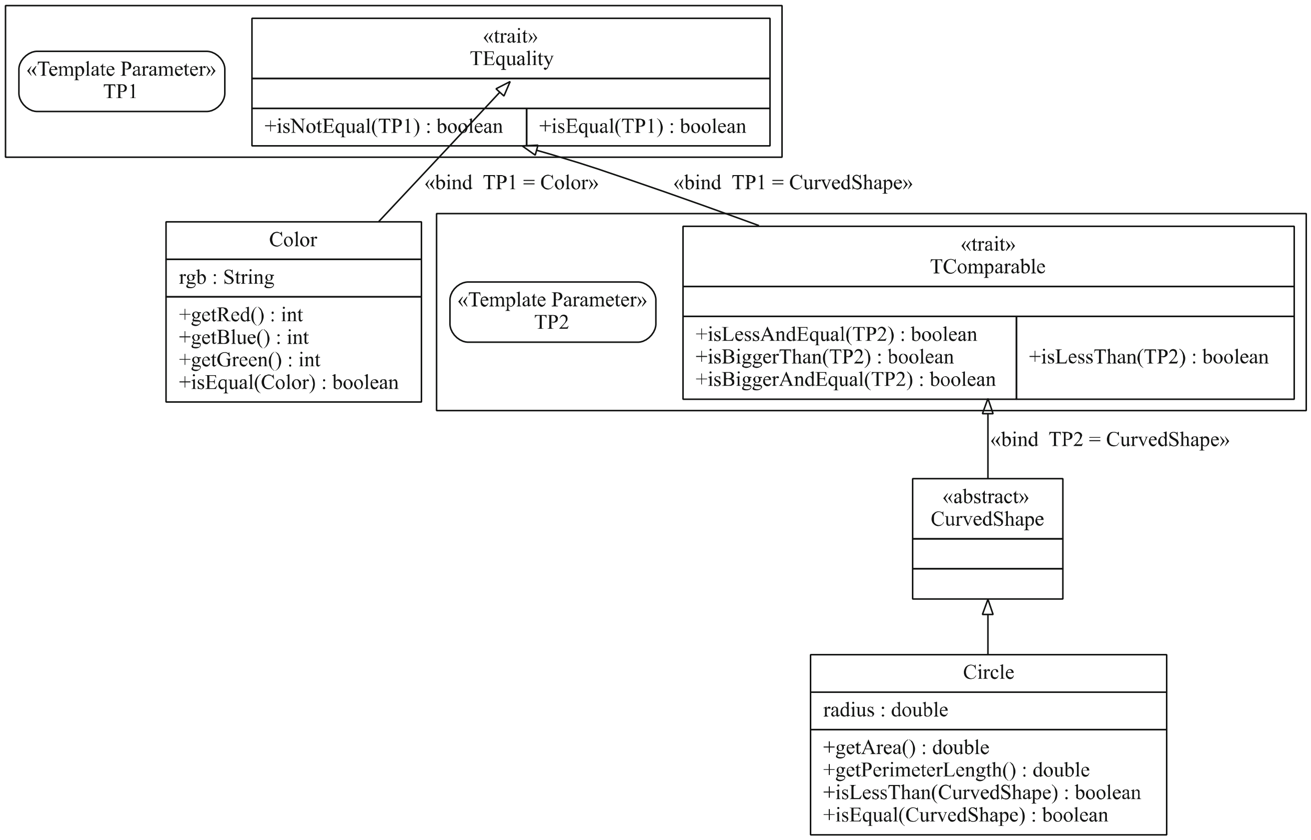

Fig. 11 Use of the trait TEquality and TComparable

body of methods and provides static type-checking during the compilation.

The trait TComparable uses trait TEquality to obtain functionality needed for its provided methods in addition to the required method named isLessThan(). It provides three provided methods named isLessAndEqual(), isBiggerThan(), and isBiggerAndEqual(). It also has a template parameter named TP2 passed into the parameter of TEquality. In other words, this trait has one required method in the body and obtains another one through the trait TEquality. It also has three provided methods in the body and obtains one through the trait Equality. To give the exact or proper type to the template parameter of these traits, they should be applied to the exact class or the first common superclass. Therefore, the trait TEquality is applied to the class Color, Point, and Line, while the trait TComparable is applied to Polygon, CurvedShape, Polyhedra, and NonPolyhedra.

Figure 11 shows a part of the diagram in which the class Color uses the trait TEquality. It implements the required method of the trait, which is isEqual(). The class CurvedShape is abstract and uses the trait TComparable. It does not have enough information to implement the required methods of the trait, so it keeps them as abstract methods and forces leaves to implement them (Circle in this case). It should be indicated that there also are other ways to design and apply these traits. For example, the traits TEquality and TComparable could be completely distinct and we would apply TEquality to the GeometryObject and the trait TComaprable to other mentioned classes. The key point here is that traits can give developers more options while designing systems.

Another feature that we want to have is having color and its related functionality for geometric objects. We also want to have another color for the edge of shapes. However, there are some shapes that mathematically do not have edges. We have designed two traits named TDrawable and TDrawableWithEdge for these purposes. The related Umple code is depicted in Listing 12. The trait TDrawable gives the general meaning of color to all geometric objects, while TDrawableWithEdge provides edge color to appropriate shapes. The trait TDrawable has an association with the class Color and does not have any required method. It provides several provided methods related to color in which three of them are wrappers for methods in the class Color. The traits TDrawableWithEdge use the trait TDrawable and add provided methods related to the color of edges. These two traits do not have any required methods, which shows that it is possible to consider traits as a mechanism to implement libraries. To have these feature in the system, the trait TDrawable is applied to the class GeometricObject because all shapes must have a color. The trait TDrawableWithEdge is applied to the

Listing 12. The traits related to the color feature

```
trait TDrawable {
  0..1 -> * Color color;
  int getRed(){/*implementation */}
  int getBlue(){/*implementation */}
  int getGreen(){/*implementation */}
  void applyTransparency(int p){/*implementation */}
  void applyPattern(int type){/*implementation */}
  void applyColorFilter(int f){/*implementation */}
}
trait TDrawableWithEdge{
  isA TDrawable;
  int getERed(){/*implementation */}
  int getEBlue(){/*implementation */}
  int getEGreen(){/*implementation */}
  void applyETransparency(int p){/*implementation */}
  void applyEPattern(int type){/*implementation */}
  void applyEColorFilter(int f){/*implementation */}
}
```

class Shape2D and Polyhedra because instances of the class NonPolyhedra do not have edges to be colored.

