

RevJava – Design Critiques and Architectural Conformance Checking for Java Software

Gert Florijn
Software Engineering Research Centre – SERC
florijn@serc.nl

Introduction

Since the late 1980's the notion of software architecture has become increasingly popular. The (still) growing interest is a clear indication that there is an ongoing need for techniques to help manage and organize the development and adaptation of large, complex software systems.

There are many definitions of (software) architecture. Here we mention the one proposed by the IEEE P1471 group that developed the recommended practice for architectural description: “the fundamental organization of a system embodied in its components, their relationships to each other and to the environment and the principles guiding its design and evolution.”

When looking at this definition (and others), we see a couple of salient points:

- Architecture is needed to manage complexity. Therefore it should be used when the system to be built is sufficiently large and/or requires several people to construct it, or when several different systems are constructed that must interoperate partially or whose parts should be reusable.
- Architecture considers systems in their context. Software systems (like buildings and other constructs) should meet the constraints and demands of different stakeholders. It is the responsibility of architects to balance these concerns in a good solution.
- Architecture is about organizing the solution. It defines the key components of a system, outlines their responsibilities and shows how they are interrelated and (should) interoperate and communicate with the environment.
- Architecture guides the construction and evolution. It is not a “one-shot” diagram of the key parts of a system. It also outlines how the various parts of a system should be

designed and constructed. We want to ensure that all parts of the system(s) are built using the right rules and conventions such that they will fit in with the overall architecture and meet the concerns considered in it.

Conformance

A crucial aspect of software development under architecture is *architectural conformance*. Once we have defined an architecture for a system (or a family of systems) we want to ensure that the actual software (the detailed design and code) conforms to the architecture. For example, we want to ensure that all components conform to the behavioral responsibilities that were assigned to them in the architecture, but also to naming, usage and other guidelines defined in the architecture. If conformance is not achieved, i.e. if the code does not behave or is not organized according to the architecture, the architecture is useless and the system will not behave in a way foreseen in the architecture.

To illustrate this point, consider a very simple example of the Observer pattern used to distribute/react to state changes. One of the “rules” that the observed component (the *subject* or *observable*) should adhere to is that any relevant change to its state leads to the calling of a change notification method/function that will propagate an event to the interested “observers”. If the subject doesn’t meet this contract, observers will not be informed and consistency in the overall system may be lost.

In practice, conformance is not easy to achieve. In large development activities, a lot of people have to understand the architecture and build software according to it. Experience learns that just publishing architecture documents is not enough. People have to know about it, understand it, appreciate it, and work accordingly to it. In other words, the architecture has to be marketed and its use must be managed.

But even if the architecture is used, it may not be used properly. If developers do not really understand the ideas behind the architecture, they will produce “wrong” code, i.e. software that does not do what it should do or is not organized according to the rules. A typical example is that people use libraries that should not be used for a particular component or copy an existing piece of code that doesn’t match the precise same role as the target component.

An additional issue is that the architecture itself typically evolves during development and maintenance. During detailed design and coding we often encounter unanticipated problems (e.g. performance or scalability issues). Under time constraints, these problems are often fixed by work-arounds instead of adapting the architecture and reorganizing all parts of the system. A related problem occurs during maintenance, where software is often adapted by people that did not build the system itself. Again, the risk occurs that the actual software will diverge from the architecture it was set up with.

An architectural description for a piece of software that does not conform to it is fairly (or should one say: completely) useless. The key problem is that we cannot trust the architecture

description anymore to make decisions about changes or additions. For example, we cannot rely on the architecture description to determine the impact of certain changes. Instead, we have to look into the design or, in most cases, into the code to find out how a particular change should be done and what is affected.

Obtaining conformance

So, how do we obtain conformance between an architecture and the actual software. Ideally, we would like to have (extensible) “programming languages” in which architectural principles and constructs (layers, contracts for patterns, etc) can be expressed as first class citizens. In this way we would obtain direct compliance since the architecture definition is just the initial version of the program. While there are several Architectural Description Languages around they generally do not (aim to) cover implementation too.

