Evolving Software with Extensible Modules

Matthias Zenger
École Polytechnique Fédérale de Lausanne
INR Ecublens, 1015 Lausanne, Switzerland
matthias.zenger@epfl.ch

Abstract

We present the programming language Keris, an extension of Java with explicit support for software evolution. Keris introduces extensible modules as the basic building blocks for software. Modules are composed hierarchically revealing explicitly the architecture of systems. A distinct feature of the module design is that modules do not get linked manually. Instead, the wiring of modules gets infered. The module assembly and refinement mechanism of Keris is not restricted to the unanticipated extensibility of atomic modules. It also allows to extend fully linked systems by replacing selected submodules with compatible versions without needing to re-link the full system. Extensibility is type-safe and non-invasive; i.e. the extension of a module preserves the original version and does not require access to source code.

1. Introduction

This paper presents *Keris*, a pragmatic, backward-compatible extension of the programming language *Java* [24] with explicit support for modular, component-oriented programming [51, 52]. Many modern programming languages provide mechanisms for modular program development. They allow to define modules that depend on functionality imported from other modules. Furthermore there is often support for *separate compilation*, allowing modules to be compiled in isolation. Separate compilation and the ability to abstract over external functionality make it possible to flexibly deploy modules in different contexts with different cooperating modules.

Opposed to this typically well supported form of reuse, most mainstream programming languages do not address the ability to extend modules without planning extensibility ahead. Since modules, as architectural building blocks, are subject to continuous change, 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 the programming language *Keris* includes primitives for creating and linking modules as well as mechanisms for extending modules or even fully linked programs statically. Programs written in *Keris* 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.

In this paper we introduce *Keris* as an extension of the programming language *Java*. In Section 2 we substantiate the need for linguistic abstractions in object-oriented programming languages for *programming in the large*. In Section 3 we present the design of the programming language *Keris* by a stepwise introduction of the new constructs for assembling and evolving modules. Our prototypical implementation of the *Keris* compiler gets reviewed in Section 5. In Section 6 we discuss related work. Section 7 concludes.

2. Motivation

2.1 Modular Programming

Like most popular object-oriented languages, *Java* does not provide suitable abstractions for programming in the large. *Java*'s package system is too weak to be useful as an abstraction for reusable software components in general. It is not even well suited for modeling larger libraries. Such libraries often require means for internal structuring. It is possible to nest packages, but this also limits access to non-public members. Therefore all classes that need to access library internal data (which does not get exposed to the outside world) have to reside in the same package.

Java's package mechanism was designed mainly for structuring the name space and for grouping classes. A package does not even allow to fully encapsulate a set of classes since the *Java* programming language does not offer a way to close packages.[1][2] Thus, like in most popular object-oriented languages, classes are basically the only means to structure software systems.

Classes itself do not allow modular programming either [50, 12]. In general, classes cannot be compiled separately; mutually dependent classes have to be compiled simultaneously.

[1] In *Java* class loaders can be used at runtime to ensure that only a fixed set of classes is loaded from a package. The concept of *sealed packages* exploits this mechanism to restrict class loading for classes of such a package only to a particular *Jar* file.

[2] Regarding the open nature of packages it is surprising to see that adding classes to a *Java* package is not type-safe. This can break programs that import all classes of a package via the star-import command.

Since classes do not define context dependencies explicitly, it gets even difficult to find out on what other classes a class depends. Basically, this can only be found out by inspecting code.

Even though classes are the basic building blocks for object-oriented programming, most classes do not mean anything in isolation. They have a role in a specific program structure, but there is only limited support to formulate this role or to make this role explicit. A priori, class interactions are implicit, if not using a special design pattern that emphasizes cooperating classes. Since a single class often does not mean anything in isolation, formulating design patterns, software components, the architecture of a system, and even expressing the notion of a library on the level of the programming language turns out to be extremely difficult in general.

A good example for this problem is the way how industrial component models represent software components in class-based object-oriented languages. In these models, the implementation of a software component is typically guided by a relatively weekly specified programming protocol (e.g. *JavaBeans* [30]). The composition of software components is mostly even performed outside of the programming language, using meta-programming technology. Thus, often neither the process of manufacturing a component nor the component composition mechanism are type-safe.

2.2 Extensibility

Java supports the development of extensible software only on a very low level by means of class inheritance and subtype polymorphism. Extensibility has to be planned ahead through the use of design patterns typically derived from the *Abstract-Factory* pattern [21]. With *Java*'s late binding mechanism developing open software that can be extended with *plug-ins* is relatively easy. But again, this has to be planned ahead and allows only to extend an application in a restricted framework. For writing applications that are open for unanticipated extensions, often complicated programming protocols have to be strictly observed (e.g. *Context/Component* [59]).

