Extended Traits for Model-Driven Software Development

Vahdat Abdelzad
School of Electrical Engineering and Computer Science
University of Ottawa, Ottawa, Canada
v.abdelzad@uottawa.ca

***Abstract*—Software reuse is an important key in developing software systems in a short time with low cost and fewer errors. Traits were introduced to provide fine-grained reusable elements so as to avoid the issues of various forms of inheritance. In spite of being powerful, traits are not used much in software development, mostly because of neither being available in general purpose programming languages nor at the modeling level. In addition, traits suffer from not having control over their clients, which result in a lack of reusability and incorrect usage respectively. In this paper, we propose applying traits to model driven software development. Traits are extended with modeling elements such as associations, state machines, and constraints for a higher level of abstraction. In addition, template parameters are integrated with associations in order to increase genericity and level of abstraction. Traits will be extended with required interfaces to enable structural control over their clients. These features will be implemented in Umple which provides textual modeling of software systems.**

***Index Terms*— Traits, Modeling, UML, Software Development, Umple.**

I. The Problem Statement

It is accepted that software reuse is a key for developing software systems with minimum cost and errors. Inheritance is one of the popular techniques in this direction. However, there are some issues regarding using various forms of inheritance like multiple inheritance and mixins [8,11,22,26]. Traits were introduced to resolve those by providing a fine-grained mechanism that can be applied freely to any level of inheritance hierarchy [24]. A trait, in its original form, is a group of pure methods that serves as a building block for classes. However, most developers are not able to use traits in their software development process. This happens because traits are not available in mainstream programming languages such as C++ and Java, or modeling languages like UML. Therefore, there is a greater amount of duplication than they otherwise might exist in software systems.

Meanwhile, model-driven technologies have started to have an important influence on the development community, although slowly. In particular, state machines and associations, are modeling abstractions that bring new opportunities for reuse, and can be manipulated by inheritance for an even greater degree of reusability. Various issues with inheritance are exposed, however, when these new abstractions become inheritable units; posing challenges, the solution of which are some of my research triggers.

II. Related Work

Nathanael et al. [23,24] introduced the concept of traits in dynamically-typed class-based languages. They are reusable sets of pure methods that serve as elements from which classes can be built. The simplest traits can merely define required and provided methods. These kinds of traits are called stateless traits because they do not directly specify attributes, and all data access must be through methods known as 'glue' code or accessors. The formal definition of traits and their basic properties were defined in [10,25]. Stateful traits [3,4] 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 issue is resolved through allowing instance variables to be defined directly in traits.

Typed trait inheritance is explored in [16,17] in which an extension called Featherweight-trait Java (FTJ) has been developed for Featherweight Java (FJ) [14]. The goal of that project was to introduce typed trait-based inheritance to bring a simple type system that typechecks traits when they are imported in classes.

Traits in Java was explored in [19,21] as well. In that research, the idea was to explore how it is possible to resolve barriers of reuse in Java through traits. IDE support based on Eclipse for this implementation was developed in [20], in which a programmer can move freely between views of the system with or without its traits.

Emerson et al. [18] suggested an implementation for Java according to their study over java.io libraries. In their research, traits are represented as stateless Java classes. Required methods are defined as abstract methods. Classes representing traits can be used with other classes so as to have composite classes. As described by their implementation, a class can be used both through inheritance and through composition. Another attempt in this direction resulted in AspectJ [15,28] being utilized to mimic traits [9]. This mechanism could implement most characteristics of traits, but it was not able to provide a full coverage regarding conflict resolution.

XTRAITx as a language for pure trait-based programming was introduced in [6]. The research achieves complete compatibility and interoperability with Java platform without reducing flexibility of traits. Furthermore, it provides an incremental adaptation of traits in existing Java projects based upon Eclipse. In the implementation, classes get the role of object generators and types while traits only play the role of units of code reuse and are not types.

Application of traits in software product line (SPL) has been investigated in [5]. Traits are used along with records [7] to model the variability of the state part of products explicitly. In their approach, class-based inheritance is ruled out and classes are constituted only by composition of traits, interfaces, and record.

As can be seen, the main thrust of all the above work is either to add traits to specific programming languages or to use them in new domains. There has so far been no attempt to increase genericity of traits or to make them work in a model-driven context. Our research aims to take steps in that direction and resolve a variety of challenges that are uncovered along the way.

III. The Proposed Solution

The extensions proposed by this research cover several dimensions and work together to provide better overall modeling and language flexibility, which can result in better reusability. These are explained in the following sections.

A. Required Interfaces

Traits use the notion of 'required methods' to specify what classes can use them. There is nothing to prevent them from being used in situations in which classes have the same method names with different purposes. Our initial work shows that required methods plus required interfaces can put restrictions on clients of traits and thus avoid traits being used incorrectly. Required interfaces provides a solution ensuring traits will be used correctly with minimum errors.