So, we have to make do with existing programming languages and must see how we can check the architecture rules for programs written in them.

One interesting approach is using frameworks. A framework provides the core functions of (families of) applications. By adding in the application specific functionality at specific points and through specific mechanisms (e.g. inheritance in an OO language) we tailor the framework to our needs. Since the framework has the control, provides the core behaviour and encodes (some of) the architecture’s key concepts, the degrees of freedom for a developer are limited and it becomes more difficult to write software that doesn’t conform to it. The framework approach can be taken one step further by introducing a (domain) specific language in which the tailoring for a particular system is described. A generator then produces the right code (e.g. plug-ins for a framework) for the final application. Since the degrees of freedom are further limited, the resulting system will typically conform very good to the architecture.

Designing frameworks/generators is not trivial however. It requires a very good understanding of the problem domain and of the characteristics of the system(s) to be built. Often, multiple attempts are needed to create a framework. This implies that it not always useful or economically viable to use this approach.

If we cannot use these approaches, we have to introduce steps in the development process that help achieve conformance. Basically it means that software must be reviewed (and reorganized) repeatedly to enforce conformance. An example of this approach can be found in some of the practices in Extreme Programming, such as “pair programming” for continuous reviewing and “merciless refactoring”. The basic idea is that software is continuously restructured and reorganized to keep it simple – and possibly also to let it conform to the architecture it should meet.

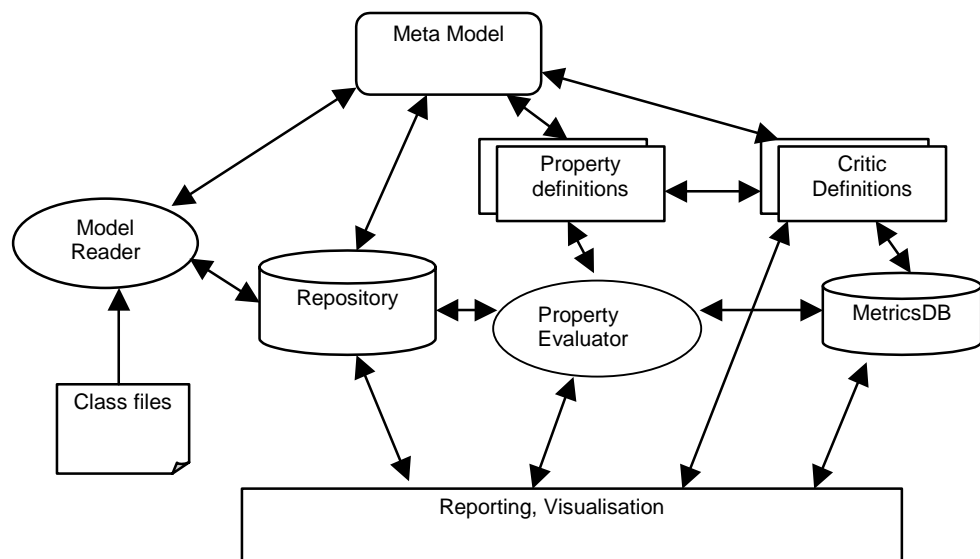
Design or code reviews can be tedious and time-consuming, especially with large teams. Therefore it is interesting to see whether such reviews can be supported by tools. Basically,

the idea is to “code” the architectural rules into a program and then let this program check whether designs or code violate these rules.