3. Extensible Modules for *Java*

The design of the programming language *Keris* was driven by the observation that extensibility on the module level can help to develop highly extensible applications [27]. *Keris* tries to facilitate the development of extensible software by providing an additional layer for structuring software components. This layer introduces modules as the basic building blocks for software. With *Keris*' modules, it is possible to give concrete implementations for concepts like design patterns, libraries, applications or subsystems. All this is done in a completely extensible fashion, allowing to refine existing software or to derive new extended software from existing pieces. To keep software extensible, *Keris* promotes software without hard links which are frequently found in *Java* programs in form of class instantiations or accesses to static methods or fields. Of course, being a conservative extension of *Java*, it is possible to introduce hard links whenever required.

3.1 Defining Modules

In *Keris*, modules are the basic top-level building blocks supporting separate compilation as well as function and type abstraction in an extensible fashion. *Keris*' modules specify context dependencies explicitly. They can only be deployed in contexts that meet these requirements.

To introduce *Keris*' module mechanism, we now present a small example that defines a module SORTER which provides functions for reading a list of words, for sorting, and for printing out lists.[3]

```
module SORTER requires INOUT {
    String[] read() {
        ... INOUT.read() ...
    }
    void write(String[] list ) {
        ... INOUT.write(list[i ]) ...
    }
    void sort(String[] list ) { ... }
}
```

The header of the module declaration states that module SORTER depends on functionality provided by another module INOUT. Within the body of a module it is possible to access the members of the own module as well as all the members of modules that are declared to be required. Members of modules are generally accessed by qualifying member names with the corresponding module. This distinguishes *requirements* from *imports* of traditional module systems that typically make members of other modules accessible so that they can be used in an unqualified form.

It remains to show a specification of module INOUT. We do this by defining a module interface that defines the signature of this module. Such a module interface does not contain any code, it only specifies the types of members provided by a concrete implementation of this module.

```
module interface INOUT {
    String read();
    void write(String str);
}
```

We will now define a module CONSOLE that implements this interface and thus is a possible candidate for being used together with module SORTER.

```
module CONSOLE implements INOUT {
    String read () { ... System.in.read() ... }
    void write(String str) { System.out.println(str); }
}
```

This module implements the functions read and write by forwarding the calls to appropriate methods of the standard *Java* API for text in- and output on a terminal. Here is an alternative implementation for INOUT based on functionality provided by a third module LOG.

```
module LOGIO implements INOUT requires LOG {
    String read () { ... System.in.read() ... }
    void write(String str ) { ... LOG.log(...) ... }
}
```

Note that module SORTER does not explicitly implement a module interface. This is not strictly necessary since every module declaration implicitly defines a module interface of the same name. Nevertheless, the separation of module implementations from interfaces is an important mechanism that is essential to enable separate compilation.[4]

[3] Note that we write module names in capital letters.

[4] Some module systems, e.g. *Oberon*'s module system, provide means

3.2 Linking Modules

Before discussing the module composition mechanism, we have to stress the distinction between modules and module instances. A module can be seen as a "template" for multiple module instances of the same structure and type. We have to differentiate between the two, since we want to be able to deploy a module more than once within an application. For instance, we could have two different instances of the SORTER module that are linked together with different INOUT module instances.

In *Keris*, modules are composed by aggregation. More concretely, a module does not only define functions and variables. It may also define module instances as its members. These nested module instances, we also call them *submodules*,[5] can depend on other instances of the same context. The following definition for module APP links module SORTER with module CONSOLE by declaring both to be submodules of the enclosing module APP.

```
module APP {
   module SORTER;
   module CONSOLE;
   void main(String[] args) {
      String [] list = SORTER.read();
      SORTER.sort(list);
      SORTER.write(list);
   }
}
```

Submodule definitions start with the keyword module followed by the name of the module implementation. The enclosing module aggregates for every submodule definition an instance of the specified module. Thus, module APP aggregates two module instances SORTER and CONSOLE. A submodule can only be defined if its deployment context, given by the enclosing module, satisfies all the requirements of the submodule. The requirements of a submodule are satisfied only if all modules required from the submodule are either provided as other submodules, or are explicitly required from the enclosing module.

The program above defines two submodules SORTER and CONSOLE. Module SORTER requires a module instance INOUT from the deployment context, CONSOLE does not have any context dependencies. The module definition of APP is well-formed since it defines a CONSOLE submodule that implements INOUT, and therefore provides the module that is required by the SORTER submodule. Note that module CONSOLE got only introduced in module APP for that reason. Module APP does not refer to members of CONSOLE directly.

Modules without any context dependencies like APP can be executed if they define a main method. For executing a module, an instance gets created and the main method is called. The main method of the previous code shows that submodules get accessed simply via the module name.

