Part 2 : Scientific Information

Main applicant:	Nierstrasz, Oscar
Project title:	A Unified Approach to Composition and Extensibility

Contents

1 Summary **15**

2 Research plan **16**

2.1 State of Research in the Field . . . 16
2.1.1 Object-Oriented Extensibility . . . 16
2.1.2 Inheritance and Mixins . . . 16
2.1.3 Modules and Extensibility in OOP . . . 17
2.1.4 Refactoring and Testing . . . 17
2.2 Research Fields . . . 21
2.3 Detailed Research Plan . . . 24
2.3.1 Traits . . . 24
2.3.2 Classboxes . . . 25
2.3.3 Diamond . . . 25
2.3.4 Composable tests . . . 26
2.4 Timetable . . . 26
2.5 Significance of the Research . . . 27

1 Summary

Real software systems constantly undergo change. For this reason, systems must be *extensible*, so that new features can be added without breaking existing functionality, and they must be *composable*, so that features can be recombined to reflect changing demands on their architecture and design. Object-oriented programming languages excel at expressing arbitrary kinds of models of domain concepts and software systems: the mechanism of *inheritance* is particular useful for specifying incremental extensions to models. However, models built in this way quickly become complex and fragile when they grow to a certain size. The goal of this project is to investigate means to support composability and extensibility in object-oriented languages, while reducing fragility.

Specifically, this project aims to address the question:

> *What is a minimal set of mechanisms needed to enable and control extensibility of software systems?*

We propose to tackle this question by exploring mechanisms for both fine-grained composability and coarse-grained extensibility. We focus mainly on *static* rather than run-time mechanisms. The project entails formal specification and modeling of new mechanisms, their implementation in the context of existing programming languages, and case studies to empirically validate their effectiveness. The project consists of the following four tracks:

Traits — Traits are *fine-grained components* that can be used to construct classes in object-oriented languages in a way that avoids many of the fragility problems inherent in multiple inheritance and mixin-based approaches. We plan to experiment with *encapsulation policies* to enable finer control over Trait composition. We propose to assess and validate Traits by using them to refactor and bootstrap Squeak, the language and environment in which our Traits implementation is developed.

Classboxes — Classboxes are *coarse-grained components* that enable developers to locally extend software in a uniform way without impacting other users of that software. As such, Classboxes support *unanticipated* changes. We propose to assess the effectiveness of Classboxes by applying them to non-trivial application domains. We plan to develop a formal model of Classboxes to enable reasoning about Classboxes and to investigate alternative implementation and evaluation strategies. Finally, we plan to integrate Traits and Classboxes, and explore more expressive composition mechanisms for Classboxes.

Diamond — we plan to develop a new, experimental object-oriented language with *first-class namespaces*. Diamond will offer (i) a *unified* formal foundation for reasoning about extensibility, and (ii) serve as a testbed for exploring new mechanisms to support it. In particular, we intend to use Diamond to develop a satisfactory formal account of the semantics of Traits and Classboxes that will enable reasoning about their integration. Diamond will also help us to explore static type systems that can accommodate Traits and Classboxes.

Composable Method Tests — automated regression tests are an important prerequisite for enabling change in complex systems. Unfortunately certain kinds of changes to a system can invalidate the design of the associated tests. Ideally tests should *co-evolve* with the system. We plan to explore fine-grained, *composable method tests* that will allow tests to be refactored into more reusable parts, and will make it easier to maintain the correspondence between tests and classes as the system is extended. We plan to explore ways of associating composable method tests to Traits so that test suites can be automatically generated when Traits are composed.

2 Research plan

2.1 State of Research in the Field

Here we review the state-of-the-art in *object-oriented extensibility*, *inheritance and mixins*, *modules and extensibility in object-oriented programming*, and finally *refactoring and testing*,

2.1.1 Object-Oriented Extensibility

Lehman and Belady's *Laws of Software Evolution* [LB85] establish that as systems evolve, they become more complex, and consequently more resources are needed to preserve and simplify their structure. They also establish that *successful* systems (*i.e.*, used in a real-world environment) *must change*, or become progressively less useful in that environment.

Various well-known techniques exist to make systems more flexible in the face of change. Many *design patterns*, in particular all of those in the original Design Patterns book [GHJV95] are intended to increase flexibility, however at the cost of increased complexity. Software architectures [SG96] establish rules that govern how the system grows and evolves. Unfortunately, certain kinds of unanticipated change can break the assumptions of an architectural style (for example, pipeline architectures intended for batch processing can be hard to migrate to a fully interactive setting).

Object-oriented frameworks [FS97] support extensibility by encoding a reusable architecture as a set of cooperating classes. Framework users specialize the framework to a specific application by subclassing and extending framework classes. Frameworks can be difficult to use and understand because of implicit dependencies that subclasses should respect. *Reuse contracts* [SLMD96] offer a means to explicitly state the dependencies amongst framework classes that should be respected by application classes.

Component-based software development (CBSD) [Szy98] adopts software architectures in which certain *stable* entities are factored out as components. System extensibility is attained through the configuration of software components, *not* by modification of components themselves. A CBSD approach assumes that the domain is well enough understood that components can be identified and developed with stable interfaces and composition rules.