B. Associations

Associations are key elements in modeling and increase the level of abstraction; generating code from associations can considerably reduce the amount of implementation code that needs writing. Having associations in traits poses a number of challenges that we have addressed in this research. To ensure modularity, specifying association ends must be done through template parameters. Our objective is to enable traits to be used freely to specify different kinds of relationship patterns.

C. State Machines

In order to increase readability of trait functionality (especially at the modeling level) and to provide abstract elements in traits, we are also working on enabling state machines to be included in traits. This will provide a way of reusing state machines at the modeling level. Although this should create a powerful mechanism, we will need to find a straightforward way to manage conflict in trait composition and enable renaming states, removing transitions, and so on.

D. Constraints

Constraints provide a straightforward mechanism that is being used increasingly in modeling and certain programming languages. They can be applied to traits so as to put restrictions on elements (such as provided methods, associations, and so on). Constraints must follow special rules to enable trait composition and so they can be used by classes with conflicting constraints. This proposal requires a deep analysis (like that needed for state machines.

E. Code generation

We are developing our work in the context of Umple [1,2], which has comprehensive code generation from models, with several targeted programming languages. Umple incorporates both code and model; a given program can consist primarily of traditional code, or primarily of abstract model elements. Our traits mechanism will operate as a model transformation operating on both traditional code and model elements, prior to the invocation of code generation from the model elements. As a result traits will become available in programming languages like C++ and Java. In our transformation we focus on programming languages which do not support traits at all. For programming languages which support traits explicitly or implicitly, it is better to have another transformation mechanism which directly maps modeling elements (traits) to specific structure/keywords in the languages. In this way, we can achieve much better traceability between models and the generated code. We should indicate that our transformation mechanism can also be used for those languages if we are not interested in getting benefits of those structures and keywords.

IV. Preliminary Work

We have implemented the some parts of our work in Umple, a textual modeling language that permits embedding of programming concepts into models. It is following syntactic conventions of C-family languages, and adding constructs to such languages. The primary top-level entities are classes, interfaces and traits (which resulted from this research). 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 directives (for generalization), stereotypes, and state machines. Indeed, we have decided to go with Umple because it has comprehensive code generation for several programming languages, provides a textual syntax which brings more expressiveness, supports state machines and constraint along with the code generation for them, and finally has been developed in our own team. Moreover, it should be expressed that our proposed extensions are not just applicable in Umple and they can be used in other modeling languages, for example, UML through stereotypes. To achieve our goal, we first implemented traditional features of traits and their conflict resolution methods. This included required and provided methods, renaming and removing provided methods, and finally trait composition. We also added a new mechanism which allowed changes to visibility of provided methods. Since there is no support for traits in general-purpose programming languages such as Java and C++, we developed a model transformation, which implements traits with basic elements in these languages. In fact, this transformation permits one to use traits at the modeling level without worries about their implementation within these languages.

We implemented required interfaces, associations, and template parameters with their constraints mechanism. The implementation includes their definitions, conflict resolutions, static type checking, and model transformations. Furthermore, we introduced a preliminary version for state machines and constraints in traits. We still need to further investigate composition and conflict resolution mechanisms in the context of traits, associations in traits, and their provided methods. This will become more critical when they are mixed with template parameters.

V. Evaluation

The following two sections Planned and Progress describe our evaluation process.

A. Planned

We have planned two phases to evaluate our research. In the first phase, we are applying our approach to several large open source systems implemented in Java or Umple. The objective are to determine how much improvement will be achieved in terms of lines of code (LOC), how traits behave at the modeling level, and whether or not we can have full functional systems based on traits at the modeling level. This will allow us to determine whether or not there is a reusability issue in current software systems that can be solved by our approach. It also helps us recognize real behavior of traits at the modeling level and prove the application of our proposal.

The second phase of evaluation is to develop a system from scratch, with extensive use of traits. This will allow us to determine the effectiveness of our work in model-driven software development.

Our main output of the evaluation phase is to prove the usability of traits along with our extended features at the modeling level and also to confirm that a system can be completely developed based on traits at the modeling level without worries about implementation challenges. Usefulness of traits has already been proved and we expect to have the same benefits and even more while we are working at the modeling level. Some of criteria which are going to be evaluated are number of reusable elements, granularity of elements, modularity of the system, line of codes, number of classes, traits, and their methods, and understandability of design.

B. Progress

So far we have completed the majority of phase one. We started searching for opportunities to add traits to systems the Umple compiler and JHotDraw [27]. The latter had already been converted to Umple. These systems have 39782 and 77647 Umple LOC respectively. An off-the-shelf tool named CodePro Analytix [29] was used to detect duplicated code. Afterwards, a manual process was utilized to consider converting them to traits.

