Extensibility in the Large

Matthias Zenger
Programming Methods Laboratory
Swiss Federal Institute of Technology Lausanne
matthias.zenger@epfl.ch

Programming Software Components

Software component technology is driven by the promise of building software from off-the-shelf components that are provided by a global software component industry consisting of independent component developers [14]. Therefore, component programming emphasizes independent development and deployment of software modules. For being independently deployable, a component has to be well separated from other components. In addition, it has to be composable with other components by a third-party that does not necessarily have access to the implementation details of all the components involved.

An important point which is often not considered is *extensibility*. Components have to be extensible, since in general, components do not fit off-the-shelf into an arbitrary deployment context. They first have to be adapted to satisfy the needs of a particular customer. Apart from this, extensibility is also an important requirement for enabling software evolution. Software evolution includes the maintenance and extension of component features and interfaces. Supporting software evolution is important, since components are architectural building blocks and as such, subject to continuous change. A typical software evolution process yields different versions of a single component being deployed in different contexts. Extensibility is also required when developing *families of software applications* [11, 2]. For instance, software product-lines [8, 16] rely heavily on a mechanism for creating variants of a system which share a common structure but which are configured with possibly different components.

Here is a list of requirements we identified to be important for component-oriented programming practice including the corresponding implications on the implementation platform:

- It is necessary that components are implemented in a modular way with explicit context dependencies, enabling type-safe separate compilation.
- Mechanisms for composing independently developed components have to be flexible but also safe.
- Component composition has to scale well, since component-oriented programming is targeted towards *programming in the large* [4].
- Reuse of components in different contexts should imply the least possible need for explicit adaptation code.
- In support for a smooth software evolution process, components have to be extensible without the need to anticipate possible extensions.
- Component systems have to be extensible on the system level as well, allowing to plug in alternative or additional components without the need for re-wiring the whole system.
- Extensibility has to be non-invasive, allowing to derive different extensions of a component independently.
- Different extensions of a component have to be able to coexist requiring an appropriate versioning mechanism.
- Component deployment and extension must not require the availability of source code since this would violate the principle of binary component deployment.

In general, these issues pose high demands on the implementation platform. It is clear by now that mainstream object-oriented programming languages, which are today predominantly used for programming software components, do not live up to most of the requirements. Recently, this observation gave rise to research how to support component technology best on the programming language level.

Language Support for Software Extensibility

Previously published approaches for supporting component-oriented programming on the language level such as *ComponentJ* [12], *ACOEL* [13], *ArchJava* [1], etc. each target specific issues (like type-safety), but none of them addresses extensibility in particular.

In order to investigate extensibility issues related to component-oriented programming, we developed a component model that emphasizes the evolution of components [18]. It is a simple prototype-based model for first-class components built on top of a class-based object-oriented language. The model is formalized as an extension of *Featherweight Java* [7]. This calculus includes a small set of primitives to dynamically create, compose, and extend software components in a type-safe way, while supporting features like explicit context dependencies, late composition, unanticipated component extensibility, and strong encapsulation. Opposed to most other approaches which link services of components explicitly, components get composed implicitly with coarse-grained composition operators. We used this framework to discuss the trade-offs between aggregation-based and mixin-based component compositions. To make some of the ideas developed in this work useable in practice, we are currently working on a module system with explicit support for extensibility [17].

Even though classical module systems like the one of Modula-3 [3], Oberon-2 [10] and Ada 95 [15] can be used to model the modular aspects of software components well, they have severe restrictions concerning extensibility and reuse. These systems allow type-safe separate compilation, but they hard-wire module dependencies; i.e. they refer to other modules by name, which makes it impossible to plug in a module with a different name but a compatible specification without performing consistent renamings on the source code level. For functional programming languages, module systems [9, 5] exist that obey the principle of external connections, i.e. the separation of component definition and component connections. These module systems maximize reuse, but they yield modules that are not extensible, since everything is hard-wired internally. We consider this lacking support for unanticipated extensibility to be a serious shortcoming. In practice one is required to use ad-hoc techniques to introduce changes in modules. In most cases this comes down to hack the changes into the source code of the corresponding modules. This obviously contradicts the idea of deploying compiled module binaries — a process which does not require to publish source code. But even for cases where the source code is available, source code modifications are considered to be error-prone. With modifications on the source code level one risks to invalidate the use of modules in contexts they get already successfully deployed.