Aspect-oriented programming (AOP) [KLM+97] is one of a family of approaches (including *subject-oriented programming* [HO93] and *multi-dimensional separation of concerns* [TOHS99]) that acknowledge that certain system concerns *cross-cut* multiple components. These *aspects* can be represented as first class entities that are independently developed, then "weaved" with the domain components to produce complete systems [ML98]. Various experimental aspect-oriented programming languages have been developed, the best-known of which is AspectJ, an extension of Java [KHH+01] in which so-called *joint points* determine the locations at which program entities may be extended and allow one to add methods to existing classes.

Other approaches complementing our proposal support the notion of *unanticipated changes*. For example, *binary component adaptation* allows components to be adapted and evolved in binary form and during program loading [Hö93, KC98]. Mezini and Osterman in [MO02] propose an approach called *on-demand remodularizations* to support flexible a-posteriori integration of generic components. At another level Lava [Kni99] introduces true delegation (*i.e.*, with rebinding of self-references during delegation) in a strongly-typed language to support unanticipated evolution.

2.1.2 Inheritance and Mixins

Inheritance is the key feature to support extensibility in object-oriented languages. One can use inheritance to *incrementally modify* [WZ88] classes to reuse as much as possible and adapt only what one needs to modify. Early on, however, significant problems with inheritance were uncovered [Sny86] [Tai96] that lead to fragile class hierarchies in the face of change. Ducournau [DH87] [DHHM92] has extensively studied different strategies for implementing multiple inheritance and resolving conflicts. Nevertheless, modern mainstream languages like Java and C# avoid these problems by adopting single inheritance instead.

Mixins [Moo86] and *Traits*[1] [CBLL82] were early attempts to construct classes from finer grained components, rather than by incremental modification alone. Bracha's JIGSAW system [Bra92] is still one of the most satisfying accounts of mixins, introducing a set of operators for mixin encapsulation and composition. Although there have been numerous attempts to introduce mixins into object-oriented programming [BC90] [MvL96] [FF98] [ALZ00], to this date, mainly due to scalability problems, mixins have not gained a foothold in mainstream languages, with the possible exception of RUBY [TH01] (a popular object-oriented scripting language). This is largely due to the need to *linearize* mixin composition, a fragile operation in the face of change.

TRAITS, like mixins and multiple inheritance, allow one to build classes from multiple components, which leads us to the question how these components should collaborate. Wolczko has reviewed encapsulation mechanisms in class-based languages and made suggestions to how they could be improved [Wol92]. Composition schemes may lead to encapsulation and name collisions because different components may need to be simultaneously coupled and disjoint, in the sense that they may (i) cooperate by sharing state or by jointly performing operations, but (ii) nevertheless need to hide parts of their implementation from each other [Mez97].

Interest in mixins, TRAITS and module systems has recently been renewed, with a corresponding attempt to formalize them using calculi with reduction systems used to specify both dynamic and static semantics. Smaragdakis and Batory, for example, have identified *mixin layers* – sets of collaborating mixins – as a key concept for programming extensible and adaptable frameworks [SB02]. Ancona, for example, has not only developed such calculi [AZ99] [AZ01] but has also worked on mixin-based extensions to conventional languages like JAVA [ALZ00]. Odersky (EPFL) has introduced TRAITS and mixins in SCALA[2], a multi-paradigm language designed for internet programming tasks.

Fisher and Reppy have developed a type system for TRAITS [FR03] using an *ad hoc* calculus. By contrast, many researchers are converging on *featherweight java* (FJ) [IPW01] as a reference calculus for formally treating object-oriented extensions. Although FJ was designed as a minimal calculus for exploring the addition of generics to JAVA, it is now beginning to serve as a standard for exploring other kinds of extensions, such as new kinds of module systems. Zenger, for example, has developed a calculus of extensible components as an extension to FJ [Zen03]. So far, FJ has not yet been explored for modeling TRAITS.

2.1.3 Modules and Extensibility in OOP

There exists a vast body of literature on module systems. Here we focus purely on some recent developments in the area of extensibility for object-oriented module systems.

In this context, *extensibility* refers to the ability to modify or extend classes and features imported from other modules. MULTIJAVA [CLCM00] brings extensions to JAVA, but does not allow methods to be redefined. MZSCHEME [FF98] offers an expressive module recomposition mechanism based on instantiation of packages through its UNIT system. However a UNIT acts as a black-box and cannot be extended.

GBETA [Ern01], KERIS [Zen02] [Zen03] and OBJECTTEAMS [Her03] each offer some form of extensible modules, but the context in which extensions may be introduced restricts expressiveness. GBETA offers class nesting and virtual class attributes [MMP89]: a nested class C of a class A can only be extended within a subclass of A. An arbitrary class that is not a subclass of A may not extend C. In KERIS, a module may only refine a single other module. Similarly, in OBJECTTEAMS, *teams* are related by single inheritance, so classes belonging to two other teams cannot both be refined.

[1] Note that, although there is some resemblance, the TRAITS mechanism in our proposal should not be confused with *Traits* in MESA referred to here. In the rest of this proposal TRAITS refers to the mechanism we are developing.

[2] http://lamp.epfl.ch/scala/

2.1.4 Refactoring and Testing

Refactoring [Cas92] [Opd92] [FBB+99] is a well-established technique for reorganizing software to improve its structure and its maintainability. On the one hand, a programming language must provide appropriate features that allow code to be refactored in an optimal way, and on the other hand, tools and techniques are needed to ensure that refactoring operations can be safely performed. Since the development of the first practical tool providing automated support for refactoring in SMALLTALK [RBJ97], similar tools have been developed for languages like JAVA.

