Dynamic Configuration of Software Product Lines in ArchJava

Sebastian Pavel[1], Jacques Noyé[1,2], and Jean-Claude Royer[1]

[1] OBASCO Group, Ecole des Mines de Nantes - INRIA
4, rue Alfred Kastler, Nantes, France
[2] INRIA, Campus Universitaire de Beaulieu
Rennes, France
{Sebastian.Pavel, Jacques.Noye, Jean-Claude.Royer}@emn.fr

Abstract. This paper considers the use of a state-of-the-art, general-purpose, component programming language, specifically ArchJava, to implement software product lines. Component programming languages provide a more straightforward mapping between components as assets and components as implementation artifacts. However, guaranteeing that the implementation conforms to the architecture raise new issues with respect to dynamic configuration. We show how this can be solved in ArchJava by making the components auto-configurable, which corresponds to replacing components by component generators. Such a scheme can be implemented in various ways, in particular with a two-stage generator. This solution goes beyond the initial technical ArchJava issue and complements the standard static generative approach to software product-line implementation.

1 Introduction

The *software product-line* approach [9, 11] is one of the three use cases of *software architectures* [6, 15, 26, 27]. When applicable, it represents one of the most promising approaches to increased reuse of software, increased quality, and decreased time-to-market and maintenance cost. The software product-line approach is an intra-organizational software reuse approach that has proven successful and achieved substantial adoption by the software industry.

The key to software reuse is to move the focus from engineering single systems to engineering families of systems. Software product lines take advantage of the commonalities and variabilities that define the software architecture of these families in order to delay design decisions. A basic strategy is to divide the process of creating product-line applications in two major activities: *Domain Engineering* and *Application Engineering* [11]. Domain Engineering is the activity of creating reusable assets. Application Engineering is responsible for creating specific products according to customer needs, by reusing and specializing the assets created during the former activity.

Software components [16, 28] are the building blocks of modern software architectures. In the software product-line approach, Component-Based Software

Engineering (CBSE) is at the root of Domain Engineering. In fact, most of the reusable assets created during this activity are components targeted to be reused in specific products.

On the one hand, Architecture Description Languages (ADLs) have been defined to describe, model, check, and implement software architectures [23]. Most of the ADLs help specify and analyze high-level designs. On the other hand, so called *component programming languages* (e.g. Fractal [10], ArchJava [3], Koala [30]) integrate different ideas from ADLs into practical programming languages. As a result, component programming languages are interesting target implementation languages for software product lines. They potentially provide a more straightforward mapping between components at the design level and the implementation. This holds the promise of facilitating the traceability between the design and the implementation level, and of improving the quality of the software products.

As a way to better understand the level of support provided by current state-of-the-art component programming languages, we considered a standard example of software product line, well-described in the literature: the bank account example of [11]. We first implemented this case study using Java as a basis for our experiment, and then considered its implementation in ArchJava. ArchJava is a component programming language that extends Java with architectural features that enforce *communication integrity* [21, 24]. The ArchJava architectural specifications, which include an explicit hierarchical architecture, components, communication interfaces called *ports*, bindings and connectors, are similar to those of Darwin [22]. Darwin is an ADL designed to support dynamically changing distributed architectures and ArchJava inherits some of its dynamic capabilities like dynamic component creation and definition of communication patterns between components. Another good point of ArchJava is that, compared to other component language prototypes, its implementation is fairly robust. In spite of these qualities, the implementation of our case study turned out not to be so easy. A major issue was the constraints imposed by communication integrity on dynamic configuration, making it impossible to configure a component from the outside.

This paper reports on this experiment. It describes how the above-mentioned problem can be solved by making the components auto-configurable, which corresponds to replacing components by component generators. This is presented as a general pattern. From an ad hoc implementation, two other implementations are derived, an interpretive one based on *connectors* and a compiled one based on a second generator. This leads to a two-stage configuration scheme whereby a static generator, resulting from Domain Engineering, produces a dynamic generator responsible for dynamic configuration. This solution goes beyond the initial technical ArchJava issue and complements the standard static generative approach to software product-line implementation.

The reminder of this paper is structured as follows: Section 2 presents our case study and discusses its object-oriented implementation in Java. Section 3 summarizes the main features of ArchJava. Section 4 describes the issue raised

by communication integrity and presents our dynamic configuration pattern and its implementation. Related work is pointed out in Section 5. The paper ends with a conclusion and some ideas about future work.

2 Case Study and Object-Oriented Implementation

We start from a case study well-described in the literature. First, we provide a classical object-oriented implementation in Java and then, we discuss its applicability in software product-line development.

We choose to base our work on the study of the bank account example as described in [11]. The reason for doing this is that the domain of banking applications is very well-suited to illustrating the difficulties that appear when trying to apply a certain technology in a business domain. It is also a well-described case study including both Domain and Application Engineering with an implementation in C++.

In Fig. 1, we present an excerpt of the *feature model* representing the concept of "personal account" as detailed in [11]. The main advantage of such a feature model is that it makes it possible to represent variability in an implementation independent way. For instance, UML diagrams force the designer to choose implementation mechanisms like inheritance, composition or template classes when modeling variation points.

The diagram shows some of the possible variation points: `owner`, `payment`, `currency`, and `number of transactions`. All but `payment` are features with several *alternative* subfeatures (symbolized by empty arcs of a circle). The subfeatures of payment are *or*-features (symbolized by filled arcs of a circle), that is, we can select any nonempty subset of them. For the sake of simplicity and because the presented features are sufficient to exemplify the difficulties encountered, we do not include other details (see [11] for a complete analysis).