Since the class Canvas should draw geometric objects along with their color, there should be a mechanism to let this class know about the changes in the properties so that it can update the canvas. This feature can be achieved through the Observable pattern. In order to implement this pattern, we reuse the trait Subject introduced in Listing 10. The class GeometricObject uses this trait and assigns the class Canvas as a binding value to the parameter Observer.

The indicated features so far are general features that other projects or classes might also want to use. Having them in terms of traits allows developers to easily reuse them without worries about the complexity of the class hierarchy. We can easily change or remove the names of provided methods either because of specifics of the domain or in case of conflicts.

4.5 Code generation

Model-driven development allows and recommends developers to model systems abstractly and focus on high-level functionality without concern for implementation details. When possible, the goal is to generate implementation code automatically, a process either called model-to-code transformation or code generation. One designs systems with abstract elements in order to focus on business problems instead of technology, to have fewer errors, and to increase speed of development.

In this section, we discuss how systems which are modeled using traits can be implemented using automatic code generation. We will discuss different strategies that depend on the type of target language, and then describe our own automatic code generation used by Umple.

Traits were first introduced and implemented in Squeak [28] and then in other languages such as PHP [54]. These constitute a first group of languages that have native keywords or structures for representing traits; their compilers are aware of traits and can analyze them. A second group of programming languages such as Ruby [55] and Javascript [60] supports traits, but without specific keywords for them. In these cases, developers adapt other structures of the language to implement traits. However, several of the most important languages such as Java and C++ do not support traits at all. There has been some research toward adding traits to these kinds of languages, but traits have never become a part of their standard versions [13,20,35,44,47,48]. When traits are represented in models, there will be three options for implementing the modeled systems corresponding to the three groups of programming languages described above.

The first option is for programming languages that directly support traits. In this case, automatic code generation is straightforward because there will be one-to-one mapping from traits in the modeling level to traits in the implementation level. It is worth pointing out, however, that currently these languages do not support associations and required interfaces as proposed in this paper. Therefore, there are not direct one-to-one mapping for these concepts.

The second option is associated with programming languages which do not have a direct keyword for traits but provide structures used to mimic traits. The implementation for these languages will be a little bit different than the first group because there will not be a direct mapping between traits at the modeling and implementation levels. However, the mapping will happen conceptually because each trait will be implemented with the required structures in the generated language. In this automatic code generation, using best practices will play the most important rule because we would like to use the minimum combination of structures and to have as modular as possible a representation for the implementation of traits. This would improve the process of understanding of the final systems for who are code-oriented or need to inspect the final system.

The final option, for languages not supporting traits at all, is to directly base code generation on the idea that compilers typically use flattening to inject provided methods into clients. After the compiler does this, all elements of traits become treated as real elements of clients, and clients have access to those elements just like any other elements. Code generation from model-based traits can do this directly for programming languages that do not support traits.

This third approach, however, requires greater intelligence in analyzing the traits at the model level to ensure the validity of the final systems. As a result, in Umple, the compiler does considerable analysis of the traits and provides many warnings and error messages when trait syntactic and semantic problems are identified.

It should be noted that the approach we describe in this paper implies that there should not be any *round-trip* model transformation. This means that there should be just a direct transformation from model to code and developers should not modify the generated code. When there is a need for modifications, it should be first applied to the model, and then, the code generation should be reapplied to update the final system. This approach is just like the standard approach for compiling a high-level programming language and is the preferred approach in model-driven development. Allowing round-tripping (taking modifications of generated code and applying them to update the model, then regenerating the code) would be too complex in the context of traits.

We have implemented automatic code generation in Umple. In the generated code, there is a traceable annotation for each provided method which indicates exactly the name of trait from which this method comes. This also specifies the other classes in which the methods have been used. Although there is no need for this in terms of generating a functional system, it helps when people want to inspect or certify the generated code.

Finally, in order to show how trait-based examples indicated in the previous sections can be implemented in object-oriented programming languages according to our approach, we depict the standard class diagrams for them in Figs. 10, 11, 12, 13, 14, 15, 16, 17, 18 and 19 . These class diagrams are generated automatically by Umple.

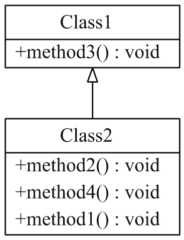

Fig. 12 The standard class diagram for Listing 1 and Fig. 1

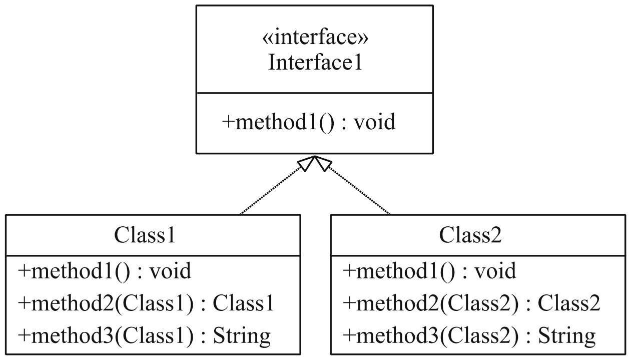
«interface»
Interface1
+method1() : void
Class1
+method1() : void
+method2(Class1) : Class1
+method3(Class1) : String
Class2
+method1() : void
+method2(Class2) : Class2
+method3(Class2) : String