Similarly to the previous code, we could try to link module SORTER with module LOGIO.

```
module BUGGYAPP {
   module SORTER;
   module LOGIO;
}
```

A verification of the context dependencies reveals that this module declaration is not well-formed. LOGIO requires a module instance LOG which does not get declared within BUGGYAPP. Since we want BUGGYAPP to be parametric in the cooperating module LOG, we have to abstract over the LOG instance by requiring it from the context. This has the effect that inside of the module body we are able to refer to a module instance LOG without actually giving a concrete definition. Therefore the following code is well-formed.

```
module LOGSORTER requires LOG {
   module SORTER;
   module LOGIO;
}
```

The previous examples show that modules get composed by hierarchically aggregating submodules. A module that hosts a set of submodules is only well-formed if it satisfies the context requirements of all of its submodules. A module satisfies the requirements of a submodule if modules required from that submodule are either present in form of other submodules, or are explicitly required by the host module.

This hierarchical composition mechanism has the advantage that the static architecture of a system gets explicit. Furthermore, module composition does not require to link modules explicitly by specifying how context dependencies are satisfied at deployment time. Instead, the module interconnection gets *infered*. With this approach we avoid linking modules by hand which can be a tedious task that raises scalability issues [57]. On the other hand, our inference technique only succeeds if we avoid ambiguities; i.e. our type system has to ensure that references to module instances are unique in every context. One implication of this is that a module can never define two nested module instances (submodules) that implement the same module. Furthermore, the set of required and nested modules has to be disjunct. This seems to be a rather big restriction, but Section 3.4 will show how to use *module specializations* to overcome some of the limitations. Details about the type system are out of this paper's scope and are therefore left out.

3.3 Refining Modules

We now come to the problem of extending a module. Since we do not want to break code that makes use of existing modules, we are not allowed to touch existing modules. Extensibility has to be *additive* instead of being *invasive*.

Keris has support for non-invasive extensions through a *module refinement* mechanism. It allows to refine an existing module by providing new functionality or by overriding existing functionality. The refined module is compatible to the original module in the sense that it can be substituted for it. Thus, *Keris* lifts the notion of compatibility between classes expressed by a subtyping relation to the more coarse-grained level of modules.

Here is a refinement of module SORTER that adds a new function filterDuplicates to the already existing set of functions for filtering out duplicate entries in lists. Furthermore it overrides the sort function by providing a version that is, for instance, more efficient than the previous one.

to support separate compilation without separating module interface definitions from module implementations.

[5] We use a terminology here which is not fully consistent with the one on the class level. *Submodules* denote *nested modules* and have nothing to do with *subclassing*. The motivation for naming nested modules *submodules* comes from nested modules modeling *subsystems*.

```
module XSORTER refines SORTER {
   String[] filterDuplicates(String[] list ) { ... }
   void sort(String[] list ) { ... }
}
```

Module XSORTER is a refinement of module SORTER. It inherits the interface and the implementation from SORTER and therefore implements the module interface of SORTER as well. Note that it also inherits the context dependencies; i.e. XSORTER requires a INOUT module.

Similar to this refinement of module implementations and their implicit interfaces, we are also able to refine plain module interfaces like INOUT.

```
module interface XINOUT refines INOUT {
   void write(int i);
   void write(float f);
}
```

Based on the already existing module CONSOLE we can now derive a module that implements this interface.

```
module XCONSOLE refines CONSOLE
                    implements XINOUT requires CONVERT {
   void write(int i ) { ... }
   void write(float f ) { ... }
}
```

So far, we only saw how to refine the functionality of atomic modules. But as desired, these refinements do not affect existing code. So how do we integrate a new module into a system that makes use of the old SORTER module? Since systems are represented by modules, it is probably not surprising to do this again with a refinement. We explained before that *Keris* promotes programming without hard links. Following this idea, we allow to override submodule declarations in module refinements. The following code refines our module APP representing an executable application by covariantly overriding the SORTER submodule.

```
module XAPP refines APP {
   module XSORTER;
}
```

The refined module XAPP replaces the nested module implementation SORTER with one for module XSORTER. Consequently, the inherited main method now refers to the XSORTER submodule. In fact, we can now access the XSORTER submodule via both module names, SORTER and XSORTER. The only difference is that when accessed via XSORTER, we can refer to the new functions. The ability to refine a module interface stepwise to allow different access levels is called *incremental revelation* [14].

This small example demonstrates that our module assembly and refinement mechanism not only supports the extension of atomic modules. It also allows us to extend fully linked programs (represented by modules with aggregated submodules) by simply replacing selected submodules with compatible version. There is no need to establish module interconnections again; we reuse the fully linked program structure and only specify the submodules and functions to replace or add.

This extensibility mechanism features *plug-and-play* programming. It does not touch existing code. After having refined our application with module XAPP we can still run the old application APP. We could even assemble a system that makes use of both modules without having to fear unpredictable interferences.