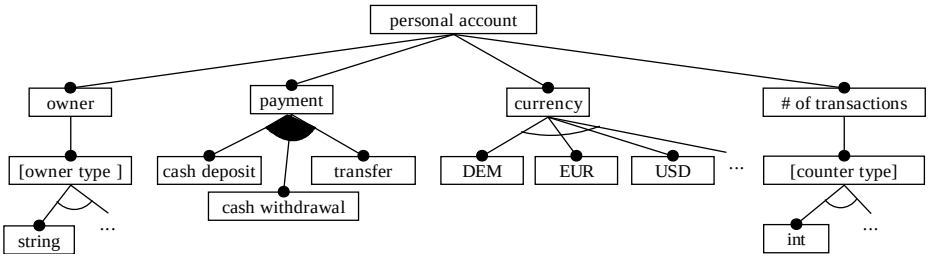

Fig. 1. Feature model of "personal account"

The next subsection presents a classical implementation approach of the case study in the Java object-oriented language .

2.1 Object-Oriented Implementation

There are various object-oriented design techniques for implementing variability in a software product line. The authors of [5] analyze the use of aggregation, inheritance, overloading, etc. A short survey about this work will be done in Section 5.

For the sake of illustration, let us consider a solution based on inheritance. From an object-oriented point of view, it is natural to express commonalities as abstract classes (or interfaces) while variabilities are implemented in concrete subclasses.

Using these ideas, we can implement the *personal account* concept as a Java class called `Account` (see Fig. 2) that offers three *services/methods* to clients: `deposit`, `transfer`, `withdraw`. The aggregated entities of `Account` (`owner:String`, `nbTransactions:String`, `Currency` and `PaymentFacilities`) represent the variation points in the architecture. The UML class diagram follows closely the *AbstractFactory* pattern ([14]) to express the `Account` configuration. In [20] Jézéquel presents and discusses the benefits and applicability of a very similar approach.

The `AccountFactory` class defines abstract methods to configure `Account` object instances. These methods are implemented in different ways depending on the subclasses of `AccountFactory`. The only concrete method is `makeAccount`, which creates a non-configured `Account` instance. This instance is responsible for calling back the configuration methods (`generateCurrency`, `generatePaymentFac`) on `AccountFactory` when needed.

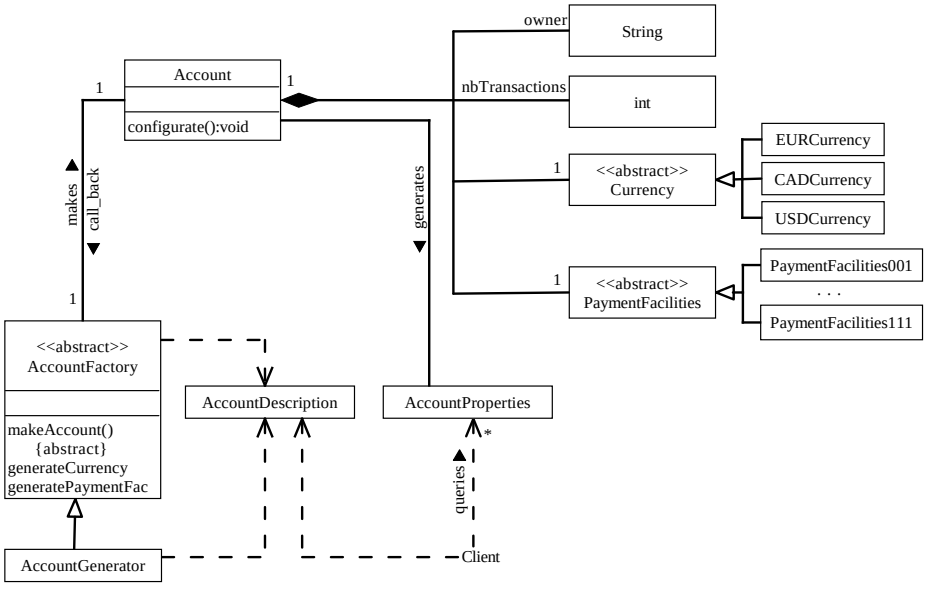

Fig. 2. Class diagram modeling the bank application

The `PaymentFacilities` class encapsulates the specification for a bank account that provides no payment facilities (the methods for deposit, transfer and withdraw are abstract), while the classes `PaymentFacilities001` to `PaymentFacilities111` provide the different combinations of the three possible services: each digit in the class name suffix indicates whether the services `deposit`, `transfer`, and `withdraw`, respectively, are present (1) or absent (0).

Considering the concrete factory `AccountGenerator`, a client creates an `AccountDescription` object first, then passes it to the `AccountGenerator` and finally calls the business methods on the newly instantiated `Account` object.

When trying to perform the operations provided by `Account`, the client calls the corresponding method on the object instance. Because the actual subclass that incorporates the payment facilities is `PaymentFacilities`, all the requests are forwarded to it. Before actually performing an operation, this object queries the different subcomponents of `Account` to check the validity of the request and the availability of the `Account` instance.

If the client wants the properties of an account at a given moment, it can request an `AccountProperties` object representing both the internal structure and the financial status of the account.

2.2 Discussion

When speaking about software product lines, there are some items like configurability, reuse and evolution that have to be taken into account. The following paragraphs discuss each of these items.

For the presented model, the configurability mechanism is fixed in the framework. The newly created `Account` instance calls back the `AccountFactory` instance, which, based on the information stored by `AccountDescription`, sets up the variation points in `Account`.

The framework can be reused only to create instances sharing the same internal structure. In other words, there is no mechanism to create `Account` instances having different instance structures.

When speaking about evolution, the framework could be evolved either by adding new variants at the existing variation points (`Currency`, `PaymentFacilities`) or by adding new variation points.