Fig. 13 The standard class diagram for Listing 3 and Fig. 2

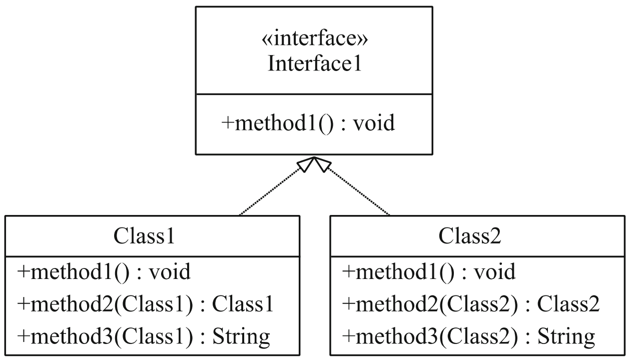
«interface»
Interface1
+method1() : void
Class1
+method1() : void
+method2(Class1) : Class1
+method3(Class1) : String
Class2
+method1() : void
+method2(Class2) : Class2
+method3(Class2) : String

Fig. 14 The standard class diagram for Listing 4 and Fig. 3

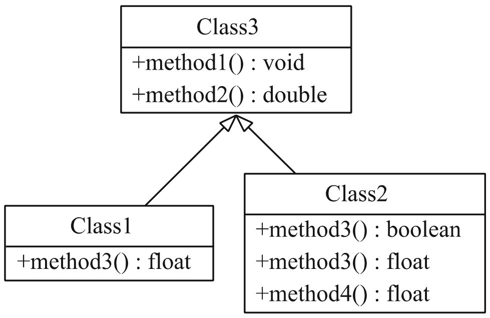

Fig. 15 The standard class diagram for Listing 5 and Fig. 4

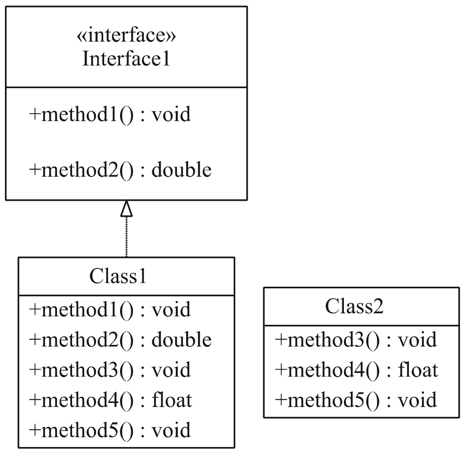

Fig. 16 The standard class diagram for Listing 6 and Fig. 5

5 Evaluation

In this section, we evaluate our proposal related to having traits in the model level and transforming them into target languages while keeping the system's behavior exactly the same as the system without modeling. The advantages of traits associated with better composition and reuse have been recognized by languages such as Scala [50], Squeak [28], Perl [42], Fortress [1], and PHP [54]. Identification of traits is still challenging in this area, and there are manual and semiautomatic approaches and tools for this purpose [11,34].

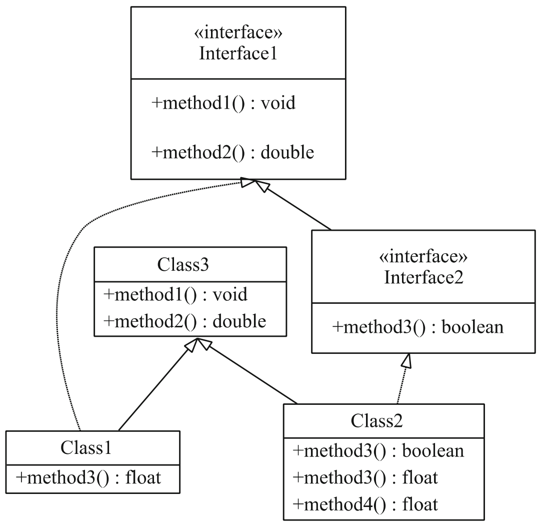

Fig. 17 The standard class diagram for Listing 7 and Fig. 6

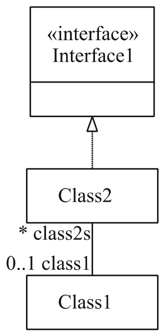

Fig. 18 The standard class diagram for Listing 9 and Fig. 8

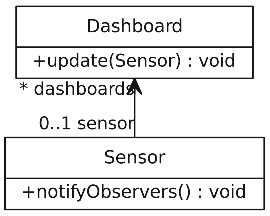

Fig. 19 The standard class diagram for Listing 10 and Fig. 9

However, there is not yet a comprehensive approach or tool that can identify traits based on all clues such as method cancelation [2] or duplication. They were also developed for specific languages, which makes it difficult to have a systematic approach.

According to the goal of our approach, detecting all possible traits, which can have more effect on reusability and LoC, is not necessary. Therefore, we just focus on detecting traits based on the exact duplicate methods. We also give static metrics and describe traits' advantages regarding case studies in our evaluation. We want to indicate that the static metrics are not the key contribution to the approach or the main factors of the evaluation.

We have applied our approach to two systems: the Umple compiler and JHotDraw [29]. The Umple compiler was written in Umple, and we use traits directly in it. However, JHotDraw was written in Java and we had to first transform it to Umple. We did this through a tool called the Umplificator which allows us to transform software systems from Java to Umple [26]. The transformed system retains the same properties as the original since the Umple compiler transforms it back to its original target language.

Detection and implementation of traits for our two case studies were a manual process. Duplicate code (clones and near clones) were detected using CodePro Analytix [19]. CodePro Analytix is a Java tool for Eclipse developers who are concerned about improving software quality and reducing development cost. It provides a feature to find clones, but was not initially designed to detect traits.