The design of our module system includes primitives for creating and linking modules as well as type-safe mechanisms for extending modules or even fully linked programs statically [17]. Module composition is based on aggregation, opposed to other approaches for extensible modules that make use of a mixin-based scheme. The extensibility mechanism relies on two concepts: module *refinements* and module *specializations*. Both of them are based on inheritance on the module level. While refinements yield a new version of a module that subsumes the original module, specializations are used to derive new independent modules from a given "prototype". The module system supports software development according to the *open/closed* principle: Programs are *closed* in the sense that they can be executed, but they are *open* for extensions that statically add, refine or replace modules or whole subsystems of interconnected modules. Extensibility does not have to be planned ahead and does not require modifications of existing source code, promoting a smooth software evolution process.

The overall design of the module system was guided by the aim to develop a pragmatic, implementable, and conservative extension of *Java* [6]. We are currently implementing a compiler based on the extensible Java compiler *JaCo* [19, 20]. *JaCo* itself is designed to support unanticipated extensions without the need for source code modifications. *JaCo* is currently written in a slightly extended *Java* dialect making use of an architectural design pattern that allows refinements in a similar way. We hope to be able to re-implement *JaCo* in future using extensible modules. This would also allow us to gain experience with extensible modules and their capabilities to statically evolve software through module refinements and specializations.

References

[1] J. Aldrich, C. Chambers, and D. Notkin. Architectural reasoning in ArchJava. In *Proceedings of the 16th European Conference on Object-Oriented Programming*, Málaga, Spain, June 2002.

[2] J. Bosch and A. Ran. Evolution of software product families. In *3rd International Workshop on*

Software Architectures for Product Families, LNCS 1951, pages 168–183, Las Palmas de Gran Canaria, Spain, 2000.

[3] L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, and G. Nelson. Modula-3 language definition. *ACM SIGPLAN Notices*, 27(8):15–42, August 1992.

[4] F. Deremer and H. H. Kron. Programming in the large versus programming in the small. *IEEE Transactions on Software Engineering*, June 1976.

[5] M. Flatt and M. Felleisen. Units: Cool modules for HOT languages. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 236–248, 1998.

[6] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Java Series, Sun Microsystems, second edition, 2000. ISBN 0-201-31008-2.

[7] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages & Applications*, volume 34(10), pages 132–146, 1999.

[8] M. Jazayeri, A. Ran, and F. van der Linden. *Software Architecture for Product Families: Principles and Practices*. Addison-Wesley, 2000.

[9] D. MacQueen. Modules for Standard ML. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 198–207, New York, August 1984.

[10] H. Mössenböck and N. Wirth. The programming language Oberon-2. *Structured Programming*, 12(4):179–195, 1991.

[11] D. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, SE-2(1):1–9, 1976.

[12] J. C. Seco and L. Caires. A basic model of typed components. In *Proceedings of the 14th European Conference on Object-Oriented Programming*, pages 108–128, 2000.

[13] V. C. Sreedhar. Programming software components using ACOEL. Unpublished manuscript, IBM T.J. Watson Research Center, 2002.

[14] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison Wesley / ACM Press, New York, 1998. ISBN 0-201-17888-5.

[15] S. T. Taft and R. A. Duff. *Ada 95 Reference Manual: Language and Standard Libraries*. Lecture Notes in Computer Science. Springer Verlag, 1997. ISBN 3-540-63144-5.

[16] D. Weiss and C. Lai. *Software Product-Line Engineering*. Addison-Wesley, 1999.

[17] M. Zenger. Evolving software with extensible modules. In *International Workshop on Unanticipated Software Evolution*, Málaga, Spain, 2002.

[18] M. Zenger. Type-safe prototype-based component evolution. In *Proceedings of the European Conference on Object-Oriented Programming*, Málaga, Spain, June 2002.

[19] M. Zenger and M. Odersky. Extensible algebraic datatypes with defaults. In *Proceedings of the International Conference on Functional Programming*, Firenze, Italy, September 2001.

[20] M. Zenger and M. Odersky. Implementing extensible compilers. In *ECOOP Workshop on Multiparadigm Programming with Object-Oriented Languages*, Budapest, Hungary, June 2001.