3.4 Specializing Modules

Refining a module is the process of extending a module by adding new functionality or by modifying existing functionality through overriding. A module refinement yields a new module that subsumes the old one. As a consequence, it is always possible to replace a module with one of its refinements. In the following code, module BUGGYMOD aggregates a submodule that subsumes another submodule. This is illegal, since references to the subsumed submodule SORTER are ambiguous within module BUGGYMOD.

```
module BUGGYMOD requires INOUT {
   module SORTER;
   module XSORTER;
}
```

Section 4 will motivate a case where we need a different form of reuse: We would like to define a new module on top of an old one but we do not want the new module to subsume the old one. We call this process of creating a new distinct module based on an existing module *specialization*. Here is a specialization of the SORTER module:

```
module BACKSORTER specializes SORTER { ... }
```

Module BACKSORTER inherits members from SORTER (including its requirements), but as a specialization it is not considered to subsume it. In particular, it does not inherit SORTER's implemented module interfaces. Otherwise, SORTER and BACKSORTER would not denote different modules. Consequently, it is perfectly legal to define a module with both a BACKSORTER and a SORTER submodule.

```
module SORTING requires INOUT {
   module SORTER;
   module BACKSORTER;
}
```

Often, mutual referential modules have to be specialized at the same time consistently. The ability to refer to a specialized version of a module requires that we are able to specialize context dependencies as well. This "rewiring" is expressed in the following code using the as operator. The FASTSORTER module specializes module SORTER and instead of requiring the original INOUT module, it now refers to a specialized FASTIO module.

```
module FASTSORTER specializes SORTER
                        requires FASTIO as INOUT { ...
}
```

A more complete example for module refinements and rewiring of context dependencies can be found in Section 4.

While module refinements promote the *substitutability* of modules, module specializations support the notion of *conceptual abstraction* on the module level [44]. Conceptual abstraction refers to the ability to factor out code and structure shared by several modules into a common "supermodule" which gets specialized independently into different directions. The specializations represent distinct modules that cannot be substituted for the common "supermodule".

3.5 Virtual Classes

Until now we only considered functional modules. With these modules, static class members would be superfluous. Such members could be implemented as module members with the benefit of extensibility and improved reusability.

Even though functions on the module level can be quite useful to model global behavior, it is probably more common in object-oriented languages to have modules that contain class definitions. Classes defined in a module can freely refer to other members of the module as well as to modules required from the enclosing module. The following module defines a class for representing points.

```
module GEO {
    class Point {
        Point(int x, int y ) { ... }
        int getX () { ... }
        int getY () { ... }
    }
}
```

Module systems for *Java*-like programming languages that allow to abstract over classes are extremely difficult to implement in practice if one wants to stick to *Java*'s compilation model [4]. In such module systems, classes can, for instance, extend classes of required modules (for which only the interface might be given). Consequently, at compile time, a compiler has to translate the class without knowing its concrete superclass. Since *Keris* is designed to support *Java*'s compilation model while being implementable on the standard *Java* platform, we decided not to offer a facility for abstracting over regular classes. Thus, classes on the module level are handled like inner classes [24, 28].

To support reuse and extensibility of types, *Keris* introduces the notion of *virtual class fields* as an alternative type abstraction mechanism. A class field defines a new class by specifying its interface and by possibly giving a concrete implementation, which is typically a reference to a regular class. Here is an example defining an interface, a class, and a virtual class field within one module:

```
module POINT {
    interface IPoint {
        IPoint(int x, int y);
        int getX () { ... }
        int getY () { ... }
    }
    class CPoint implements IPoint {
        CPoint(int x, int y ) { ... }
        int getX () { ... }
        int getY () { ... }
    }
    class Point implements IPoint = CPoint;
    Point root() { return new Point(0, 0); }
    void print(Point p ) { ... p.getX () ... p.getY () ... }
}
```

Module POINT defines an interface IPoint[6] as well as a class CPoint for representing points. Furthermore, it introduces a class field Point by separately specifying its interface and implementation. Methods print and root show that class fields behave like classes: They can be instantiated and members of corresponding objects can be dereferenced.[7] The main difference is that class fields are virtual and therefore can be covariantly overridden in refined modules.

```
module COLORPOINT refines POINT requires COLOR {
    interface IColor {
        COLOR.Color getColor();
    }
    class CColPoint extends CPoint implements IColor {
        ...
    }
    class Point implements IPoint, IColor = CColPoint;
    void print(Point p ) { ... p.getColor () ... super.print(p); }
}
```

Refinement COLORPOINT specifies that class field Point now also supports the IColor interface and is implemented by the CColPoint class. Furthermore, print is overridden to include the color in the output. At this point, one might wonder what happens to method root of the original module POINT which instantiates class field Point. In fact, for the refined module it now returns a colored point since we were overriding class field Point.