In the first round of the process, we detected methods sharing the same signature and body. Each method was assigned to a trait, and then, its required methods were discovered. We followed this approach because we wanted to have fine-grained traits and then compose them into composite traits. When the owner of a method had implemented special interfaces that were crucial for the methods, we considered them as required interfaces.

Next, we looked for methods which had (a) the same number and order of parameters but different types, and (b) the same body. We again assigned each method to a trait and discovered the required methods and required interfaces. Template parameters were added to traits according to the number of differences in types. Then, names of different types were replaced with template parameters. When special restrictions were detected that are needed for binding the values of template parameters, they were applied to parameters.

Static metrics of these two systems before and after applying traits are depicted in Tables 1 and 2, respectively. As can be seen in Table 2, 17 traits were detected for Umple that resulted in 0.84 % deduction in LOC. In other words, 335 lines of code were saved because of traits. All traits have one required method and have been applied at least to two places on average. There are 68 detected traits for JHotDraw, which resulted in 1.36 % (1062 LOC) reduction in code volume. Each trait has at least two required and provided methods on average. The improvement in JHotDraw is more than Umple because there was more interface implementation in JHotDraw as compared to Umple. This is summarized in the fifth column of Table 1. There are 7.2 % interfaces and 92.7 % classes in Umple while 5.7 % interfaces and 94.2 % classes in JHotDraw.

While we were detecting traits in these systems, we made some interesting observations. Some of these were confirmations of already-made points in other scientific papers, while

Table 1 Static metrics of Umple and JHotDraw

	LOC (.java)	LOC (.ump)	Types	Percentage by Kinds	Methods
Umple	100,613	39,782	1133	7.2 % interface, 92.7 % class	2018
JHotDraw	80,535	77,647	1068	5.7 % interface, 94.2 % class	6893

Table 2 Traits specification for Umple and Jhotdraw

	Traits	Required methods (avg)	Clients (avg)	Saved LOC	Saved LOC
Umple	17	1.0	2.3	335	0.84 %
JHotDraw	68	2.5	2.1	1062	1.36 %

the remainder is related to our improvements to traits. Firstly, it was confirmed that code generation is an essential part of model-driven development. The benefits of traits would not be present if we were not generating code and were just using them for documentation, as is often the case for models in industry [43].

Secondly, having big methods clearly decreases reusability. We detected lots of the same code in many methods, but they were mixed up with other implementation code, preventing us from being able to straightforwardly make those methods into provided methods of traits. If the methods had been more fine-grained, we would have likely been able to create more reusable traits and further reduce the number of lines of code. This issue was particularly noticeable in cases where an interface was implemented by several classes. Thirdly, further refactoring of big methods is likely to lead to additional traits, hence further code reduction. Fourthly, as indicated before, we used duplicate methods as a clue for traits, if we used other factors, we would be able to detect more traits. Development of new systems, where traits can be used right from the start, might lead to even greater use of traits in such systems. We leave the validation of these additional hypotheses to future research.

Having relatively small 1 % reduction in code volume through the use of traits in the two reengineered systems described above may seem to suggest the use of traits might not be worthwhile. Having code savings is nice, but not necessary for there to be a valid contribution. The benefit of having the traits goes beyond mere code volume reduction; with the introduction of traits, we have reduced the risk of errors due to duplicate code and have potentially improved the understandability of the system. For example, in our evaluation systems, we found in one duplication case that there was a comprehensive comment regarding the functionality for just one of them and nothing for others. Therefore, developers, who will read the clone instance that does not have the comment, would have to put lots of cognitive effort to understand it.

Finally, our evaluation confirms that traits can be applied in model- driven software development. In fact, we were able to model those open-source systems which are not generally modeled with traits, and provided flexible and reusable elements (traits) that can provide fine and coarse granularity of reusability. Moreover, by our model transformation, we were able to show that if traits are used at the modeling level, there is no problem regarding their implementation in programming languages such as Java.

6 Related work

The term ‘trait’ was first used by Ungar et al. [58] in dynamically typed prototype-based languages, implemented in Self [32]. Traits were then introduced into dynamically typed class-based languages by Schärli et al. [52] as a group of pure methods that serves as a building block for classes and as a primitive unit of code reuse. That research specified the preliminary definitions, interpretation rules, structures, and available conflicts that could happen among traits and classes. Defined traits could only have required and provided methods. These are called stateless traits because the traits do not directly specify attributes and must access data through methods, also known as ‘glue’ code. They implemented traits in Squeak [28], an un-typed language and an open-source dialect of Smalltalk-80. In order to have a realistic evaluation of traits’ usability, they used traits to refactor the Smalltalk-80 collection hierarchy as it is implemented in Squeak 3.2. A graphical representation has been proposed as well to provide visual representation and better understanding.

The formal definition of traits and their basic properties were defined in [21]; the authors also modeled the internal dependencies created by self and supersends so that one can specify precisely when two classes are equivalent.

Stateful traits [9,10] were introduced to avoid the issue of incompleteness in stateless traits. Incompleteness causes classes to have a significant amount of boilerplate glue code when they use traits. This challenge is resolved by letting traits define instance variables (attributes). Instance variables are purely local to the scope of a trait, unless they are explicitly made accessible.

Our research takes this work to a further level, adding modeling concepts, and reducing the amount of glue code even further. We do support attributes (instance variables with associated get and set methods) but with certain rules, as follows. Firstly, code generated from disjoint traits and classes cannot give rise to conflicts due to the design of our compiler. Secondly, if name conflicts would arise in code generated from traits, our system detects them and shows warnings to developers, who will normally elect to change the names of the traits. Thirdly, if Umple generates code for target languages directly support stateful traits, there will not be any issues regarding name conflicts in our approaches.