In the first round of the process, we discovered methods which have the same signature and body. Each method was considered as a trait and then the required methods were recognized. The benefit of doing in this way is to first uncover fine-grained traits and then, when needed, to compose them into composite traits. In order to guarantee to have correct future clients for traits, the interfaces of each class were explored. If they were crucial for the method, we considered them as required interfaces.

Next, we found methods which had a) the same number and order of parameters but different types, and b) the same body. We again applied the same process, which is assigning each method to a trait and discovering the required methods and required interfaces. Based on the differences in types, template parameters were added to the traits. Afterwards, the names of different types were substituted for template parameters. When special restrictions were recognized needed for binding the values of template parameters, they were applied to parameters. The results showed having better reusable elements, reduction in the risk of errors due to duplication, improvement of the understandability of the system, and somehow code volume reduction.

Despite the fact we got an improvement, we have not yet been able to extract state machines and other modeling elements. Therefore, we are planning to apply our approach to two more systems and then we will start developing two systems from scratch.

VI. Potential Applications

The features proposed open new opportunities for traits to be used in different applications. The first is developing libraries based on traits for the most-used functionality. For instance, the functionality related to reading from and writing to files are used in the majority of software systems. These appear as simple methods with routine commands inside. Typically, they are implemented in classes (often as static methods) and used in other classes. There is an issue regarding having those methods in classes which already have superclasses. In this case, we have to import those classes and write wrappers for their methods. This takes effort and creates performance issues because of the overhead of wrappers. By having those methods in traits, we are able to use them directly in classes and even can change their visibilities and give them different names if needed. Potential performance issues will be resolved because the methods will be considered as native methods of the classes. At the current state of our research, we are investigating how we can find and extract such useful reusable traits.

The second opportunity is related to developing a repository for design patterns. Extended traits with template parameters, required interfaces, and associations bring a mechanism by which we can apply patterns directly to candidate classes. The structure of patterns is encapsulated in traits in terms of associations. The dynamics of associations is given by template parameters. The proper classes that can be used as patterns are checked by required interfaces of traits. We plan to conduct research like that conducted when investigating implementing potential design patterns by aspect-oriented programming [13].

The third opportunity is associated with software product lines. Traits are fine-grained elements that can also become more coarse-grained though composition. They can be applied to any level of inheritance hierarchy as well. Traits serve as a mechanism for conflict resolution, which can be used for configuration in software product lines (SPLs.) In other words, we think of having configurable artifacts in terms of traits and applying them freely in an SPL. When needed, we can remove and rename functionality. This potential and new extended features in the modeling level may provide a strong configuration mechanisms for SPLs. We have made a small move in this direction by allowing change the visibilities of provided methods. This is not applicable for conflict resolution but is useful for SPLs. We plan to move in this direction after completing our work on full traits with modeling extensions such as state machines and constraints.

VII. The Expected Contribution

The expectation at the end of this PhD research is to have full traits at the modeling level. We expect to be able to use such modeling elements easily in traits and to generate code for them. We also expect to get positive results for the applications mentioned in Section VI with much more focus on variability and separation of concerns.

The plan for the remaining time of this research is to investigate completely the use of state machines in traits and also implement them in Umple. Afterwards, we integrate constraints into traits and explore how it can affect the design of systems. When we are done with these features and have fully-functional model-based traits, we design and implement a functional system from scratch based on our proposed ideas. We also planned to explore how many designed patterns described in GoF [12] can be implemented by traits.

VIII. Publications

The first paper related to this research has been submitted to the journal Software and System modeling and we passed the first review. It includes details of the work with more examples and the results of the first phase of evaluation.

IX. Acknowledgment

I would like to thank my supervisor Timothy Lethbridge, for the encouragement and advice provided throughout my time. I have been extremely lucky to have a supervisor who cared so much about my work and responded to my questions so promptly.

X. Conclusion and Future Work

In this paper, we explained our research towards extending traits so they can be used in model driven software development. In particular we are working on adding state machines, associations, and constraints in traits. We have already completed the integration of template parameters with associations and also required interfaces in traits in order to have genericity and well-controlled traits respectively. As future research, we want to develop a library of reusable patterns using traits as a basis. We also want to apply an extended version of our work to facilitate work in software product lines and explore variability based on traits.

References

[1] Badreddin, O., Forward, A., and Lethbridge, T.C.Model oriented programming: an empirical study of comprehension. *Proceedings of the 2012 Conference of the Center for Advanced Studies on Collaborative Research*, IBM Corp. (2012), 73–86.

[2] Badreddin, O., Forward, A., and Lethbridge, T.C.Exploring a Model-Oriented and Executable Syntax for UML Attributes. *Software Engineering Research, Management and Applications, Studies in Computational Intelligence 496*, (2013), 33–53.