The ability to covariantly refine types (or class fields in our case) is essential for extending object-oriented software. Most object-oriented languages support interface and implementation inheritance. But inheritance alone does not support software refinement well. Existing code refers to the former type and cannot be overridden covariantly in a type-safe way to make use of the extended features. For special cases like binary methods, some languages support the notion of *self types* [11, 10, 41]. But these are not suitable for mutually referential classes that have to be refined together to ensure consistency [17]. Here, only virtual types are expressive enough [29, 54, 16, 36]. Unfortunately, virtual types rely in general on dynamic type-checking. Therefore recent work concentrated on restricting the mechanism to achieve static type-safety [55, 9].

Keris' class fields are statically type-safe. This is mainly due to the nature of refinements: A refined module subsumes the former module and cannot coexist with the former module within the same context. It rather replaces the former module consistently in explicitly specified contexts. Module specializations do not endanger type-safety either, since they conceptually create completely new modules with class fields that do not have a (subtype) relationship with the original class fields.

A distinct feature of the class field mechanism, in comparison with virtual types, is the possibility to declare dependencies between virtual class fields. These dependencies define a subtype relationship among virtual class fields and therefore promote the consistent refinement or specialization of class field hierarchies.

4. Design Patterns as Module Aggregates

In this section we briefly describe the usage of modules to develop generic implementations of design patterns in a modular fashion. We pick the *Subject/Observer* pattern as an example [21]. Figure 1 introduces three modules as the building blocks of this pattern. The observer type is defined

[6] Interfaces in *Keris* can also specify the signature of constructors allowing class fields to be instantiated like regular classes.

[7] Class fields cannot be extended via subclassing, but it is possible to define a subtype relationship between class fields as we will briefly explain later.

```
module OBSERVER requires SUBJECT, EVENT {
    interface IObserver {
        IObserver();
        void notify(SUBJECT.Subject subj, EVENT.Event evt);
    }
    class CObserver implements IObserver {
        void notify(SUBJECT.Subject subj, EVENT.Event evt) {
            ...
        }
    }
    class Observer implements IObserver = CObserver;
}
module interface EVENT {
    class Event;
}
```

```
module SUBJECT requires OBSERVER, EVENT {
    interface ISubject {
        ISubject();
        void add(OBSERVER.Observer obs);
        void notify(EVENT.Event evt);
    }
    class Subject implements ISubject = {
        OBSERVER.Observer[] obs;
        void add(OBSERVER.Observer obs) { ... }
        void notify(EVENT.Event evt) {
            for (int i = 0; i < obs.length; i++)
                observers[i].notify(this, evt);
        }
    }
}
```

Figure 1: A modular Subject/Observer implementation

in module OBSERVER by the class field Observer. Module OBSERVER has to require the corresponding SUBJECT module since the observer type refers to the subject. Similarly, module SUBJECT requires module OBSERVER for defining a class field Subject.[8] We have no concrete implementation for events, so the EVENT module gets described by a module interface.

We can now link the mutually dependent modules together yielding a single module SUBJECT_OBSERVER that represents the complete Subject/Observer pattern. In addition to the aggregated modules we also define a function attach. The composed module SUBJECT_OBSERVER is a natural place for defining functions that belong logically to the whole pattern, and not to a specific participant.

```
module SUBJECT_OBSERVER requires EVENT {
    module SUBJECT;
    module OBSERVER;
    void attach(SUBJECT.Subject s, OBSERVER.Observer o) {
        s.add(o); }
}
```

We could create refined versions of that pattern with alternative properties, but here, we are mainly interested in specializing it for a specific application. Following the example in [54], we derive a data structure for modeling a window manager by consistently specializing the mutually referential modules SUBJECT and OBSERVER. We start with the covariant specialization of the SUBJECT module.

```
module MANAGER specializes SUBJECT
                    requires WINDOW as OBSERVER,
                             WINEVENT as EVENT {
    interface IManager { ... }
    class Subject implements ISubject, IManager = ...
}
```

Module MANAGER also has to specialize the requirements of the original SUBJECT module with the as construct. This "rewiring" has the effect that all former references to the OBSERVER module now refer to module WINDOW. The same holds for EVENT. Without this specialization we could not link module MANAGER with the corresponding module WINDOW since WINDOW is distinct from OBSERVER and therefore cannot play its role.

```
module WINDOW specializes OBSERVER
                    requires MANAGER as SUBJECT,
                             WINEVENT as EVENT {
    interface IWindow { ... }
    class Subject implements ISubject, IWindow = ...
}
module WINEVENT specializes EVENT { ... }
```

Finally, we compose the modules to represent the window manager pattern as a specialization of the Subject/Observer pattern. Here we have to specialize the submodules accordingly. We cannot simply override the original SUBJECT and OBSERVER submodules, since our specialized modules do not subsume them.

```
module WIN_SYSTEM  specializes SUBJECT_OBSERVER
                           requires WINEVENT as EVENT {
    module MANAGER as SUBJECT;
    module WINDOW as OBSERVER;
}
```

5. Implementation

We implemented a compiler prototype for *Keris*. The compiler reads *Keris* source code and produces standard *Java* classfiles for classes as well as modules. Since *Keris* is designed to be a conservative extension of *Java* that fully interoperates with regular *Java* classes, the *Keris* compiler can also be used as a drop-in replacement for *javac*.