Having typed trait inheritance is explored in [35,36] in which an extension called Featherweight-trait Java (FTJ) has been developed for Featherweight Java (FJ) [27]. The main goal of the work was to introduce typed trait-based inheritance as a simple way to provide a simple type system that type-checks traits when imported in classes. This could be considered as a first step to adding statically typed trait inheritance to the full Java language.

Supporting traits in Java also was explored in [44,46]. In that research, the goal was to explore how it is possible to resolve barriers of reuse in Java through traits in terms of a case study of Java Swing. These barriers counted as lack of multiple inheritance, inaccessible private inner classes, non-extensible final classes, and synchronized variations. In addition, traits were directly implemented as an extension to mini-Java (MJ), a subset of the Java language, by devising a compiler (TMJC) that works by translation, taking source extended with traits, and translating it to pure mini-Java. An environment based on Eclipse for this implementation of traits in Java was done in [45]. In this, a programmer can move freely between views of the system with or without its traits. The advantage of representing traits as classes is that existing Java development tools can be used.

Murphy-Hill et al. [40] proposed an implementation for Java based on their study of java.io libraries. In their research, traits are represented as stateless Java classes. Required methods are expressed as abstract methods. Classes representing traits can be used with other classes to produce composite classes. According to their implementation, a class can be used both through inheritance and through composition. This implementation is in contrast with their earlier proposal of type traits [46] in which traits were special program units.

Attempts in the direction of exploring traits in Java resulted in the research of Denier [20] in which AspectJ has been utilized as a subset of aspect-oriented programming [3,30]. This mechanism could implement most of properties of traits, but it was not able to provide a full support regarding conflict resolution. The main reason for shortcoming was lack of fine-grained operators in AspectJ.

XTRAITx, a language for pure trait-based programming, was introduced in [13]. The research achieves complete compatibility and interoperability with Java without reducing flexibility of traits. It also provides an incremental adaptation of traits in existing Java projects based on Eclipse. In this implementation, classes play the role of object generators and types, while traits only play the role of units of code reuse and are not types. In this research, there are several operations for traits (as conflict resolution methods) including method alias, method restrict, method hiding, and method/field rename.

In our work, we support two of these conflict resolution methods and would be valuable as future work to apply the others to our model-based traits. XTRAITx uses model transformation to generate Java code, as we do. However, ours supports Java and several other programming languages.

Application of traits in software product lines (SPL) has been investigated in [12]. Traits are used along with records [16] to model the variability of the state part of products explicitly. In their approach, class-based inheritance is ruled out and classes are built only by composition of traits, interfaces, and records. They introduced Featherweight Record-Trait Java (FRTJ) which ensures type-safety of a SPL by type-checking its artifacts only once and ensuring type-safety of an extension of a (type-safe) SPL by checking only the newly added parts. Our work can be extended to be used in the context of SPLs.

UML components are a modular and reusable element in modeling, which having a common terminology with traits. Components and traits have provided and required interfaces and can have fine and coarse level of control on them. For example, they can have one or several methods as required or provided interfaces/methods. There is no restriction on the granularity of both. They can be substituted by another one of their own types if provided and required interfaces/methods are identical. However, these do not mean that they function like one another.

We consider the following differences between them which make traits unique in the modeling level: (a) components can be instantiated, but traits cannot because they are inherently abstract; (b) a UML component may encompass classes, but this is not allowed for traits; (c) we cannot remove/rename/change visibility of provided interfaces in UML components, while we can do these operations for provided methods of traits; (d) we can compile a component and use its binary version, but we cannot perform this for traits; (e) internals of components (except provided interfaces) are hidden for elements that use them, but for traits those internals are flattened in the elements; (f) there is no concept like port for traits; and (h) it is possible to use some provided methods of traits, but when we want to use components, we should load all provided interfaces.

What the traits in the modeling level are trying to do is about providing better flexibility regarding reusability. The features offered by traits are not available in UML components. Furthermore, they are some features which are unique for components, and they are not available in traits. The reason is that traits are aiming at different purposes.

As can be seen, all of the related work has been either in the direction of adding traits to programming languages or using them in new domains. Our approach can be considered to be at a higher level of abstraction and allows having different target languages through model transformation. Furthermore, our online IDE [57] provides two separate views for developers in which they can see simultaneously traits and their flattened view in terms of a graphical class diagram.

7 Challenges

There are three categories of challenges regarding our research: challenges our work will help developers overcome, challenges we faced when developing traits, and challenges that might be faced by those trying to use our traits ideas in models.

7.1 Developers' challenges our work should help with

The first category of challenges is the technical challenges we are allowing software engineers to overcome if they use our approach. Many developers want to avoid multiple inheritance for a variety of reasons, yet at the same time reduce duplication in their model, but there is a lack of techniques to overcome this if the developer wants to work at the modeling level. Moreover, developers who use traits may want to define semantics-level restrictions on traits and make them as a part of traits' usage. On the other hand, our new feature regarding having associations inside traits opens a new design technique which is portable and flexible. For example, we depicted in Listing 10 how the observable pattern can be redesigned.

7.2 Challenges we faced in this work

The second category covers the technical challenges we faced during our research including design and implementation phases. Our first challenge was to develop a usable syntax for traits. We had to define the necessary keywords inside Umple, and there were two options: (1) using specific keywords for each concept and (2) reusing other keywords available in Umple. Since the main philosophy of Umple is simplicity, we defined the minimum new keywords and reused already-existing keywords. For example, instead of having the keywords "required" and "provided" for required and provided methods, we consider abstract methods as required methods and normal methods as provided methods.

The next challenge we faced was about developing semantics of traits in the modeling context. The new semantics must be compatible with original semantics and be usable and meaningful at the modeling level. Trait semantics had to be extended to admit modeling elements as a part of trait definitions. Exploring the soundness of the approach in numerous scenarios was challenging. We had to explore the elements' positive and negative effects and then see whether or not they are in compliance with the original definition of traits. Most of incompatibilities came from the fact that finally those elements must be flattened to clients and there must be minimum conflict and maximum effectiveness for those elements.

