

# Java/A

Florian Hacklinger

Institut für Informatik  
Ludwig-Maximilians-Universität  
Oettingenstraße 67, 80538 München, Germany  
`hackling@informatik.uni-muenchen.de`

**Keywords:** Software Architecture, Programming Languages

**Classification:** 4 month's work (beginning of PhD thesis)

**Abstract.** Software architecture is now over one decade a matter of research and software engineering. However there is a gap of architectural concepts in programming languages. Thus I propose Java/A (Java extended by architectural concepts), a programming language that supports architectural concepts by first order language constructs.

## 1 Motivation and Approach

There are no adequate architectural concepts at the level of programming languages. The architecture of software systems is in danger of being damaged due to invisibility during maintenance. Furthermore there are shortcomings in techniques that allow one to check if an implementation conforms to an architecture. So one major problem I see in the current use of software architecture is ensuring that an implementation conforms to the underlying architecture during its whole lifetime [Sel01].

As a solution I propose to extend a programming language, e. g. Java, by the architectural concepts *component*, *connector* and *configuration*. I call this approach Java/A.<sup>1</sup> Some programming languages support the concept *component* to a certain degree, e. g. Java features *package*; the insufficiency of *package* as architectural concept is evidently. Furthermore the component concepts Java Bean and Enterprise Java Bean are attached to Java, but these are no primary language constructs and do not suffice either. They support offered interfaces but do not provide required ones. So correctness of communication can not be assured. The concept *connector* is in traditional programming languages handled mostly implicitly, not explicitly as needed in software architecture. The concept *configuration* is missing at all.

Java/A fits in the gap between the model of an architecture given in an ADL, e. g. Wright, and “real” code in a traditional programming language. From a model given in an ADL may a Java/A program body be generated. The

---

<sup>1</sup> The name is chosen alluding to Modula/R.

Java/A code then may be translated into traditional Java. As an extension to my approach one may also consider other languages like Cobol or Modula-2 as base languages. To deploy the advantages of Java/A the level where the programming is done must be Java/A.

There are other approaches that can generate code from a model, e. g. ROOM or Rapide. ROOM has some deficiencies and Rapide just allows to create a simulation from a model. Java/A in contrast is designed to be used as a full programming language.

## 2 Outline of Approach

The Java/A approach is based on the so called “*Interface Connection Architectures*” [Luc96]. In detail we use the definition given in [Sto00]. That is, an architecture consists of self-contained components that interact via connectors, that are attached to the components’ ports. A set of components and their linking connectors is called a configuration. Components themselves can be configurations, thus components can be composed hierarchically. In Figure 1 I give a schematic example of such an architecture: two components (*P1* and *P2*) that are connected with one connector (*C1*).

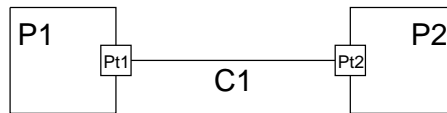


Fig. 1. Schematic Architecture Model

### 2.1 Java/A sample code

Java/A components are related to Java classes: they provide a capsule for data and algorithms. In contrast to Java classes, that do not control communication, Java/A components provide a strong encapsulation, such that communication occurs exclusively through ports. A Java/A component description consists of the following parts:

**Uses statements** Show used libraries, imported classes, etc. Similar to the *import* statement in Java. The uses statements allow programmers to use Java classes in Java/A in the way of wrapping Java/A concepts around them.

**Implementation style** Indicates the style to that this Java/A component should be translated. This may be for example one single class, a set of classes, etc. More about that in the next section.

**Variables** Similar as classes components can have component wide visible variables.

**Port definitions** Ports are the only points of interaction for this component. Thus they must define required and offered interfaces. A port has a type that describes the way it expects signals<sup>2</sup> to be transferred. Types may be PC, DataStream, Event, etc. Additionally a port's behavior, that is an order of signals to be received and sent, can be specified (e. g. with an UML state machine). In the this phase of Java/A I omit behavior, but it will be added later.

**Constructor and start method** Similar to Java classes a Java/A component must provide a constructor. The start method is invoked after assembly of a configuration to start processing.

**Signal handlers** To handle incoming signals for every signal in an offered interface an appropriate handler must be provided.

**Private methods** Components can have private methods.

According to this description I present an Java/A implementation of the component *P1* from the example in Figure 1. Due to place restrictions I will compress the code so that it is not in a proper kernel style.

```
component P1 {
  uses java.lang.Math;
  implementation style: Complex Class;
  int y; int z;

  port Pt1 {
    type PC;
    in { void setY(int x); }
    out { int getX(); }
  }

  constructor P1() { y = 1; }
  void start() { z = Pt1.getX(); }
  void setYImpl() implements Pt1.setY(int x) { y = x; }
}
```

The same as for components applies also to connectors (except that connectors do not have a start method). The implementation style is replaced by a connector type statement. A connector type indicates the way a connector transmits signals. Types may be PC, RMIPC, CorbaPC, DataStream, etc. Connectors have additionally “connector end” statements. They define the number of connector ends and their types. The types of connector ends are the same as those for port types.

```
connector C1 {
```

---

<sup>2</sup> Java/A provides further means of communication than procedure call, so I speak of signals and signal handlers instead of invocations and methods.

```

connector_type PC;
ends PC, PC;
constructor C1(Port port1, Port port2) {
  connect(port1, port2); }
}

```

A configuration simply instantiates components and connectors, and plugs them together. In the configuration's start method is specified which component is to start processing:

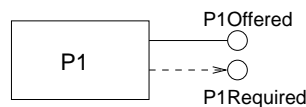
```

configuration Example forms Application {
  Component P1 = new P1();
  Component P2 = new P2();
  Connector C1 = new C1(P1.Pt1, P2.Pt2);
  void start() { P1.start(); }
}

```

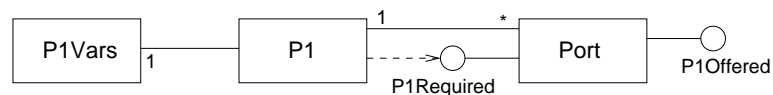
## 2.2 Java/A Mappings to Java

There exist a lot of ways to translate Java/A components to Java. In Figures 2 and 3 are two of them listed. Figure 2 shows the "simple class style": a component is mapped to a Java class that implements an offered interface.