In our implementation (as in any other implementation based on a classical object-oriented language), what it is called "software component" is actually represented by an ordinary class. A component instance is therefore an object. In much more complicated examples, we expect that the relation between component and class would be 1 to N, making the task of understanding and evolving "component-based" applications more difficult.

Using component programming languages makes software architectures easier to understand, implement and evolve. Components in architectures have a corresponding component instance in implementations and the connections between implementation components are more explicit. Evolving an application means changing components and connections, which is much easier than evolv-

ing a set of related classes (playing the role of components) and their relations with other sets of classes.

In the next section we present ArchJava, a component programming language, and in the following sections, our proposals based on this language.

3 ArchJava

ArchJava [2, 1, 3] is a small, backwards-compatible extension to Java that integrates software architecture specifications into Java implementation code. It extends a practical implementation language to incorporate architectural features and enforce *communication integrity* [21, 24]. *The benefits of this approach include better program understanding, reliable architectural reasoning about code, keeping architecture and code consistent as they evolve, and encouraging more developers to take advantage of software architecture [3].*

In ArchJava, a *component* is a special kind of object capable of communicating with other components in a structured way. The communication is performed using logical communication channels called *ports*. Each port is allowed to declare methods qualified by the keywords `requires` and `provides`. Only the provided methods have to be implemented in a component.

The hierarchical software architecture is expressed with *composite components* made of connected subcomponents. To connect two or more ports in an architecture, the `connect` primitive is employed. This primitive binds each required method in a port to a provided method with the same signature in other ports.

It is possible to create pass-through connections to subcomponents or to other ports using the `glue` primitive. The glue primitive differs from the connect primitive in that it glues the inside of a port to another port instead of connecting the outside of that port.

ArchJava supports component inheritance and architectural design with abstract components and ports. This allows an architect to specify and type-check an ArchJava architecture before beginning program implementation.

ArchJava also supports the design of dynamically changing distributed architectures. It allows the creation and connection of a dynamically determined number of components. Components can be dynamically created using the same `new` syntax used to instantiate objects in Java. At creation time, each component records the component instance that created it as its *parent component*.

Dynamically created components can be connected together at runtime using *connect expressions*. Each connect expression must match a *connect pattern* declared in the enclosing component. A connect pattern describes a set of possible connections. A connect expression matches a connection pattern if the connected ports in the expression are identical with those in the pattern and if each connected component instance is an instance of the type specified in the pattern.

Often, a single component participates in several connections using the same conceptual protocol. In ArchJava, a *port interface* describes a port that can be instantiated several times to communicate through different connections. Each

port interface defines a type that includes the set of the required methods in the port. A port interface type combines a port required interface with an instance expression that indicates which component instance the port belongs to. Port interfaces are instantiated by connect expressions that return objects representing the corresponding connections.

The connector abstraction supported by ArchJava [4] cleanly separates reusable connection code from application logic, making the semantics of connections more explicit and allowing the easy change of the connection mechanism used in a program. In ArchJava, each connector is modularly defined in its own class and the components interact with the connectors in a clean way using Java method call syntax. The connector used to bind two components together is specified in a higher-level component. In this way the communicating components are not aware of and do not depend on the specific connector used. This makes it easy to change connectors in a system without having to modify the communicating entities.

Developers can describe both the runtime and type-checking semantics of a connector by using the `archjava.reflect` library, which reifies connections and required method invocations.

Connectors get instantiated whenever a connect expression that specifies a user-defined connector is executed at runtime. *The principal benefit of using connectors in ArchJava is that the same connector can be reused to support the same interaction semantics across many different interfaces, while still providing a strong, static guarantee of type safety to clients [4].* The main drawback is that they are defined using a reflective mechanism implying runtime overhead associated with dynamically reifying method calls.

Communication integrity [21, 24] is a key property in ArchJava. It ensures that the implementation does not communicate in ways that could violate reasoning about control flow in the architecture. *Intuitively, communication integrity in ArchJava means that a component instance A may not call the methods of another component instance B unless B is A's subcomponent, or A and B are sibling subcomponents of a common component instance that declares a connection or connection pattern between them [2].* ArchJava enforces communication integrity in the cases of direct method calls and method calls through ports using both static (compile-time) and dynamic (runtime) checks. It also places restrictions on the ways the components are used. In particular, subcomponents are not allowed to escape the scope of their parent component.

4 Proposals/Experiments

Let us now see how to implement the example of Sect. 2 using ArchJava.

Our proposals are oriented towards facilitating the use of ArchJava components in dynamic architectures. More specifically, we want components to be created and (re)configured at runtime. In the case of our banking application, the architecture is far from being static. The accounts are created, configured and connected dynamically at any time after the application is started and running.

It is also necessary to allow the reconfiguration of these account components at runtime without having to instantiate new components to meet the new requirements. For example we can imagine an account component that allows `deposit` and `transfer` as its services. At runtime, we could also add the `redraw` service. This change has to be done without recreating a new account (providing the three services) but by seamlessly modifying the existing one.

Based on Java, ArchJava supports only a limited number of the mechanisms presented in [5] that could be used to code variability. We choose to use inheritance as for the Java implementation of the bank account example. The commonalities are represented by abstract component classes while the variabilities ar represented by concrete ones.

Due to the fact that the ArchJava model focuses on keeping the communication integrity property, components cannot be passed as arguments through ports. This makes impossible the creation of components outside a component and their assignment as subcomponents of this component. So, there is no mechanism for configuring a component when the configuration details are located outside the component. The only solution is to encapsulate in the composite component all the information about the possible configurations of its subcomponents. At runtime the component instance takes some external description and uses it to specialize its direct subcomponents. This mechanism is heavy and not extendible. Each time we add a new component class or we extend an existing one, each component including the configuration information has to be updated and then reinstantiated in all the applications that use it.