Developing rules that show when traits are correct (and producing appropriate error messages in other cases) was also another challenge. We wanted to check the original rules of traits in addition to the ones we promoted for model-based traits in an automatic way. This cause use to check both flattened and normal models. However, it brought a good mechanism for our implementation in which we can show flattened and normal model of the final system in an easy way. Having these two views ought to have a positive effect on developers regarding understanding of the final system [15].

7.3 Challenges to be faced by adopters

The third category of challenges is those that might be encountered by adopters of our work. We believe that the challenges will be minimal.

Since traits provide a layer of functionality sitting on top of the existing modeling language, they simply allow rational copying of reusable elements in a controlled way. Previous research has shown that this can be done with elements such as methods and attributes. We show how it can just as easily be done with associations.

Regarding learnability of our concepts, developers would define traits in a way very similar to how they define classes, albeit with a few unique details related to traits. Developers will need to learn the rules of applying traits in various class hierarchy contexts and how to resolve conflicts through rename/remove operations. These rules are simple, as indicated earlier in this paper.

Moreover, the syntax has been designed to be straightforward, and all conflicts are detected automatically by Umple's compiler. The compiler helps modelers regarding the conflict resolution through concrete examples. In our laboratory, developers could understand how to use traits in their design after a 30-min presentation. Of course, an empirical study ought to be conducted to assess usability of our technique; this is future work.

8 Discussion regarding reusability

In this section, we discuss how our approach can provide better reusability. The goal is to show how each part of our solution along with other preliminary features of traits plays its role in software reuse. For this purpose, we will make connections between specific parts of our approach and the most common concepts in the majority of reuse techniques: abstraction, selection, specialization, and integration [14,31].

8.1 Abstraction

Abstraction plays a central role in software reuse so if we would like to reuse software artifacts effectively, concise and expressive abstractions are essential [31]. Abstraction reflects the point that concealment of details and focusing on the most important factors facilitate reusability. Indeed, having a high level of abstraction helps us to have more reusable elements. In our approach, we concentrate on traits in the modeling level, which is a higher level than traditional programming. This helps us to put implementation concerns beside and focus more on reusing functionality. The implementation concerns will be resolved through automatic code generation.

We introduce elements to traits that increase both abstraction and reusability. Required interfaces can be used in the design of other parts of systems to have or create proper clients. This helps developers to understand requirements of traits more easily or put restrictions on clients in a modular way. Associations are also more abstract than classic code in which developers have to define instance variables and implement the necessary APIs.

8.2 Selection

Selection deals with locating, understand, comparing, and selecting reusable software artifacts. This is covered by the approach in two levels. Firstly, developers can select proper traits (e.g., from a repository) according to their need and use them inside of classes or traits. They may get more understanding about traits by looking at either traits' required methods and interfaces or the interfaces needed by template parameters. Secondly, developers are also allowed to select from the provided methods of each trait. The power of this is increased by the fact that Umple lets developers customize visibility and names. Renaming provided methods toward having domain-specific vocabularies helps developers to have better understanding of the whole functionality of the system. It is important to mention that customization is also used for conflict resolution.

8.3 Specialization

Specialization focuses on generalized or generic artifacts and specializes them by inheritance, parameters, transformation, constraints, and some other forms of refinement. In our approach, inheritance is available through the composing mechanism. It means that traits can extend the whole behavior of their supertraits (composing traits) and classes can extend their traits under the flattening mechanism. The more effective result of specialization can show itself when the modeling elements of traits (e.g., state machines and associations) are specialized in subtraits. For example, when there is an association in a composing trait and there is the same association in the composed trait, then multiplicities of the composed trait will affect the composing one.

Moreover, template parameters allow developers to define generic traits and specialize generic traits by binding specific types to parameters. They also let one put restrictions on types of parameters by defining required interfaces for parameters. Template parameters are used with associations to provide a better configuration mechanism to associate clients of traits with other classes.

8.4 Integration

Integration considers how reusable elements will be integrated into a software system effectively. In other words, how developers combine a collection of selected and specialized artifacts into a complete system. This is achieved by knowing more about artifacts' interfaces. In our approach, there are two complementary ways that developers can understand traits. The first is required methods and interfaces, which reveal how a trait can be used in what classes or traits. The second one is required interfaces for template parameters, which indicate allowable bindings for the parameters. Furthermore, as a common way of helping developers to know much about reuse artifacts, we can assign comments to either traits or each element of traits.

9 Conclusion and future work

This paper proposed and implemented an enhanced mechanism for reuse based on traits. We extended traits to be more abstract and coherent in order to provide better reusability and integration with model- driven software development. Our contributions can be summarized as follows. Firstly, we extended traits with required interfaces that allow us to have additional structural restrictions on clients of traits as well as to reuse already-existing interfaces. Secondly, associations were added to traits, and these were made adaptable to difference contexts through template parameters. Thirdly, for better management of provided methods, a mechanism was defined which allows developers to change visibility of the methods in addition to renaming them. Finally, the already-existing features of traits along with our contributions to traits were implemented in Umple, allowing modeling with traits.

In addition, the code generation mechanism of Umple was also extended so that traits in the modeling level can be implemented in languages such as Java, PHP, and C++. This in turn allows us to use our approach as a generic extension providing traits in those languages. We applied our proposed approach to two systems the Umple compiler itself and JHotDraw. The results showed reduction in duplication and a better reusable mechanism in the modeling level.

Our work should not be seen as an attempt to extend UML (which is a graphical language). We do show a graphical representation generated from our notation, but this is purely to help in understanding the textual syntax we present.