[3] Bergel, A., Ducasse, S., Nierstrasz, O., and Wuyts, R.Stateful traits. *Advances in Smalltalk, Proceedings of 14th International Smalltalk Conference (ISC 2006), LNCS*, (2007), 66–90.

[4] Bergel, A., Ducasse, S., Nierstrasz, O., and Wuyts, R.Stateful traits and their formalization. *Computer Languages, Systems & Structures 34*, 2-3 (2008), 83–108.

[5] Bettini, L., Damiani, F., and Schaefer, I.Implementing software product lines using traits. *the ACM Symposium on Applied Computing*, (2010), 2096–2102.

[6] Bettini, L. and Damiani, F.Pure trait-based programming on the Java platform. *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform Virtual Machines, Languages, and Tools*, ACM Press (2013), 67–78.

[7] Bono, V., Damiani, F., and Giachino, E.Separating type, behavior, and state to achieve very fine-grained reuse. *Electronic proceedings of Formal Techniques for Java-like Programs (FTfJP)*, (2007).

[8] Bracha, G. and Cook, W.Mixin-based inheritance. *ACM SIGPLAN Notices 25*, 10 (1990), 303–311.

[9] Denier, S.Traits Programming with AspectJ. *RSTI-L'objet 11*, 3 (2005), 69–86.

[10] Ducasse, S., Nierstrasz, O., Schärli, N., Wuyts, R., and Black, A.P.Traits: A Mechanism for Fine-grained Reuse. *ACM Transactions on Programming Languages and Systems 28*, 2 (2006), 331–388.

[11] Duggan, D. and Techaubol, C.-C.Modular mixin-based inheritance for application frameworks. *ACM SIGPLAN Notices 36*, 11 (2001), 223–240.

[12] Gamma, E., Helm, R., Johnson, R., and Vlissides, J.*Design patterns: elements of reusable object-oriented software.* Addison-Wesley Longman Publishing Co., Inc., 1995.

[13] Hannemann, J. and Kiczales, G.Design pattern implementation in Java and aspectJ. *ACM SIGPLAN Notices; the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications; 37*, 11 (2002), 161–173.

[14] Igarashi, A., Pierce, B.C., and Wadler, P.Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems 23*, 3 (2001), 396–450.

[15] Kiczales, G., Lamping, J., Mendhekar, A., et al.Aspect-oriented programming. *the European Conference on Object-Oriented Programming (ECOOP), Springer-Verlag LNCS 1241*, (1997), 220–242.

[16] Liquori, L. and Spiwack, A.Featherweight-trait Java: A trait-based extension for FJ. (2004), 27.

[17] Liquori, L. and Spiwack, A.FeatherTrait: A Modest Extension of Featherweight Java. *ACM Transactions on Programming Languages and Systems 30*, 2 (2008), 1–32.

[18] Murphy-Hill, E.R., Quitslund, P.J., and Black, A.P.Removing duplication from java.io. *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, ACM Press (2005), 282–291.

[19] Quitslund, P.J. and Black, A.P.Java with Traits — Improving Opportunities for Reuse. *Proceedings of the 3rd International Workshop on MechAnisms for SPEcialization, Generalization and inHerItance (ECOOP)*, (2004), 45–49.

[20] Quitslund, P.J., Murphy-Hill, E.R., and Black, A.P.Supporting Java traits in Eclipse. *Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange*, ACM Press (2004), 37–41.

[21] Quitslund, P.J.*Java Traits — Improving Opportunities for Reuse.* Technical Report CSE-04-005, OGI School of Science & Engineering Oregon Health & Science University, 2004.

[22] Sakkinen, M.Disciplined inheritance. *European Conference on Object-Oriented Programming (ECOOP)*, (1989), 39–56.

[23] Schärli, N., Ducasse, S., Nierstrasz, O., and Black, A.*Traits : Composable Units of Behavior.* Beaverton, USA; Bern, Switzerland, 2002.

[24] Schärli, N., Ducasse, S., Nierstrasz, O., and Black, A.P.Traits: Composable Units of Behaviour. *ECOOP 2003 – European Conference on Object-Oriented Programming, volume 2743 of Lecture Notes in Computer Science*, Springer Berlin Heidelberg (2003), 248–274.

[25] Schärli, N., Nierstrasz, O., Ducasse, S., Roel, W., and Black, A.*Traits : The Formal Model.* Technical Report CSE-02-013,CSETech, Software Composition Group, University of Bern, Switzerland, 2003.

[26] Snyder, A.Encapsulation and inheritance in object-oriented programming languages. *ACM SIGPLAN Notices 21*, 11 (1986), 38–45.

[27] JHotDraw 7. *2004*. http://www.randelshofer.ch/oop/jhotdraw/.

[28] AspectJ. 2014. https://www.eclipse.org/aspectj/.

[29] CodePro Analytix. 2014. https://developers.google.com/java-dev-tools/codepro/doc/.