Unit testing is a well-established practice that facilitates refactoring [BG98]. But unit tests can be difficult to construct, compose and maintain. In particular, it may be difficult to set up scenarios to test domain code.

Mackinnon *et al.* [MFC00] have proposed *mock objects* that simulate concrete test scenarios, as a way to abstract from the testing infrastructure. Mock objects are generally much simpler than server stubs [Bin99], and serve purely to substitute or instrument domain code (like databases or servers) in the context of unit tests. Mock objects have already achieved some success as an industrial testing practice [HT03].

Schuh and Punke propose another approach to composing test scenarios [SP01] [Sch01]. An *ObjectMother* is a factory object that manages the entire lifecycle of creation, customization, and eventually deletion of test objects. ObjectMothers may collaborate to compose test objects for testing complex scenarios.

Unit tests are to be contrasted with assertions posed in Design by Contract [Mey97]. Contracts are abstract, establishing class invariants and pre- and post-conditions for public methods of objects. Although contracts may be evaluated at run-time, they are strictly complementary to unit tests. Madsen [Mad03] has explored the synergy between design by contract and unit testing, and has proposed a testing strategy driven by a partitioning based on analysis of contract assertions. Ernst *et al.* [ECGN99] infer contracts and invariants out of running programs. Cheon *et al.* [CL02], by contrast, create test triggers out of assertions stated in JML. Boyapati *et al.* [BKM02] also create the test data themselves out of preconditions.

Another difficulty is that testing code is highly susceptible to code duplication, due to the presence of boilerplate scenarios. Van Deursen *et al.* [DMBK01] analyze code smells in unit tests, and propose several possible refactorings tailored to unit tests.

Bachmann and Bass have proposed a general technique for managing variability in product-line architectures [BB01] and McGregor has proposed adopting this strategy to compose reusable test cases [McG01].

References

[ALZ00] Davide Ancona, Giovanni Lagorio, and Elena Zucca. Jam — a smooth extension of Java with mixins. In *ECOOP 2000*, number 1850 in Lecture Notes in Computer Science, pages 145–178, 2000.

[AZ99] D. Ancona and E. Zucca. A primitive calculus for module systems. In G. Nadathur, editor, *Principles and Practice of Declarative Programming*, number 1702 in LNCS, pages 62–79. Springer Verlag, 1999.

[AZ01] D. Ancona and E. Zucca. A theory of mixin modules: Algebraic laws and reduction semantics. *Mathematical Structures in Computer Science*, 12(6):701–737, 2001.

[BB01] Felix Bachmann and Len Bass. Managing variability in software architectures. In *ACM SIGSOFT Symposium on Software Reusability*, pages 126–132, 2001.

[BC90] Gilad Bracha and William Cook. Mixin-based inheritance. In *Proceedings OOPSLA/ECOOP '90, ACM SIGPLAN Notices*, volume 25, pages 303–311, October 1990.

[BG98] Kent Beck and Erich Gamma. Test infected: Programmers love writing tests. *Java Report*, 3(7):51–56, 1998.

[Bin99] Robert V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Object Technology Series. Addison Wesley, 1999.

[BKM02] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on java predicates. In *ISSTA'02*, pages 123–133, Roma, Italy, 2002. ACM.

[Bra92] Gilad Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. Ph.D. thesis, Dept. of Computer Science, University of Utah, March 1992.

[Cas92] Eduardo Casais. An incremental class reorganization approach. In O. Lehrmann Madsen, editor, *Proceedings ECOOP '92*, volume 615 of *LNCS*, pages 114–132, Utrecht, the Netherlands, June 1992. Springer-Verlag.

[CBLL82] Gael Curry, Larry Baer, Daniel Lipkie, and Bruce Lee. TRAITS: an approach to multiple inheritance subclassing. In *Proceedings ACM SIGOA, Newsletter*, volume 3, Philadelphia, June 1982.

[CL02] Y. Cheon and G. T. Leavens. A simple and practical approach to unit testing: The jml and junit way. In Boris Magnusson, editor, *Proceedings ECOOP 2002*, volume 2374 of *LNCS*, pages 231–255, Malaga, Spain, June 2002. Springer Verlag.

[CLCM00] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *OOPSLA 2000 Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 130–145, 2000.

[DH87] R. Ducournau and Michel Habib. On some algorithms for multiple inheritance in object-oriented programming. In J. Bézivin, J-M. Hullot, P. Cointe, and H. Lieberman, editors, *Proceedings ECOOP '87*, volume 276 of *LNCS*, pages 243–252, Paris, France, June 1987. Springer-Verlag.

[DHHM92] R. Ducournau, M. Habib, M. Huchard, and M.L. Mugnier. Monotonic conflict resolution mechanisms for inheritance. In *Proceedings OOPSLA '92, ACM SIGPLAN Notices*, volume 27, pages 16–24, October 1992.

[DMBK01] A. van Deursen, L. Moonen, A. van den Bergh, and G. Kok. Refactoring test code. In M. Marchesi, editor, *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes (XP2001)*, pages 92–95. University of Cagliari, 2001.

[ECGN99] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of ICSE '99*, May 1999.

[Ern01] Erik Ernst. Family polymorphism. In J. L. Knudsen, editor, *ECOOP 2001*, LNCS, pages 303–326. Springer Verlag, 2001.