A more natural way of performing internal configuration of a component is to keep the needed information outside the component and to use specific ports to perform the configuration. The diversity interfaces in the Koala model [29, 30] provide such a mechanism but only for static configuration. In many cases a static approach is sufficient. However, in the case of a product line that requires the dynamic creation and configuration of components in an architecture, ArchJava does not provide a direct mechanism to implement variability.

In the case of static architectures where the component are created, configured, and connected at compile time, the general components could be specialized for a specific application using generators. Using ArchJava, a generator takes all the specifications for the specific application and all the general components and specializes these components to match the specifications. The result is a specific architecture made of specific components. After the instantiation of the components nothing can be modified. The architecture remains the same along the life of the application.

Adopting a dynamic approach, all our proposals are based on the same pattern. To facilitate the internal configuration of a component we use what we call a component configurator (see `ComponentConfigurator` in Fig. 3). This special component plays the role of a dynamic generator. At runtime, it takes an external description containing implementation details for an account. Using this description, it generates its specific component (`Component` in Fig. 3) as a subcomponent, declares the same external ports and glues these ports to the ports

published by `Component`. A client (the `Application` using its *internal port*) sees and interacts only with the component configurator instance as it provides the same ports as the actual one (the generated one).

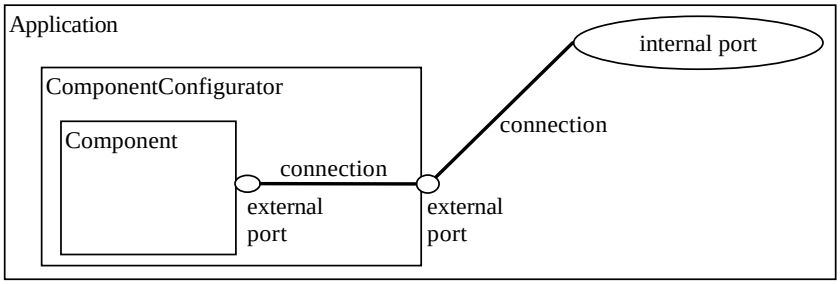

Fig. 3. The Component Configurator Pattern

While all three proposals are based on the same pattern, there are also differences either in the way of creating the component configurators or in the way the components are connected and communicate.

In the first proposal, all the component configurators (there is one for each composite component type) are created manually by the software engineer. Trying to overcome this drawback, the second proposal addresses the issue of using static software generators (created during Domain Engineering). We see these generators as software entities used to automatically generate (usually, during Application Engineering) all the component configurators in an application, starting from a set of input descriptions. In the third proposal, we use custom connectors instead of simple connections between components to specialize communication at runtime and avoid the generation of multiple component configurators.

In the following, we present the details of each proposal focusing on the issues raised by the implementation of the bank account example.

4.1 Ad Hoc Implementation

The first proposal is the direct implementation of the ideas presented above.

Principles. Fig. 4 presents a configuration based on the pattern in Fig. 3. This configuration involves two account instances (`AccountX` and `AccountY`) and two corresponding component configurators (`ConfigAccountX` and `ConfigAccountY`). The two component configurators are connected to two internal ports (`properties` and `payment`) of `BankApplication`, which plays the role of the banking application.

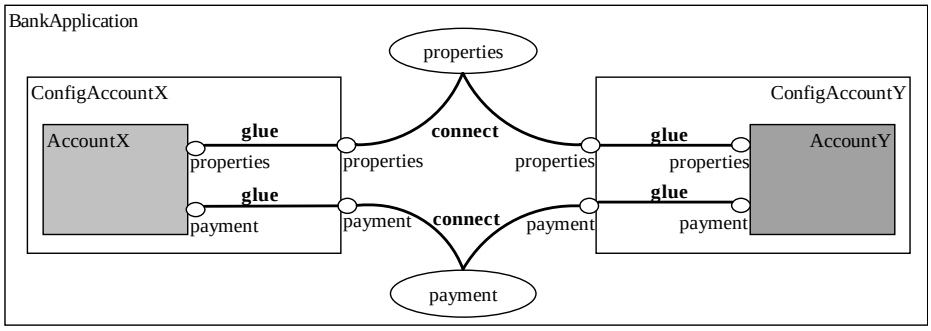

Fig. 4. Pattern Based Configuration

In general, the `BankApplication` component instance will dynamically create `ConfigAccount` subcomponents as needed. A `ConfigAccount` is responsible for instantiating and configuring its `Account` subcomponent. The ports and associated methods declared by `Account` are the same as those declared by `ConfigAccount`. The ports are connected using the glue primitive. All the outside requests to a port of `ConfigAccount` are delegated to the glued port in the `Account` component. Methods in the ports declared by `ConfigAccount` do not need to be implemented, they are just declared identically as those in the ports of `Account`.

The two accounts are represented by `ConfigAccountX` and `ConfigAccountY`, respectively. Each of them declares two ports:

- `properties`, to give access to the properties of the account instance,
- `payment`, to give access to the payment services.

To allow the dynamic connection of `ConfigAccount` instances we use the dynamic connection facilities of ArchJava. We declare just once a connect pattern for each port:

```
connect pattern payment, ConfigAccount.payment;
connect pattern properties, ConfigAccount.properties;
```

representing the pattern of connections between the *internal* ports of `BankApplication` (`properties` and `payment`) and each port of the `ConfigAccount` component class. After creating a `ConfigAccount` instance we use connect expressions, for example: `connect(payment, account.payment)`, to actually connect the ports.