The compiler is implemented as an extension of the extensible *Java* compiler *JaCo* [58, 59]. JaCo itself is designed to support unanticipated extensions without the need for source code modifications. Since JaCo is written in a slightly extended *Java* dialect using an architectural design pattern that allows refinements in a similar way like *Keris*, we hope to be able to implement JaCo in future in the programming language *Keris* itself. Furthermore, with this project we hope to gain experience with the language and its capabilities to statically evolve software through module refinements and specializations.

[8] In Figure 1 we use an *anonymous class* declaration to define the implementation for class field Subject.

6. Related Work

Classical module systems like the one of *Modula-2* [56], *Modula-3* [14], *Oberon-2* [40], and *Ada 95* [53] can be used to model modular aspects of software components well, but they have severe restrictions concerning extensibility and reuse. These systems allow type-safe separate compilation, but they hard-wire module dependencies by refering to other modules by name. This makes it impossible to plug in a module with a different name but a compatible specification without performing a consistent renaming on the source code level.

The module systems of *Oberon-2* and *C#* [25] allow to define local aliases for imported modules or classes. Here, one can easily replace an imported module with a compatible version just by modifying an alias definition. Such a modification would be destructive but would not require extensive source code renaming.

Initially, functional programming languages introduced module systems that obey the principle of *external connections* [19], 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. Module systems with external linking facilities include *SML*'s *functors* [35] and *MzScheme*'s *units* [20]. Opposed to *ML functors*, *units* offer separate compilation of independent modules with cyclic dependencies. *Units* provide first-class module abstractions and linking facilities to compose modules hierarchically. A general problem of *unit*-style module systems is scalability due to modules importing fine-grained entities like classes, functions, etc. and due to explicit module wiring. For this reason, *MzScheme* offers *signed units* that support bundles of variables, called signatures, which get linked in one step [19].

Only recently, proposals have been put forward to bundle class-based object-oriented languages with similar module systems [18, 4]. So far, we only know about two attempts to integrate a module system into *Java*. The proposal by Ancona and Zucca is rather theoretical, leaving unclear if their work is feasible in practice [4].

Independently to our work, Ichisugi and Tanaka observed that extensibility on the level of modules greatly enhances the ability to extend applications [27]. Ichisugi and Tanaka describe a practical module system for *Java* based on the notion of *difference-based modules*. Their modules are solely linked by a form of inheritance which also combines module members. Since their modules are not expressive enough to abstract over context dependencies (which are hard-wired), this module system must be seen rather as a tool for aspect-oriented programming [31] than for developing reusable, context independent software components. In Ichisugi and Tanaka's language, modules get exclusively linked by inheritance. Based on a similar idea, we investigated in former work a component calculus that explains component composition in terms of component refinements [57]. This component calculus supports a mixin-based composition scheme.

Duggan and Sourelis propose *mixin-modules* to make *ML* modules extensible [15]. An alternative proposal which is targeted towards *OCaml* [33] got recently published by Hirschowitz and Leroy [26]. Their work is based on *CMS* [3, 5], a simple but expressive module calculus which can be instantiated over an arbitrary core calculus. The calculus supports various module composition mechanisms including mixin module composition with overriding. The work on mixin-based composition goes back to Bracha who observed that inheritance can be seen as a general mechanism for modular program composition [8, 7]. With his work on the programming language *Jigsaw* [6], he lifts the notion of class-based inheritance and overriding to the level of modules. A consistent refinement of a family of classes is possible with the notion of *mixin layers*, introduced by Smaragdakis and Batory [48]. Related to mixins is the concept of *delegation*. Integrated into a statically typed object-oriented language, delegation yields a powerful mechanism for object-based inheritance [32, 13].

Rüping analyzed the modularity of object-oriented systems during design and specification [44]. He substantiates the need for modules in object-oriented languages as a means to encapsulate cooperating classes. Our module refinement and specialization mechanisms implement his notion of compatibility between modules which facilitates the type-safe extension of systems by the substitution of compatible modules.