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[FF98] Matthew Flatt and Matthias Felleisen. Units: Cool modules for hot languages. In *Proceedings of PLDI '98 Conference on Programming Language Design and Implementation*, pages 236–248. ACM Press, 1998.

[FR03] Kathleen Fisher and John Reppy. Statically typed traits. Technical Report TR-2003-13, University of Chicago, Department of Computer Science, December 2003.

[FS97] Mohamed E. Fayad and Douglas C. Schmidt. Object-oriented application frameworks (special issue introduction). *Communications of the ACM*, 40(10):39–42, October 1997.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, Mass., 1995.

[Her03] Stephan Herrmann. Object confinement in Object Teams – reconciling encapsulation and flexible integration. In *3rd German Workshop on Aspect-Oriented Software Development*. SIG Object-Oriented Software Development, German Informatics Society, 2003.

[HO93] William Harrison and Harold Ossher. Subject-oriented programming (a critique of pure objects). In *Proceedings OOPSLA '93, ACM SIGPLAN Notices*, volume 28, pages 411–428, October 1993.

[Höl93] Urs Hölzle. Integrating independently-developed components in object-oriented languages. In Oscar Nierstrasz, editor, *Proceedings ECOOP '93*, volume 707 of *LNCS*, pages 36–56, Kaiserslautern, Germany, July 1993. Springer-Verlag.

[HT03] Andy Hunt and Dave Thomas. *Pragmatic Unit Testing in Java with JUnit*. ThePragmaticProgrammers, 2003.

[IPW01] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450, May 2001.

[KC98] Wolfgang Keller and Jens Coldewey. Accessing relational databases: A pattern language. In Robert Martin, Dirk Riehle, and Frank Bushmann, editors, *Pattern Languages of Program Design 3*, pages 313–343. Addison Wesley, 1998.

[KHH+01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In *Proceeding ECOOP 2001*, 2001.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *Proceedings ECOOP '97*, volume 1241 of *LNCS*, pages 220–242, Jyvaskyla, Finland, June 1997. Springer-Verlag.

[Kni99] Günter Kniesel. Type-safe delegation for run-time component adaptation. In R. Guerraoui, editor, *Proceedings ECOOP '99*, volume 1628 of *LNCS*, pages 351–366, Lisbon, Portugal, June 1999. Springer-Verlag.

[LB85] M. M. Lehman and L. Belady. *Program Evolution — Processes of Software Change*. London Academic Press, 1985.

[Mad03] Per Madsen. Unit testing using design by contract and equivalence partitions. In Michele Marchesi and Giancarlo Succi, editors, *Extreme Programming and Agile Processes in Software Engineering*, LNCS, pages 425–426. Springer, 2003.

[McG01] John D. McGregor. Testing a software product line. Technical report, Carnegie Mellon University, 2001.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, second edition, 1997.

[Mez97] Mira Mezini. Dynamic object evolution without name collisions. In *Proceedings ECOOP '97*, pages 190–219. Springer-Verlag, June 1997.

[MFC00] Tim Mackinnon, Steve Freeman, and Philip Craig. Endotesting: Unit testing with mock objects, 2000.

[ML98] Mira Mezini and Karl Lieberherr. Adaptive plug-and-play components for evolutionary software development. In *Proceedings OOPSLA '98 ACM SIGPLAN Notices*, pages 97–116, October 1998.

[MMP89] Ole Lehrmann Madsen and Birger Moller-Pedersen. Virtual classes: A powerful mechanism in object-oriented programming. In *Proceedings OOPSLA '89, ACM SIGPLAN Notices*, volume 24, pages 397–406, October 1989.

[MO02] Mira Mezini and Klaus Ostermann. Integrating independent components with on-demand remodularization. In *Proceedings OOPSLA 2002*, pages 52–67, November 2002.

[Moo86] David A. Moon. Object-oriented programming with flavors. In *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, volume 21, pages 1–8, November 1986.

[MvL96] Tom Mens and Marc van Limberghen. Encapsulation and composition as orthogonal operators on mixins: A solution to multiple inheritance problems. *Object Oriented Systems*, 3(1):1–30, 1996.

[Opd92] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. Ph.D. thesis, University of Illinois, 1992.

[RBJ97] Don Roberts, John Brant, and Ralph E. Johnson. A refactoring tool for Smalltalk. *Theory and Practice of Object Systems (TAPOS)*, 3(4):253–263, 1997.

[SB02] Yannis Smaragdakis and Don Batory. Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. *ACM TOSEM*, 11(2):215–255, April 2002.

[Sch01] Peter Schuh. Recovery, redemption and Extreme Programming. *IEEE Computer*, 18(6):34–41, 2001.

[SG96] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.

[SLMD96] Patrick Steyaert, Carine Lucas, Kim Mens, and Theo D'Hondt. Reuse contracts: Managing the evolution of reusable assets. In *Proceedings of OOPSLA '96 Conference*, pages 268–285. ACM Press, 1996.

[Sny86] Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, volume 21, pages 38–45, November 1986.

[SP01] Peter Schuh and Stephanie Punke. ObjectMother, easing test object creation in XP, 2001. http://www.xpuniverse.com/2001/pdfs/Testing03.pdf.

[Szy98] Clemens A. Szyperski. *Component Software*. Addison Wesley, 1998.

[Tai96] Antero Taivalsaari. On the notion of inheritance. *ACM Computing Surveys*, 28(3):438–479, September 1996.

[TH01] David Thomas and Andrew Hunt. *Programming Ruby*. Addison Wesley, 2001.