The implementation details of an `Account` component are presented in Fig. 5. All the subcomponents representing variation points in `Account` (`Currency`, `PaymentFacilities`) are connected to only one `internal` port in `Account`: the port `properties` within `AccountX`. `Currency` publishes a port named `type`, containing its specific methods. `PaymentFacilities` declares two ports: `payment` and `interncommunication`. The first one represents the services (`deposit`,

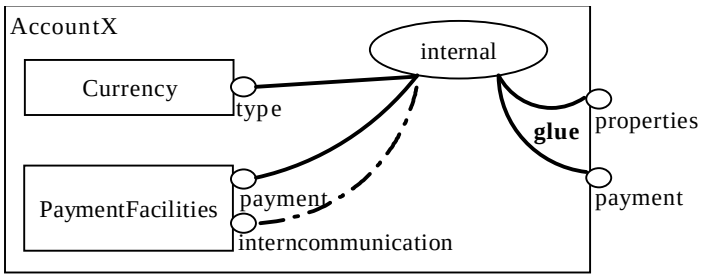

Fig. 5. Account Details

`transfer`, `withdraw`) provided by `PaymentFacilities`. The second one is only used to communicate with the other subcomponents of `Account` (`Currency` in our particular case). This communication is needed for checking the necessary internal properties before answering payment service requests. The purpose of connecting all these ports to the same `internal` port (`properties` in `AccountX`) in their parent component is to centralize all the possible method calls coming either from the outside or from the inside. This need becomes more evident when we multiple subcomponents need to communicate together in order to reply to a client. For example, `PaymentFacilities` needs to call methods in the `Currency` port before validating client requests. Instead of declaring all the possible connections among the subcomponents, we choose to declare a kind of proxy port: the `internal` port. Ports `properties` and `payment` are glued to the same internal port.

Implementation. The configurability of components at runtime is the purpose of this design and using a component configurator is the key to realize it in ArchJava. The component configurator plays the role of a dynamic generator. To create a specific `Account` instance at runtime, a client simply passes some input description data to the `ConfigAccount` instance. This data includes the name of the account component and parameters defining its variation points. A parent component can only access public methods in its direct subcomponents. In this case, `Currency` and `PaymentFacilities` are not visible from `ConfigAccount`. To allow the configuration of a specific `Account` component (containing specific `Currency` and `PaymentFacilities` subcomponents), `ConfigAccount` generates the corresponding source code, compiles it and then instantiates it.

The actual mechanism used for specializing the `Account` component is inheritance. Firstly, we declare a general `Account` component class. Secondly, we generate the source code for a specific `Account` component, let us say `AccountX`, that extends the `Account` class. The effort of generating a specific account is reduced due to the fact that we will generate only the needed functionality according to input requirements. These requirements are transmitted to the constructor of `ConfigAccount`. When creating a new `ConfigAccount` instance with these requirements, the constructor also creates the required `Account` instance.

It is also possible to reconfigure an existing `Account` component at runtime. We simply pass the new configuration parameters to the corresponding `ConfigAcount`. While it remains unchanged when regarded from outside, `ConfigAccount` reinstantiates the required `Account` subcomponent. Notice that the connections between `ConfigAccount` and its possible clients remain unchanged, only inside glue connections need to be reestablished.

Discussions. The ideas presented in the example above can be fully applied to any other application that requires the creation, specialization and dynamic connection of ArchJava components at runtime. This is an important requirement for product-line applications requiring the specialization of composite component at runtime.

The example presented above considers only one configuration component (`ConfigAccount`) because we have only one composite component in the application. In a product-line approach, however, there are usually a large number of possible composite components. Applying this proposal requires to manually define a generic component class and a specific configuration component class, for each composite component.

The effort of building the assets (component and component configurator classes) for this proposal in the phase of Domain Engineering for a product line is usually huge. Since the components are manually defined, the code can easily become error prone. When trying to evolve a component in a product-line application, we also need to modify the corresponding component configurator, for instance by inheriting from the older one. When adding new components, new configuration components have to be created, too. Doing all this work manually is not an easy task. The next proposals try to address this shortcoming.

4.2 Using Component Configurator Generators

The second proposal aims to reduce the effort of building the components by using generators.

Principles. The idea is to automatically generate all the component configurators. Instead of manually implementing a component configurator (e.g. `ConfigAccount` in Fig. 4) for each composite component in the application, a generator is made responsible for automatically creating the configurators. The generator takes a description of all the composite components in the application and generates the corresponding component configurators. The generator is made as general as possible. It encapsulates all the knowledge for generating the different component configurators needed in different product lines.

The generator represents an additional tool that comes with the product-line software. It is created during the Domain Engineering phase and it could be reused each time component configurators are required (either at Domain or Application Engineering time).

To properly generate component configurators, the generator needs a detailed description of the corresponding components. Another idea is to make the generator as general as possible to allow the generation of as many different component configurator structures as possible.