Linguistic abstractions for component-oriented programming often have similar properties like module systems. Component-oriented programming languages that are built on top of *Java*-like object-oriented languages are *ComponentJ* [46, 45], *ACOEL* [49], and *ArchJava* [1]. [57] gives a short overview over these languages. *Jiazzi* [38] is a system for creating large-scale binary components in *Java* based on *MzScheme*'s *units*. *Jiazzi*'s units are conceptually containers of compiled *Java* classes with support for well-defined connections, externally specified through a set of imported and exported classes.

Component-oriented programming languages feature concepts originating from architectural description languages [39] like *ACME* [23], *Aesop* [22], *Darwin* [37], *Rapide* [34], *Wright* [2], *SOFA/DCUP* [43] etc. In general, architectural description languages are used to specify a software architecture formally. A software architecture describes the organization of a software system in terms of a collection of components, connections between these components, and constraints on the interactions [42, 47, 51]. By using architectural description languages, the details of a design get explicit and more precise, enabling formal analysis techniques. Furthermore, they can help in understanding the structure of a system, its implementation and reuse.

7. Conclusion

The paper presented *Keris*, an extension of the programming language *Java* with linguistic support for the evolution of software. The main contributions are

- a module system that combines the benefits of classical module systems for imperative languages with the advantages of modern component-oriented formalisms. In particular, modules are reusable components that can be linked with different cooperating modules without the need to resolve context dependencies by hand. Instead, *Keris* implicitly *infers* the module-wiring.
- a module composition scheme based on aggregation that makes the static architecture of a system explicit, and
- a type-safe mechanism for extending atomic modules as well as fully linked systems statically. This 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 that subsumes the original module, specializations are used to derive new (independent) modules from a given "prototype". *Keris*' extensibility mechanism is non-invasive; i.e. the extension of a module preserves the original version and does not require access to source code. Thus, extending modules does not invalidate existing code.

The overall design of the language was guided by the aim to develop a pragmatic, implementable, and conservative extension of *Java* which supports software development according to the *open/closed* principle: Systems written in *Keris* are closed in the sense that they can be executed, but they are open for extensions that add, refine or replace modules or whole subsystems without planning extensibility ahead. Another constraint was that we did not want to change *Java*'s compilation model or use a modified target platform.

The *Keris* compiler is based on an extensible *Java* compiler developed in previous work [59]. In the long run we plan to re-implement this compiler in *Keris* for two reasons: First, it would enable us to bootstrap the system. Furthermore, we would gain experience in using *Keris* for building large, extensible software.

Acknowledgments

I would like to thank the reviewers of the *First International Workshop on Unanticipated Software Extensibility* for their detailed and constructive comments. Furthermore, I am grateful to Martin Odersky for many fruitful discussions about related topics.

8. 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] R. Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, May 1997.

[3] D. Ancona and E. Zucca. A primitive calculus for module systems. In *Principles and Practice of Declarative Programming*, LNCS 1702. Springer-Verlag, 1999.

[4] D. Ancona and E. Zucca. True modules for Java-like languages. In *Proceedings of European Conference on Object-Oriented Programming*, LNCS 2072. Springer-Verlag, 2001.

[5] D. Ancona and E. Zucca. A calculus of module systems. *Journal of Functional Programming*, 2002.

[6] G. Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. PhD thesis, University of Utah, 1992.

[7] G. Bracha and W. Cook. Mixin-based inheritance. In N. Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications*, pages 303–311, Ottawa, Canada, 1990. ACM Press.

[8] G. Bracha and G. Lindstrom. Modularity meets inheritance. In *Proceedings of the IEEE Computer Society International Conference on Computer Languages*, pages 282–290, Washington, DC, 1992. IEEE Computer Society.

[9] K. B. Bruce. Safe static type checking with systems of mutually recursive classes and inheritance. Technical report, Williams College, Williamstown, MA, USA, 1997.

[10] K. B. Bruce. *Foundations of Object-Oriented Programming Languages: Types and Semantics*. MIT Press, Cambridge, Massachusetts, February 2002. ISBN 0-201-17888-5.

[11] K. B. Bruce, A. Fiech, and L. Petersen. Subtyping is not a good "Match" for object-oriented languages. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 104–127, 1997.

[12] K. B. Bruce, L. Petersen, and J. Vanderwaart. Modules in LOOM: Classes are not enough. Technical report, Williams College, Williamstown MA, USA, 1998.

[13] M. Büchi and W. Weck. Generic wrappers. In *Proceedings of the 14th European Conference on Object-Oriented Programming*, pages 201–225, June 2000.

[14] 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.

[15] D. Duggan and C. Sourelis. Mixin modules. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, pages 262–273, Philadelphia, Pennsylvania, June 1996.

[16] E. Ernst. *gBeta: A language with virtual attributes, block structure and propagating, dynamic inheritance*. PhD thesis, Department of Computer Science, University of Aarhus, Denmark, 1999.