[TOHS99] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton, Jr. N Degrees of Separation: Multi-dimensional Separation of Concerns. In *Proceedings of ICSE '99*, pages 107–119, Los Angeles CA, USA, 1999.

[Wol92] Mario Wolczko. Encapsulation, delegation and inheritance in object-oriented languages. *IEEE Software Engineering Journal*, 7(2):95–102, March 1992.

[WZ88] Peter Wegner and Stanley B. Zdonik. Inheritance as an incremental modification mechanism or what like is and isn't like. In S. Gjessing and K. Nygaard, editors, *Proceedings ECOOP '88*, volume 322 of *LNCS*, pages 55–77, Oslo, April 1988. Springer-Verlag.

[Zen02] Matthias Zenger. Evolving software with extensible modules. In *International Workshop on Unanticipated Software Evolution*, Malaga, Spain, June 2002.

[Zen03] Matthias Zenger. *Programming Language Abstractions for Extensible Software Components*. Ph.D. thesis, EPFL, 2003.

2.2 Research Fields

In this section we describe recent work of the Software Composition Group (SCG) pertaining to this proposal. The work on Traits, Classboxes and Piccola has been carried out in the course of our ongoing SNF Project no. 2000-067855.02, *"Tools and Techniques for Decomposing and Composing Software"*.

Most of the research of the SCG is motivated by the conviction that *software evolution* is the key to productivity [Nie02] in software engineering processes, and that consequently we should focus our research efforts on activities that enable and facilitate change in complex software systems. Traits, Classboxes and Piccola each address some aspect of this theme.

Traits [SDNB03] are a new approach to composing classes from reusable parts, that overcomes many of the problems inherent in multiple inheritance and mixins. Traits are essentially sets of collaborating methods that may be composed with a superclass and instance variables to form a class. A Trait not only *provides* methods, but may also *require* methods that should be provided by the superclass or other Traits. Traits may be *composed* to form other Traits. If naming conflicts arise during composition, then the conflicts must be explicitly resolved. Traits respect the *flattening property*, which effectively states that the way in which Traits are composed has no effect on the way in which they are used — Traits can always be "flattened", or compiled away. In this way fragility due to linearization approaches (common with mixins) is avoided.

We have implemented a prototype of Traits in the Squeak environment[3]. Our implementation has been used convincingly to refactor the Smalltalk collection hierarchy [BSD03]. Appropriate tool support [SB03] further alleviates some of the problems that arise when programming with mixins [SDNB03]. Nevertheless, there are still many open questions concerning how to flexibly and conveniently compose classes from Traits. *Encapsulation policies* that offer fine-grained control over the visibility and access rights to features present in Traits are a first step in this

[3] Squeak is a public-domain Smalltalk implementation: http://www.squeak.org/

direction [SDNW04]. We have also made some first steps in formalizing Traits [SND+02] using a simple set-theoretic approach that features first-class namespaces.

The current prototype serves as a proof-of-concept of Traits. The integration with the host language was not a goal of this first version. As a consequence, the design of this prototype is brittle and it cannot run on newer versions of Squeak, drastically limiting the accessibility of Traits for external users. Still the current implementation shows us that Traits can be implemented without loss of performance, while relying only on the default method lookup mechanism.

The rewrite will not only let us give more exposure to Traits in Squeak, but more fundamentally it will serve as a reference implementation for the introduction of Traits in other languages. Questions about the internal design choices (such as how to efficiently represent required methods, and how to recompute the conflicts) will be answered in this reference implementation. This way we will be able to focus on the key aspects of the integration of Traits in statically typed languages, like C#.

Classboxes [BDW03] address the problem that classical module systems do not offer the ability to add or replace a method in a class that is not defined in that module. Classboxes offer a minimal module system for object-oriented languages in which extensions (method addition and replacement) to imported classes are *locally visible*. Essentially, a Classbox defines a scope within which certain entities, *i.e.*, classes, methods and variables, are defined. A Classbox may *import* entities from other Classboxes, and optionally extend them *without impacting the originating* Classbox. Concretely, classes may be imported, and methods may be added or redefined, without affecting clients of that class present in other Classboxes. Local rebinding strictly limits the impact of changes to clients of the extending Classbox, leading to better control over changes, while giving the illusion from a local perspective that changes are global.

We have implemented a proof-of-concept prototype of Classboxes in Squeak. In our implementation, the method lookup mechanism in the Squeak virtual machine has been modified to take Classboxes into account. This first prototype exhibits an overall 60% slowdown in performance. We expect that a less naive approach, in which Classboxes are largely compiled away, will lead to significant performance improvements without impacting space requirements. (Classboxes could be macro-expanded, leading to better performance, but with a resulting explosion in multiple instantiations of Classboxes wherever they are imported.)

The prototype clearly demonstrates the feasibility of the approach. Empirical studies with real applications are still needed to evaluate the current design choices. A formal specification of Classboxes is missing, though some first steps with a simple set-theoretic model have been taken. This formalization is needed (i) to justify alternative implementation strategies, and (ii) to investigate the applicability of Classboxes to other programming languages, particularly statically typed languages like Java.