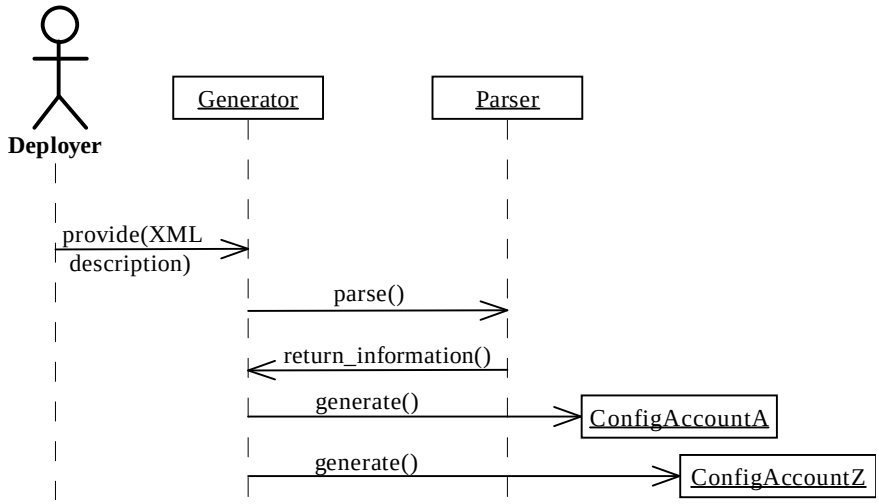

Fig. 6. Generator Scenario Sequence Diagram

Implementation. While generators are usually very complex tools, in our case the generator is a simple Java program. We used it to test the creation of component configurators for the bank account example. It takes a description of our general components and generates the corresponding component configurators. This description includes the names of the general (abstract) components serving as a base for the creation of new specific generated components. In addition, for each general component, there is a list of ports and the corresponding `requires/provides` methods that the specific component will contain. This information is also used to create the ports (glued to the actual specific component) in the component configurator.

To simplify as much as possible the component descriptions, we choose to use XML and additional technologies [32]. The generator takes a `.xml` file containing a structured description of components. It uses a parser to extract the information and then generates the component class files (see Fig. 6).

At the implementation level, the XML file conforms to a predefined XSL schema. This schema describes how the `.xml` documents are structured. To facilitate the use of XML files in Java applications, we rely on JAXB (Java XML Binding) technology [19].

Discussion. The generator generates components starting from a general (common) architecture and by adding functionality. In our case, this common architecture is related to the fact that all the generated configuration components deal with the same kind of account. In other applications, it is the structure of the configured component that dictates the common architecture of all the configuration components.

Based on the first proposal, the second one is more appropriate to the product-line approach. The generator is very general, it could be reused across many product-line applications. The effort during the application deployment is reduced to the simple definition of an XML file containing the descriptions of the components, to be passed as an argument to the generator.

Another benefit of using generators is that the resulting code is more reliable than handmade code. In particular, code complexity and performance can be controlled by using generators. A generator encapsulates the logic for creating and optimizing the code. Once developed and tested, a generator guarantees the quality of the resulting generated applications.

At runtime, the application remains as configurable as in the first proposal. A component is configured using another corresponding component configurator.

The asset that is reused the most in this approach is the generator. The fact that it is made as general as possible, encapsulating the logic for creating any kind of component and its corresponding configuration component, makes the generator a first order asset in a product-line application. It is reused for each component type necessary in an application.

The evolution of existing components, seen as assets of the product line, is a simple step: we just change the descriptions of the desired components and the generator creates the corresponding component types. After this step, the resulting assets (component and component configurators types) can be reused in any kind of product-line application that requires their presence.

4.3 Using Custom Connectors.

The third proposal is based on connectors, as described in [4].

Principles. Instead of manually building or automatically generating all the configuration components as in the first and second proposal, why not have only one component configurator that is as general as possible (see Fig. 7).

A component has its ports connected to a single port of the `ConfigComponent`. Instead of a classical connection we choose to employ a custom connector. The role of the connector is to intercept all the invocations coming either from inside the component instance or from outside the `ConfigComponent`. Once intercepted, the requests are transformed to match the signature of the target method in the specified port. A component instance in a product line could have a number of ports not known when building the configuration component. Moreover, the number of components in the architecture is not constant during the life cycle of the product-line application. In this context, it is important to define a

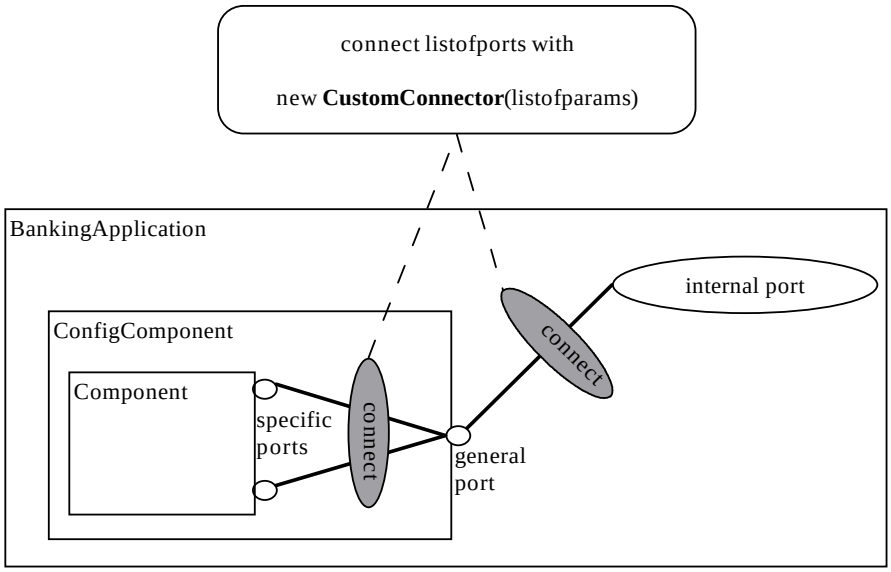

Fig. 7. The Connector Pattern

generic port that can potentially be connected to any other port. To allow such flexibility, there are two major difficulties:

- the definition of the port (general port in Fig. 7) in the `ConfigComponent` component, and
- the definition of the the custom connector that codes/decodes the requests between the ports.

When intercepting calls from methods in component ports, the connector first encodes all the information about the request: required method signature and calling parameters. Then, it calls the methods in `ConfigComponent` port with this information. From there, the information is decoded by another custom connector instance and transmitted to the final port. The connector represents a communication channel that is invisible to the ports at its edges. It has no knowledge of the specific ports it connects. Its job is just to code/decode and forward the requests.