As future work, we intend to provide a more formal definition for our trait extensions like the one defined for basic traits in [21]. We would also like to create an Umple extension to let developers automatically discover traits in their already-developed systems, to apply our mechanism on further real-world open-source systems, and to design an empirical study regarding learnability. Our initial studies have shown that the approach has potential in domain of design patterns and software product lines; therefore, we would like to work on implementing design patterns with traits and gathering them under a repository of traits for Umple. Furthermore, we plan to continue expanding the example systems on which we have tested our work and include systems written in languages other than Java.

References

1. Allen, E., Chase, D., Hallett, J., et al.: The Fortress language specification. Sun Microsyst. **139**, 140 (2005)
2. Arévalo, G.: Understanding behavioral dependencies in class hierarchies using concept analysis. LMO'03: Langages et Modèles à Objets (Object Oriented Languages and Models), pp. 47–59 (2003)
3. AspectJ. 2014. https://www.eclipse.org/aspectj/
4. Badreddin, O., Forward, A., Lethbridge, T.C.: Model oriented programming: an empirical study of comprehension. In: Proceedings of the 2012 Conference of the Center for Advanced Studies on Collaborative Research. IBM Corp., pp. 73–86 (2012)
5. Badreddin, O., Lethbridge, T.C., Forward, A., Elasaar, M., Aljamaan, H.: Enhanced Code Generation from UML Composite State Machines. Modelsward (2014)
6. Badreddin, O., Forward, A., Lethbridge, T.C.: Exploring a model-oriented and executable syntax for UML attributes. Softw. Eng. Res. Manag. Appl. Stud. Comput. Intell. **496**, 33–53 (2013)
7. Badreddin, O., Forward, A., Lethbridge, T.C.: Improving code generation for associations: enforcing multiplicity constraints and ensuring referential integrity. Softw. Eng. Res. Manag. Appl. Stud. Comput. Intell. **496**, 129–149 (2013)
8. Benedicenti, L., Succi, G., Valerio, A., Vernazza, T.: Monitoring the efficiency of a reuse program. ACM SIGAPP Appl. Comput. Rev. **4**(2), 8–14 (1996)
9. Bergel, A., Ducasse, S., Nierstrasz, O., Wuyts, R.: Stateful traits. Advances in smalltalk. In: Proceedings of 14th International Smalltalk Conference (ISC 2006). LNCS, pp. 66–90 (2007)
10. Bergel, A., Ducasse, S., Nierstrasz, O., Wuyts, R.: Stateful traits and their formalization. Comput. Lang. Syst. Struct. **34**(2–3), 83–108 (2008)
11. Bettini, L., Bono, V., Naddeo, M.: A trait based re-engineering technique for Java hierarchies. In: Proceedings of the 6th International Symposium on Principles and Practice of Programming in Java, pp. 149–158 (2008)
12. Bettini, L., Damiani, F., Schaefer, I.: Implementing software product lines using traits. In: Proceedings of the ACM Symposium on Applied Computing, pp. 2096–2102 (2010)
13. Bettini, L., Damiani, F.: Pure trait-based programming on the Java platform. In: Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform Virtual Machines, Languages, and Tools, ACM Press, pp. 67–78 (2013)
14. Biggerstaff, T., Richter, C.: Reusability framework, assessment, and directions. IEEE Softw. **4**(2), 41–49 (1987)
15. Black, A. P., Scharli, N.: Traits: tools and methodology. In: Proceedings of the 26th International Conference on Software Engineering (2004)
16. Bono, V., Damiani, F., Giachino, E.: Separating type, behavior, and state to achieve very fine-grained reuse. In: Electronic Proceedings of Formal Techniques for Java-like Programs (FTfJP) (2007)
17. Bracha, G., Cook, W.: Mixin-based inheritance. ACM SIGPLAN Not. **25**(10), 303–311 (1990)
18. Chung, W., Harrison, W., Kruskal, V., et al.: Concern manipulation environment (CME). In: Proceedings of the 27th International Conference on Software Engineering, pp. 666–667 (2005)
19. CodePro Analytix. 2014. https://developers.google.com/java-dev-tools/codepro/doc/
20. Denier, S.: Traits programming with AspectJ. RSTI-L'objet **11**(3), 69–86 (2005)
21. Ducasse, S., Nierstrasz, O., Schärli, N., Wuyts, R., Black, A.P.: Traits: a mechanism for fine-grained reuse. ACM Trans. Program. Lang. Syst. **28**(2), 331–388 (2006)
22. Duggan, D., Techaubol, C.-C.: Modular mixin-based inheritance for application frameworks. ACM SIGPLAN Not. **36**(11), 223–240 (2001)
23. Frakes, W.B., Kang, K.: Software reuse research: status and future. IEEE Trans. Softw. Eng. **31**(7), 529–536 (2005)
24. Frogley, T.: An introduction to C++ traits. Overload J. **43** (2001). http://accu.org/index.php/journals/442
25. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Longman Publishing Co., Inc., Redwood City (1995)
26. Garzon, M. Lethbridge, T.C.: Exploring how to develop transformations and tools for automated umplification. In: Proceedings of the19th Working Conference on Reverse Engineering. IEEE, pp. 491–494 (2012)
27. Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight Java: a minimal core calculus for Java and GJ. ACM Trans. Program. Lang. Syst. **23**(3), 396–450 (2001)
28. Ingalls, D., Kaehler, T., Maloney, J., Wallace, S., Kay, A.: Back to the future: the story of Squeak, a practical Smalltalk written in itself. ACM SIGPLAN Not. **32**(10), 318–326 (1997)
29. JHotDraw 7. 2004. http://www.randelshofer.ch/oop/jhotdraw/
30. Kiczales, G., Lamping, J., Mendhekar, A., et al.: Aspect-oriented programming. In: The European Conference on Object-Oriented Programming (ECOOP), LNCS 1241. Springer, pp. 220–242 (1997)
31. Krueger, C.W.: Software reuse. ACM Comput. Surv. (CSUR) **24**(2), 131–183 (1992)
32. Lee, E.H.-S.: Object Storage and Inheritance for SELF, a Prototype-Based Object-Oriented Programming Language. Thesis, Stanford University (1988), 508 pp