Piccola [ALSN01] is a minimal language of *first-class namespaces* [AN00] used to model software components and compositional styles. The formal semantics of Piccola is defined with the help of a process calculus [NA03]. The key concepts of the calculus are *forms* (*i.e.*, first-class namespaces), *agents* and *channels*. Forms are used to model or encode not only usual programming language entities like objects, classes, components and modules, but they are also used to express and compose environments in which these entities are defined. Agents model concurrent autonomous entities, and channels serve as a communication and coordination medium between agents. This leads to an expressive programming model that supports a great variety of advanced programming styles (for example, mixin layers [NA00]). We have shown that Piccola and the formal calculus provide a minimal foundation for expressing many ways of separating programming concerns [NA04].

The current Piccola implementation is stable [NAK03] and is available from the SCG web site. Although Piccola is very expressive, it is still very low-level. Although objects, classes, components, Traits and Classboxes can be encoded using namespaces, these encodings can become unwieldy for real programming tasks. A higher-level language based on first-class namespaces would offer a more convenient tool for experimenting with language features to support extensibility.

We have started to investigate how composition can be applied to the testing process to support

software extensibility and system change. A first experiment [GNW03] suggests that applications with good test coverage typically contain test suites that *include* one another, in the sense that the parts of the system they test may overlap. We are investigating the use of *composable method tests* to refactor complex tests. Each composable method test (i) instantiates a test object and a number of parameter objects, (ii) invokes a *specific method* of the test object, with the parameters, (iii) may perform some set of assertions, and (iv) if successful, returns a vector of the test object and parameters. Method tests are composable in the sense that other method tests may be used in step (i) to create the test object and the parameters.

Although not directly pertinent to this proposal, we have also carried out extensive research on *reverse engineering* and *re-engineering* techniques in the context of evolving software systems. The book *Object-Oriented Reengineering Patterns* [DDN02] describes a series of lightweight techniques for understanding, assessing and restructuring complex software systems. We have also carried out various experiments to analyze the way in which systems evolve, and to identify modules in object-oriented applications. We have been able to detect refactorings by analyzing software metrics of different versions of a system [DDN00]. The *evolution matrix* is a compact visual presentation of the growth and decline of the classes in a system [LD02]. We have also shown how to combine the version-based information with history information to improve the detection of design flaws [RDGM04].

The majority of these publications are available as PDF downloads from www.iam.unibe.ch/∼scg/.

References

[ALSN01] Franz Achermann, Markus Lumpe, Jean-Guy Schneider, and Oscar Nierstrasz. Piccola — a small composition language. In Howard Bowman and John Derrick, editors, *Formal Methods for Distributed Processing — A Survey of Object-Oriented Approaches*, pages 403–426. Cambridge University Press, 2001.

[AN00] Franz Achermann and Oscar Nierstrasz. Explicit Namespaces. In Jürg Gutknecht and Wolfgang Weck, editors, *Modular Programming Languages*, volume 1897 of *LNCS*, pages 77–89, Zürich, Switzerland, September 2000. Springer-Verlag.

[BDW03] Alexandre Bergel, Stéphane Ducasse, and Roel Wuyts. Classboxes: A minimal module model supporting local rebinding. In *Proceedings of JMLC 2003 (Joint Modular Languages Conference)*, volume 2789 of *LNCS*, pages 122–131. Springer-Verlag, 2003.

[BSD03] Andrew P. Black, Nathanael Schärli, and Stéphane Ducasse. Applying traits to the Smalltalk collection hierarchy. In *Proceedings OOPSLA 2003 (International Conference on Object-Oriented Programming Systems, Languages and Applications)*, pages 47–64, October 2003.

[DDN00] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. Finding refactorings via change metrics. In *Proceedings of OOPSLA '2000 (International Conference on Object-Oriented Programming Systems, Languages and Applications)*, pages 166–178, 2000.

[DDN02] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.

[GNW03] Markus Gälli, Oscar Nierstrasz, and Roel Wuyts. Partial ordering tests by coverage sets. Technical Report IAM-03-013, Institut für Informatik, Universität Bern, Switzerland, September 2003.

[LD02] Michele Lanza and Stéphane Ducasse. Understanding software evolution using a combination of software visualization and software metrics. In *Proceedings of LMO 2002 (Langages et Modèles à Objets*, pages 135–149, 2002.

[NA00] Oscar Nierstrasz and Franz Achermann. Supporting Compositional Styles for Software Evolution. In *Proceedings IEEE International Symposium on Principles of Software Evolution (ISPSE 2000)*, pages 11–19, Kanazawa, Japan, November 2000.

[NA03] Oscar Nierstrasz and Franz Achermann. A calculus for modeling software components. In *FMCO 2002 Proceedings*, volume 2852 of *LNCS*, pages 339–360. Springer-Verlag, 2003.

[NA04] Oscar Nierstrasz and Franz Achermann. Separating concerns with first-class namespaces. In Tzilla Elrad, Siobán Clarke, Mehmet Aksit, and Robert Filman, editors, *Aspect-Oriented Software Development*. Addison-Wesley, 2004. To appear.

[NAK03] Oscar Nierstrasz, Franz Achermann, and Stefan Kneubühl. A guide to JPiccola. Technical Report IAM-03-003, Institut für Informatik, Universität Bern, Switzerland, June 2003.

[Nie02] Oscar Nierstrasz. Software evolution as the key to productivity. In *Proceedings Radical Innovations of Software and Systems Engineering in the Future*, Venice, Italy, October 2002. preprint.

[RDGM04] Daniel Ratiu, Stéphane Ducasse, Tudor Gîrba, and Radu Marinescu. Using history information to improve design flaws detection. In *Proceedings of the Conference on Software Maintenance and Reengineering (CSMR 2004)*, 2004.