[17] E. Ernst. Family polymorphism. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 303–326, Budapest, Hungary, 2001.

[18] R. B. Findler and M. Flatt. Modular object-oriented programming with units and mixins. In *Proceedings of the ACM International Conference on Functional Programming*, volume 34(1), pages 94–104, Baltimore, Maryland, 1999.

[19] M. Flatt. *Programming Languages for Reusable Software Components*. PhD thesis, Rice University, Department of Computer Science, June 1999.

[20] 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.

[21] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[22] D. Garlan, R. Allen, and J. Ockerbloom. Exploiting style in architectural design environments. In *Proceedings of SIGSOFT '94: Foundations of Software Engineering*, pages 175–188, New Orleans, Louisiana, USA, December 1994.

[23] D. Garlan, R. Monroe, and D. Wile. ACME: An architecture description interchange language. In *Proceedings of CASCON '97*, November 1997.

[24] 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.

[25] A. Hejlsberg and S. Wiltamuth. C# language specification. Microsoft Corporation, 2000.

[26] T. Hirschowitz and X. Leroy. Mixin modules in a call-by-value setting. In *Proceedings of the European Symposium on Programming*, Grenoble, France, April 2002.

[27] Y. Ichisugi and A. Tanaka. Difference-based modules: A class independent module mechanism. In *Proceedings of the European Conference on Object-Oriented Programming*, Málaga, Spain, June 2002.

[28] A. Igarashi. On inner classes. In *Proceedings of the European Conference on Object-Oriented Programming*, Cannes, France, June 2000.

[29] A. Igarashi and B. C. Pierce. Foundations for virtual types. In *Proceedings of the European Conference on Object-Oriented Programming*, Lisbon, Portugal, 1999.

[30] JavaSoft. JavaBeans™. `http://java.sun.com/beans`, December 1996.

[31] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming*, pages 220–242, Jyväskylä, Finland, 1997.

[32] G. Kniesel. Type-safe delegation for run-time component adaptation. In *Proceedings of the 13th European Conference on Object-Oriented Programming*, pages 351–366, Lisbon, Portugal, 1999.

[33] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. The Objective Caml system release 3.00, documentation and user's manual, April 2000.

[34] D. Luckham, L. Augustin, J. Kenney, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using Rapide. In *IEEE Transactions on Software Engineering*, April 1995.

[35] 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.

[36] O. L. Madsen and B. Møller-Pedersen. Virtual Classes: A powerful mechanism for object-oriented programming. In *Proceedings OOPSLA'89*, pages 397–406, October 1989.

[37] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proceedings of the 5th European Software Engineering Conference*, Barcelona, Spain, September 1995.

[38] S. McDirmid, M. Flatt, and W. Hsieh. Jiazzi: New-age components for old-fashioned Java. In *Proceedings of the 2001 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications*, October 2001.

[39] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. In *IEEE Transactions on Software Engineering*, volume 26, pages 70–93, January 2000.

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

[41] M. Odersky. Report on the programming language Scala. École Polytechnique Fédérale de Lausanne, Switzerland, 2002. `http://lamp.epfl.ch/~odersky/scala`.

[42] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. In *ACM SIGSOFT Software Engineering Notes*, volume 17, pages 40–52, October 1992.

[43] F. Plasil, D. Balek, and R. Janecek. SOFA/DCUP: Architecture for component trading and dynamic updating. In *Proceedings of ICCDS '98*, Annapolis, Maryland, USA, May 1998.

[44] A. Rüping. Modules in object-oriented systems. In *Technology of Object-Oriented Languages and Systems*, 1993.

[45] J. C. Seco. Adding type safety to component programming. In *Proceedings of the FMOODS 2002 Student Workshop*, University of Twente, The Netherlands, March 2002.

[46] 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.

[47] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[48] Y. Smaragdakis and D. Batory. Implementing layered designs with mixin layers. In *Proceedings of the European Conference on Object-Oriented Programming*, Brussels, Belgium, July 1998.

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

[50] C. Szyperski. Import is not inheritance — Why we need both: Modules and classes. In *Proceedings of the 4th European Symposium on Programming*, Rennes, France, February 1992.

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

[52] C. Szyperski. Modules and components — Rivals or partners? In *The School of Niklaus Wirth: The Art of Simplicity*. Morgan Kaufmann Publishers, 2000.

[53] 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.

[54] K. K. Thorup. Genericity in java with virtual types. In *Proceedings of the European Conference on Object-Oriented Programming*, LNCS 1241, pages 444–471, June 1997.

[55] M. Torgersen. Virtual types are statically safe. In *5th Workshop on Foundations of Object-Oriented Languages*, San Diego, CA, USA, January 1998.

[56] N. Wirth. *Programming in Modula-2*. Springer Verlag, Berlin, 1982.

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

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

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