33. Lethbridge, T.: Teaching modeling using Umple: principles for the development of an effective tool. In: Proceedings of the 27th IEEE Conference on Software Engineering Education and Training (CSEE&T), pp. 23–28 (2014)
34. Lienhard, A., Ducasse, S., Arévalo, G.: Identifying traits with formal concept analysis. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering—ASE'05. ACM Press, p. 66 (2005)
35. Liquori, L., Spiwack, A.: Featherweight-trait Java: a trait-based extension for FJ. [Research Report] RR-5247. INRIA, INRIA Sophia Antipolis - Méditerranée (2004). https://hal.inria.fr/inria-00070751/PS/RR-5247.pdf
36. Liquori, L., Spiwack, A.: FeatherTrait: a modest extension of featherweight Java. ACM Trans. Program. Lang. Syst. **30**(2), 1–32 (2008)
37. Meyers, S.: The most important design guideline? IEEE Softw. **21**(4), 14–16 (2004)
38. Meyers, S.: Effective C++: 55 specific ways to improve your programs and designs, 3rd edn. Addison-Wesley Professional, Redwood City (2005)
39. Morisio, M., Ezran, M., Tully, C.: Success and failure factors in software reuse. IEEE Trans. Softw. Eng. **28**(4), 340–357 (2002)
40. Murphy-Hill, E.R., Quitslund, P.J., Black, A.P.: Removing duplication from java.io. In: Companion to the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, ACM Press, pp. 282–291 (2005)
41. Nierstrasz, O., Reichhart, S., Sch, N.: Adding Traits to (Statically Typed) Languages. Technical report IAM-05-006, Institut für Informatik und Angewandte Mathematik University of Bern, Switzerland (2005)
42. Perl. https://www.perl.org/
43. Petre, M.: No shit or Oh, shit!: responses to observations on the use of UML in professional practice. Softw. Syst. Model. **13**(4), 1225–1235 (2014)
44. Quitslund, P.J., Black, A.P.: Java with traits — improving opportunities for reuse. In: Proceedings of the 3rd International Workshop on MechAnisms for SPEcialization, Generalization and inHerItance (ECOOP), pp. 45–49 (2004)
45. Quitslund, P.J., Murphy-Hill, E.R., Black, A.P.: Supporting Java traits in eclipse. In: Proceedings of the 2004 OOPSLA Workshop on Eclipse Technology Exchange. ACM Press, pp. 37–41 (2004)
46. Quitslund, P.J.: Java Traits—Improving Opportunities for Reuse. Technical report CSE-04-005, OGI School of Science & Engineering Oregon Health & Science University (2004)
47. Reichhart, S.: Traits in CSharp. Software Composition Group. University of Bern, Switzerland (2005)
48. Reppy, J., Turon, A.: Metaprogramming with traits. In: Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP), pp. 373–398 (2007)
49. Sakkinen, M.: Disciplined inheritance. In: European Conference on Object-Oriented Programming (ECOOP), pp. 39–56 (1989)
50. Scala. http://www.scala-lang.org/
51. Schaerli, N., Ducasse, S., Nierstrasz, O.: Classes = Traits + States + Glue (Beyond mixins and multiple inheritance). In: Proceedings of the International Workshop on Inheritance, pp. 1–6 (2002)
52. Schärli, N., Ducasse, S., Nierstrasz, O., Black, A.P.: Traits: composable units of behaviour. In: ECOOP 2003—European Conference on Object-Oriented Programming, volume 2743 of Lecture Notes in Computer Science, pp. 248–274. Springer, Berlin (2003)
53. Snyder, A.: Encapsulation and inheritance in object-oriented programming languages. ACM SIGPLAN Not. **21**(11), 38–45 (1986)
54. Traits in PHP (2014). http://www.php.net/manual/en/language.oop5.traits.php
55. Traits in Ruby (2014). http://ruby-naseby.blogspot.ca/2008/11/traits-in-ruby.html
56. Umple User Manual. Cruise Group, University of Ottawa (2015). http://www.umple.org
57. UmpleOnline (2015). http://www.try.umple.org
58. Ungar, D., Chambers, C., Chang, B.W., Hölzle, U.: Organizing programs without classes. Lisp Symb. Comput. **4**(3), 223–242 (1991)
59. Unified Modeling Language (UML)—Object Management Group (2014). http://www.omg.org/spec/UML/
60. Van Cutsem, T., Miller, M.S.: Traits.js: robust object composition and high-integrity objects for ecmascript 5. In: Proceedings of the 1st ACM SIGPLAN International Workshop on Programming Language and Systems Technologies for Internet Clients. pp. 1–8. ACM Press (2011)

Vahdat Abdelzad is currently a Ph.D. candidate and research assistant at the University of Ottawa under the supervision of Professor Timothy C. Lethbridge. His major is software engineering and has received his B.Sc. and M.Sc. in software engineering as well. He has led and developed several commercial projects and worked several years in the field of software-in-the-loop simulations. He is currently researching on separation of concerns and modularity in model-driven software development.

Timothy C. Lethbridge Ph.D., P.Eng. has been a faculty member at the University of Ottawa since 1994 and is now Vice-Dean Governance of the Faculty of Engineering. His areas of research include software modeling, usable software engineering tools, and software engineering education. Some of his professional activities include curriculum co-chair (2002–2004) of the IEEE/ACM Software Engineering project, Chair (2008–2014) of the Canadian Computer Science Accreditation Council, and General Chair (2015) of the Models conference.