**Fig. 2.** Simple Class Style

A more elegant way to map Java/A components to pure Java is the "complex class style" that is presented in Figure 3: in this mapping the architectural concept "port" is mapped to an extra class that implements the required and offered interfaces.



**Fig. 3.** Complex Class Style

### 2.3 Example Mapping

In the following I will present a translation of the Java/A example to traditional Java. The component *P1* from the Java/A example will be mapped to Java using the “Complex Class Style” (Fig. 3).

The main class is owner of the ports and has a reference to the component variables.

```
public class P1 {
    private Pt1 pt1 = new Pt1(this);
    private P1Vars v = new P1Vars();

    public P1() { v.y = 1; }
    public void start() { v.z = pt1.getX(); }
    public void setYImpl(int x) { v.y = x; }
    public Pt1 getP1() { return pt1; }
}
```

The component’s variables are collected in a very simple class that’s only purpose is to be a variable container:

```
class P1Vars {
    public P1Vars() { }
    public int y;
    public int z;
}
```

A port implements two interfaces: one interfaces that is externally visible, the offered interface, and the required interface that is internally visible. The port’s methods just delegate incoming signals to the referring handlers and outgoing signals to the connected connector:

```
public interface Pt1Off {
    public void setY(int x); }

public interface Pt1Req {
    public int getX(); }

public class Pt1 implements Pt1Off, Pt1Req {
    private P1 owner;
    private Pt2Off pEnd;

    public Pt1(P1 owner) { this.owner = owner; }
    public void connect(Pt2Off port) { pEnd = port; }
    public int getX() { return pEnd.getX(); }
    public void setY(int x) { owner.setYImpl(x); }
    public Pt1Off getP1Off() { return this; }
}
```

In this schematic example the implementation of the connector *C1* is not necessary. As both ports and the connector have the type PC, it is sufficient if both ports communicate directly.

### 3 Conclusion and Research Plan

The timetable for this thesis covers approximately three years. I plan to work further on mappings of Java/A into Java. Further I plan to develop tools to support Java/A: an IDE like editor and compilers for the above mentioned translations. Additionally I will address a mapping from an ADL (e. g. Wright) to Java/A. Furthermore I will enhance the Java/A editor with round trip engineering capabilities.

At the end of my thesis I hope to have bridged the gap between the architectural model of a software system and the concrete implementation of it. I will have given developers tools in their hands that support them in the implementation phase to ensure the consistency of their code with the underlying architecture and tools to help them maintaining software systems without damaging it.

### References

- [BCK98] Bass L., Clements P., Kazman R.: *Software Architectures in Practice*. SEI series in Software Engineering, Addison Wesley, 1998.
- [GS94] Garlan D., Shaw M.: *Characteristics of Higher-level Languages for Software Architecture*. Technical Report CMU-CS-94-210, Carnegie Mellon University, 1994.
- [GS96] Garlan D., Shaw M.: *Software Architecture, Perspectives of an Emerging Discipline*. Prentice Hall, 1996
- [Gar00] Garlan D., *Software Architecture: a Roadmap*. ICSE - Future of SE Track, pp. 91–101, 2000.
- [GAO95] Garlan D., Allen R., Ockerbloom J.: *Architectural Mismatch or Why It's Hard to Build Systems Out Of Existing Parts*. International Conference on Software Engineering, pp. 179–185, 1995
- [Kos01] Koskimies K.: *Towards architecture-oriented programming environments*. [www.cs.ualberta.ca/~kenw/conf/awsa2001/papers/koskimies.pdf](http://www.cs.ualberta.ca/~kenw/conf/awsa2001/papers/koskimies.pdf), 2001
- [Luc95] Luckham D., Kenney J., Augustin L., Vera J., Bryan D., Mann W.: *Specification and Analysis of System Architecture using Rapide*. IEEE Transactions on Software Engineering, 21(4):336–355, April 1995
- [Luc96] Luckham D.: *A Language and Toolset for Simulation of Distributed Systems by Partial Orderings of Events*. DIMACS Partial Order Methods Workshop IV, 1996
- [Med97] Medvidovic N., Taylor, R.: *A Framework for Classifying and Comparing Architecture Description Languages*. Proc 6th European Software Engineering Conference, Lecture Notes in Computer Science 1301, 1997, pages 60–76
- [PW92] Perry D., Wolf A.: *Foundations for the Study of Software Architecture*. ACM SIGSOFT, Software Engineering Notes, 17(4), pp. 40–52, 1992.
- [Sel01] Selic, B.: *Reflections on Software Architectures*. Lecture notes, Software Architecture Summer School, Turku, Finland 2001
- [Sto00] Störkle H.: *Models of Software Architecture*. Books on Demand, 2000