[SB03] Nathanael Schärli and Andrew P. Black. A browser for incremental programming. *Computer Languages, Systems and Structures*, 2003. (To appear, special issue on Smalltalk).

[SDNB03] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew Black. Traits: Composable units of behavior. In *Proceedings ECOOP 2003*, number 2743 in LNCS, pages 248–274. Springer Verlag, July 2003.

[SDNW04] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Roel Wuyts. Composable encapsulation policies. In *Proceedings ECOOP 2004*, LNCS. Springer Verlag, June 2004. To appear.

[SND+02] Nathanael Schärli, Oscar Nierstrasz, Stéphane Ducasse, Roel Wuyts, and Andrew Black. Traits: The formal model. Technical Report IAM-02-006, Institut für Informatik, Universität Bern, Switzerland, November 2002. Also available as Technical Report CSE-02-013, OGI School of Science & Engineering, Beaverton, Oregon, USA.

2.3 Detailed Research Plan

In this project, we will investigate various means to improve extensibility of object-oriented systems. Specifically we will (i) improve the *fine-grained* mechanism of Traits, (ii) develop the *coarse-grained* mechanism of Classboxes, and (iii) develop the *fine-grained* mechanism of composable method tests. Finally, we will develop (iv) Diamond, a formal language for studying Traits, Classboxes and other mechanisms within a unified framework of first-class namespaces.

We expect that this research will yield insights into:

- how to apply Traits and Classboxes effectively to real software systems,
- how to integrate Traits and Classboxes efficiently in mainstream object-oriented languages,
- how to develop an effective unit testing strategy that takes extensibility into account.

2.3.1 Traits

Traits are well-suited for building classes from smaller components as long as the involved Traits do not use the same method names for incompatible purposes. We plan to (i) develop mechanisms to offer the programmer more fine-grained control over the way that Traits are bound to tackle this limitation, and (ii) validate Traits by using them to bootstrap a second generation implementation.

Concretely, we are aiming for a mechanism to encapsulate Traits at composition time to avoid unintended name clashes. In a first step towards this goal, we have developed a general notion of *composable encapsulation policies* [SDNW04], which express access rights of features to different classes of clients. The choice of encapsulation policy is deferred to composition time, resulting in a very flexible and expressive approach to feature composition. In the context of Traits, the next step is to choose an appropriate semantics for the encapsulation operations that are expressed by such encapsulation policies. In particular, we are investigating what is the most appropriate semantics for early-binding (and hiding) methods that are defined in a Trait.

In parallel we plan to bootstrap Squeak, the language in which Traits are currently implemented, *i.e.*, we plan to rewrite the core of the language using Traits themselves. This bootstrap is a non-trivial application of Traits. The reflective class-metaclass core of Smalltalk has to

be changed to integrate Traits as in this context any class is potentially composed of a number of Traits. This will lead to a general restructuring of the Smalltalk meta-architecture. By refactoring the core of the language as a composition of Traits we expect to arrive at a new and better factored meta-object protocol. Besides the core metaclass architecture restructuring we want to bootstrap other language infrastructure aspects such as memory management, processes and the process scheduler, and basic types to obtain a language kernel that fully benefits from the Traits capabilities.

2.3.2 Classboxes

Classboxes are well-suited for realizing isolated extensions to portions of complex applications, but practical experience with the mechanism is lacking. We plan to (i) develop a formal specification of Classboxes, (ii) carry out empirical case studies to assess their effectiveness in real applications, (iii) investigate extensions and alternative implementation strategies.

Currently Classboxes are both specified and implemented operationally in terms of a modified method lookup algorithm that takes the scoping rules of Classbox imports into account. We seek a more satisfactory formal model of Classboxes that explains local rebinding in a way that separates the semantics of method lookup from the scoping rules of Classboxes by means of first-class namespaces. Such a formalization will help us explore alternative implementation strategies that do not require a modified method lookup and thereby exhibit better performance.

We plan to evaluate the expressiveness and usefulness of Classboxes with a non-trivial case study based on re-structuring a large web server: Seaside [4] is a framework for developing sophisticated web applications written in Squeak. It is composed of more than 150 classes and nearly 1200 method definitions. Seaside runs on top of a web server. Comanche [5] is a full-featured web server environment. It is mainly composed of three pieces: KomHttpServer, KomServices and KomPackaging. These rely on various extensions of Squeak related to network facilities. We plan to refactor Seaside and Comanche using Classboxes to isolate the various components and their extensions. We expect the case study to highlight not only the advantages but also weaknesses of the current Classbox mechanism in relation to the current Smalltalk packaging and scoping mechanisms.

Currently Classboxes function purely as a packaging and scoping mechanism. We intend to investigate various extensions of Classboxes. We expect that an integration with Traits will be most fruitful, as this will enable packaging of collaborating Traits (and their associated tests). Presently Classboxes lack any notion of a *component model*. We expect that explicit interfaces and composition mechanisms for Classboxes will increase their usefulness. In particular, we intend to investigate the application of encapsulation policies to Classboxes.

2.3.3 Diamond

The objective of this activity is to develop a *formal testbed* for modeling and reasoning about language mechanisms to support extensibility. Diamond is intended to be a small object-oriented language with a precisely defined formal semantics. We intend to use featherweight java (FJ) as a starting point for defining Diamond, that is, we will define the language as a small calculus whose operational semantics is defined by means of reduction rules. Such a reduction system lends itself well to interpretation in a first implementation. Diamond will extend FJ with concepts that are essential for modeling extensibility and software composition.