Implementation. Unfortunately, the actual version of the prototype compiler of ArchJava is not fully operational with respect to the connector specifications. For this reason we could not test whether the ideas presented above could be successfully applied to create a specific custom connector.

Discussion. The use of custom connectors in ArchJava simplifies the previous proposals. When assembling a product-line configuration we simply plug

in a well-defined custom connector between the ports we need to connect. The step of creating the configuration components for each component type in the application is omitted. A general custom connector is created during Domain Engineering. Then, it is reused each time we need to connect a component instance to its configurator or each time we connect the configurator port to the general application ports. The code encapsulated in the general connector is very complex. It takes into account all the possible communication possibilities between the edges it connect. For the same reason and because in ArchJava connectors are implemented using a reflective mechanism, the performance of such a connector component is limited. It is a kind of "interpreted" version of the previous "compiled" proposal using generators.

The advantage is that the connector, as the generator in the second proposal, is reusable across all the specific products in the product line. We simply plug in the connector between two components. The connector will eventually perform exactly the same in all the cases.

5 Related Work

5.1 Implementing Product-Line Variabilities with Components

In [5], Anastasopoulos and Gacek survey a number of techniques making it possible to produce code for generic product-line assets. This includes a wide range of techniques going from conditional compilation to aspect-oriented programming, but component-oriented programming is not considered. However, component-oriented programming has some very interesting properties with respect to the list of qualities that Anastasopoulos and Gacek consider for assessing the implementation techniques: scalability, traceability, and separation of concerns. Indeed, scalability is one of the main objectives of component-oriented programming, which is designed to program in the large. Traceability is improved by keeping the architecture explicit in the implementation. Actually, this goes beyond traceability as conformance of the implementation to the architecture of the product line is provided. Finally, component-oriented programming facilitates the separation of architectural concerns and global reasoning from behavioral concerns and local reasoning.

5.2 Koala

In the middle of the 90's, Philips had some in-depth experience in developing a large range of televisions worldwide. The hardware was reasonably modular, but the software parts were developed using a classic approach. To handle diversity the approach mainly used compiler switches, runtime options and code duplication with changes. But the new coming products needed more and more functionalities and combinations of these functionalities. The company had to integrate different pieces of software coming from different areas and developed at different times. Since, at this time, the existing component technology was not suited to the existing constraints, the Koala language was designed [30].

Koala was inspired by the Darwin language [22] dedicated to distributed system architectures. Koala is a component programming language that provides interfaces (provided, required), first-class components, configurations and that is targeted to the C language. Most of the connections are known at configuration time, but the language also defines switches or code to dynamically bind components. It offers an easy-to-use graphical notation and an elegant parameterization mechanism. Interface diversity is a specific kind of interface to manage configuration from outside of the component, providing better independence from context. Partial evaluation is used to optimize static configuration. A compiler has been developed but also a user interface and component web-based repositories. Koala was designed for resource-constrained systems, it makes it possible to apply the component technology at a small grain and with no significant overhead.

The main criticism that can be made to Koala is that it deals exclusively with static architectures. Once created, a configuration already deployed cannot be modified in the final product (in C code). Any modification requires to change the configuration, possibly the definitions of components and interfaces, and then to recompile all the application. This perfectly fits the domain of consumer electronics where the architecture of internal components, once deployed, will not change. However, there are applications requiring a dynamic architecture, an architecture that makes it possible to create new components and connections or to change existing ones during the lifetime of the product. In its actual form, Koala is not usable to describe and implement such a dynamic architecture.

On the other hand, ArchJava, inspired from the same Darwin [22] language as Koala, was especially designed to support dynamically changing architectures. Even if Koala proved its benefits in developing software product lines in the domain of consumer electronics, ArchJava seems to be more suited to implementing software products with a dynamic structure.

5.3 Components and Generators

The idea that generators are an effective way of automating component customization and assembly in the context of program families is not new (see for instance [8, 18, 13, 12]). However, generators are usually presented as operating at compile time only. From a description of the configuration of the product to be built, they customize and assemble prefabricated generic components in one stage. Our case study has hopefully made clear that, although this approach remains valid when dynamic configuration must be taken into account, it has to be generalized to a two-stage approach. This means that, apart from dealing with static customization and assembly, the (static) generator has also to prepare dynamic generation. This can be done in two different ways: an interpretive way, relying on connectors, or in a more specific and efficient way, using a (dynamic) generator.

6 Conclusion

In order to assess the benefits of using a component language to implement software product lines, we have experimented with different implementations in ArchJava of a well-known example from the literature. When implementing component-based applications, software architectures in ArchJava are much more explicit than architectures in a standard object-oriented language, like Java. Using components makes the architecture application architecture easier to understand, implement and evolve.

However, trying to implement a software product-line application in ArchJava is not an easy task. Even if the language is well suited to dealing with architectural concepts, communication integrity turns out to be a constraint. This important consistency property in ArchJava guarantees that the *implementation components only communicate directly with the components they are connected to in the architecture [3].* The drawback is that, due to this property, a component instance cannot be passed as an argument through ports, an important requirement for a composite component that has to be specialized (or configured) at runtime.

There is no direct relation between the software product-line approach, which is the organization of some activities to achieve reusable software architectures, and communication integrity, which is an architecture implementation property. Despite this remark, the modality of integrating this property into the ArchJava language does not allow the easy creation of software product lines with dynamic architectures.

Our main contribution was to design and develop a pattern allowing the implementation of software components in ArchJava that can be dynamically specialized at runtime. In addition to this advantage, the pattern also keeps the communication integrity property as proposed by ArchJava.