Diamond will build on some of the ideas of Piccola. In particular, *first-class namespaces* will play a key role for modeling various kinds of extensible and composable modules. However, the focus in Diamond will be on reasoning about extensibility in mainstream object-oriented languages, not on developing a minimal language for modeling software composition.

We expect that Diamond will help us to provide a *unified* foundation for (i) formally specifying Traits and Classboxes, (ii) integrating Traits and Classboxes, (iii) reasoning about

[4] http://www.beta4.com/seaside2/
[5] http://squeaklab.org/comanche/

implementation strategies, (iv) providing insights into applying Traits and Classboxes in other mainstream object-oriented languages, particularly statically typed ones, and (v) formally specifying and experimenting with new mechanisms to support composition and extension.

Since Diamond is intended to be an executable specification language, we will be able to use it in the context of a test harness to verify that the Traits and Classbox implementations conform to their formal specification for a set of test runs. (This strategy has been successfully applied for our set-theoretic accounts of Traits and Classboxes.)

2.3.4 Composable tests

We plan to investigate various means to exploit unit testing in the context of extensibility. Specifically, we plan to (i) explore composable method tests as a means to refactor unit tests, (ii) associate *abstract* tests to Traits to generate unit tests during Trait composition, (iii) investigate tools and mechanisms composing tests and analysing effective test coverage.

We have already performed some small case studies in refactoring tests into composable method tests, and plan to carry out more extensive case studies. We specifically seek to investigate the reduction in the size of the test code base, reduction in execution time, and focus on most specific tests during debugging. Positive results in any one of these three tracks will improve the effectiveness and usefulness of unit testing in extensible systems.

We expect that Traits will be more useful and attractive if they carry their tests with them. Since Traits are abstract (*i.e.*, in the same sense that abstract classes are), one cannot construct concrete unit tests for them. However, one may associate abstract tests to Traits that would become concrete when the Traits are composed to construct classes. We believe that the combination of Traits, Classboxes and composable method tests will be highly promising: a Classbox could encapsulate one or more Traits with their associated abstract tests. When Traits are composed, their abstract tests can be simultaneously composed, yielding both concrete classes and concrete unit tests. We plan to carry out a series of experiments to assess the feasibility and practicality of this idea.

Currently, although it is straightforward to know which parts of a system are *exercised* by a test suite, it is not trivial to know which methods and classes are actually *tested* by some assertions. Composable method tests yield a more refined notion of test coverage that exposes which actual methods of the system are tested. We envisage a series of tools and techniques that exploit this correspondence. Coverage statistics, debugging guidance, automatic test composition, and even automatic generation of skeleton method tests suggest themselves. We plan to explore these possibilities and evaluate which of these are effective in enabling change in complex systems.

2.4 Timetable

We expect to obtain the following results over the two years of the project:

Year 1

- Complete implementation of Traits in Squeak
  Extend Traits with encapsulation policies
- Apply Classboxes to Seaside and Comanche (case study)
  Elaborate the Classbox formal model
- Specify and implement first version of Diamond that extends FJ with first-class namespaces
  Develop an executable specification of Traits and Classboxes in Diamond
- Refactor Squeak unit tests as composable method tests (case study)
  Apply composable method tests to Traits

Year 2

- Use Traits to bootstrap Traits (case study)
- Extend Classboxes with encapsulation policies
- Elaborate a static type system for Diamond
  Use Diamond to elaborate Traits and Classboxes for statically typed languages
- Explore test composition using Traits and Classboxes
  Develop tools to analyze and compose method tests

2.5 Significance of the Research

We have already established a number of important contacts in relation to the further development of Traits. It appears highly likely that the next generation of Squeak will include Traits. This proposal will answer some open questions concerning performance and usability of Traits.

Considerable interest is also emerging into the applicability of Traits for statically typed languages like Java and C#.

The OBASCO (Objects, Aspects and Components) team at INRIA led by Prof. Pierre Cointe has started to implement Traits in Java and use Traits in the context of Aspect-Oriented Programming. We are now evaluating how to collaborate with them.

We have submitted a proposal to Microsft for a small pilot project to evaluate the feasibility of integrating Traits in C# in the context of the Microsoft Shared Source Common Language Infrastructure, also known as *Rotor*[6].

We are collaborating with Prof. Andrew Black at OGI School of Science and Engineering in Oregon on the further development of Traits. We also also in close contact with Prof. Martin Odersky at EPFL, who has incorporated Traits into his Scala language.

The work on Classboxes is much newer and consequently less advanced. However, we note that there is currently renewed and intense interest in module systems for object-oriented languages. We expect that the results emerging from our research on Traits, Classboxes, composable method tests and diamond will have an influence on the design of practical programming languages.

Although unit testing is a well-established practice, few language researchers have shown much interest in the topic until recently. We believe that the importance of a better integration of testing with programming is just starting to be recognized, and that key research results will be achieved in the coming years. Practical, new ideas that facilitate testing of complex systems promise to have a major impact on industrial software development practices.

As usual, we plan to publish our results in top-ranked, peer-reviewed international fora, such as OOPSLA, ECOOP and TOPLAS.

[6] http://research.microsoft.com/Collaboration/University/Europe/RFP/Rotor2/default.aspx