All our implementation proposals are based on this pattern. The basic proposal uses what we call a *component configurator*. This is actually a dynamic generator used to create/configure component instances at runtime. The second proposal automates the ad hoc building of component configurators using static generators. These generators are used to automatically generate all the component configurators. The third proposal replaces the generation of component configurators by the use of a unique custom connector. While the first proposal was fully implemented in ArchJava, the second one was only partially implemented in order to solve our specific case study. The implementation of the third proposal is still waiting for the ArchJava prototype compiler to come to maturity.

While initially developed to solve the problem of specializing component instances at runtime for our case study, the pattern could be successfully used to solve any problem requiring dynamic creation, specialization and connection of composite component instances at runtime. While the first proposal involves a great development effort, the connector-based proposal raises some efficiency problems. Based on two-stage generators, the second proposal is the most interesting in terms of generality and efficiency.

The question is then whether such a pattern could not be better supported within a general-purpose component programming language. We have seen that static configuration is well-covered in Koala through the combination of diversity interfaces, used to define configuration information, and partial evaluation, used to do the actual configuration, without the explicit manipulation of generators. This should be extended to dynamic configuration. As far as ArchJava is concerned, one could imagine to introduce a notion of *diversity port* that would be less restrictive than regular ports. A more general direction would be to provide linguistic and tool support for the two-stage generator approach. Another important issue that we have not addressed here is to explicitly support the specification of various forms of variabilities.

Acknowledgements

This work was supported in part by the *ACI Sécurité Informatique*, in the context of the DISPO project.

References

1. J. Aldrich, C. Chambers, and D. Notkin. ArchJava web site. http://www.archjava.org/.
2. J. Aldrich, C. Chambers, and D. Notkin. Architectural reasoning in ArchJava. In B. Magnusson, editor, *ECOOP 2002 - Object-Oriented Programming,16th European Conference*, number 2374 in Lecture Notes in Computer Science, pages 334–367, Malaga, Spain, June 2002. Springer-Verlag.
3. J. Aldrich, C. Chambers, and D. Notkin. ArchJava: Connecting software architecture to implementation. In ICSE2002 [17], pages 187–197.
4. J. Aldrich, V. Sazawal, C. Chambers, and D. Notkin. Language support for connector abstractions. In L. Cardelli, editor, *ECOOP 2003 - Object-Oriented Programming,17th European Conference*, number 2743 in Lecture Notes in Computer Science, pages 74–102, Darmstadt, Germany, July 2003. Springer-Verlag.
5. M. Anastasopoulos and C. Gacek. Implementing product line variabilities. In Bassett [7], pages 109–117.
6. L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Series in Software Engineering. Addison Wesley, Reading, MA, USA, 1998.
7. P.G. Bassett, editor. *Proceedings of SSR'01 - 2001 Symposium on Software Reusability*, Toronto, Canada, May 2001. ACM Press.
8. D. Batory and S. O, Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Programming Languages and Systems*, 1(4):355–398, October 1992.
9. J. Bosch. *Design and Use of Software Architectures - Adopting and Evolving a Product Line Approach*. Addison-Wesley, 2000.
10. T. Coupaye, E. Bruneton, and J.-B. Stefani. The Fractal composition framework. Specification, The ObjectWeb Consortium, February 2004. Draft, Version 2.0-3.
11. K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

12. K. Czarnecki and U.W. Eisenecker. Components and generative programming. In Nierstasz and Lemoine [25], pages 2–19.
13. S. Eckstein, P. Ahlbrecht, and K. Neumann. Techniques and language constructs for developing generic informations systems: A case study. In Bassett [7], pages 145–154.
14. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.
15. D. Garlan and M. Shaw. An introduction to software architecture. Technical Report CS-94-166, Carnegie Mellon University, School of Computer Science, 1994.
16. G.T. Heineman and W.T. Councill, editors. *Component-Based Software Engineering – Putting the Pieces Together*. Addison-Wesley, 2001.
17. *Proceedings of the 24th International Conference on Software Engineering*, Orlando, FL, USA, May 2002. ACM Press.
18. S. Jarzabek and P. Knauber. Synergy between component-based and generative approaches. In Nierstasz and Lemoine [25], pages 429–455.
19. Java web site. http://www.sun.java.com/.
20. J.-M. Jézéquel. Reifying variants in configuration management. *ACM Transactions on Software Engineering and Methodology*, 8(3):284–295, July 1999.
21. D.C. Luckham and J. Vera. An event-based architecture definition language. *IEEE Transactions on Software Engineering*, 21(9):717–734, 1995.
22. J. Magee and J. Kramer. Dynamic structure in software architectures. In *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 3–14, San Francisco, CA, USA, October 1996. ACM Press.
23. N. Medvidovic and R.N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, January 2000.
24. M. Moriconi, X. Qian, and R. A. Riemenschneider. Correct architecture refinement. *IEEE Transactions on Software Engineering*, 21(4):356–372, 1995.
25. O. Nierstasz and M. Lemoine, editors. *Software Engineering – ESEC/FSE'99: 7th European Software Engineering Conference*, volume 1687 of *Lecture Notes in Computer Science*, Toulouse, France, 1999. Springer-Verlag.
26. D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, October 1992.
27. M. Shaw and D. Garlan. *Software Architecture. Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.
28. C. Szyperski, D. Gruntz, and S. Murer. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, 2nd edition, 2002.
29. R. van Ommering. Building product populations with software components. In ICSE2002 [17], pages 255–265.
30. R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The Koala component model for consumer electronics software. *Computer*, 33(3):78–85, 2000.
31. J. Withey. Investment analysis of software assets for product lines. Technical Report CMU/SEI-96-TR-010, Software Engineering Institute, Carnegie Mellon University, 1996.
32. XML - Extensible Markup Language - web site. http://www.w3.org/xml.