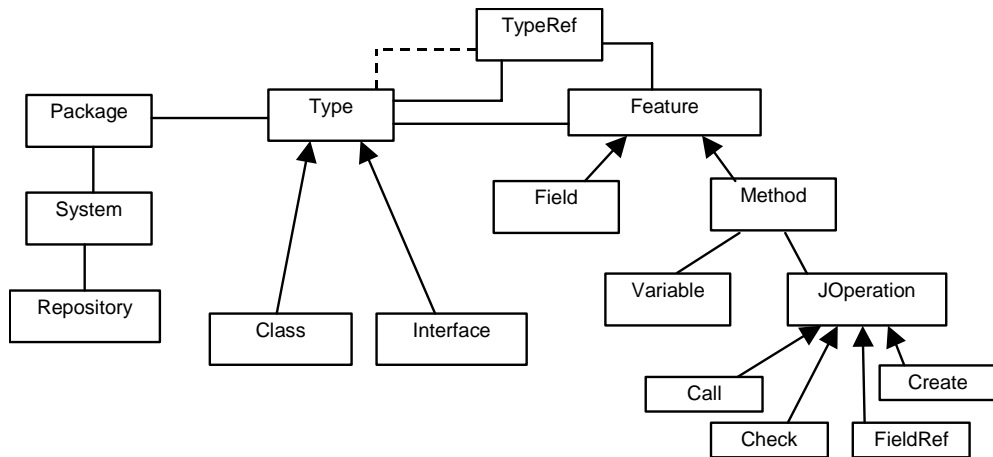
RevJava - Supporting reviews and conformance checks

Over the past year we have developed and experimented with such a review support tool. Our ambition in these experiments was not to fully automate architectural conformance checking. Instead, we wanted to create an assistant that can be parametrized easily with (local) architectural rules and checks existing software for (potential) violations of these rules. Phrased differently, we want a tool that highlights “bad smells” in software so that a developer, reviewer or architect can fix problems as they arise.

The tool we developed is called RevJava and it is used to analyze and critique object oriented software. While the design is fairly generic, the current implementation operates on compiled Java code (i.e. Java class files).



The figure above highlights the main components of RevJava. A reader reads in the program (or design model) and stores it in a repository. This repository is organized according to a meta model which defines all relevant entities in an OO/Java program such as “Package”, “Class”, “Interface”, “Method” and “Field”. Since we do a partial analysis of the code in methods, the meta model (see below) also contains entities for method calls, class instantiation and the-like. The model also defines all basic relationships among these entities, e.g. that a method is defined in a class, or that a method refers to a field in a particular class. Note that while the current implementation is geared towards Java programs, the meta-model we use is very similar to the Famix model and appears general enough to capture other (OO) languages.



For each meta-model type we can define and plug-in derived “properties” that calculate particular information about a model element. For example, for “class” elements we can define a property “all subclasses” that produces a transitive closure over the built-in “subclass” relation defined between classes. If this definition is loaded into RevJava, the property can be obtained (on demand) for all program elements of type “class”.

Some of the properties of a loaded program are treated as “metrics” (e.g. average inheritance depth, response for a class) and can be collected in a database. In this way we can collect averages of certain properties (e.g. inheritance depth) over larger bodies of software and compare the program under consideration to these averages.

The final and most appealing part of the tool are the critics. A critic checks whether a particular model element violates a particular design rule. Critics can be plugged in easily and can use derived properties to obtain the information needed to do the check. Critics are inspired by similar ideas in, among others, ArgoUML – an open source UML case tool (see www.argouml.org).

The information collected in the repository and defined by the properties and critics can (obviously) be exploited in different kinds of reporting and visualisation tools. We have been (and still are) experimenting with such tools. Besides stand-alone “critic” and “metric” reporters (see screenshots below), we have created visualisations that highlight particular critic violations in large collections of classes.

By deriving more information from the basic relations stored in the repository we can also highlight more high-level problems. For example, we can derive the “usage” relationship among packages and highlight “layer” breaking or cyclic dependencies. In a similar vein we can (try to) derive a higher-level insight into the program structure. For example, we have experimented with “pattern detection”, where we use the information in the repository to find potential occurrences of patterns and try to highlight errors or wrong usages, e.g. a singleton class that is not fully implemented as a singleton. Our earlier example of the responsibilities imposed on a Subject in an Observer contract can (within limits) also be checked.

Of course, we can also use the RevJava tool to handle situational rules. By defining and plugging in the right properties and critics we can check naming conventions, calling conventions, behavioral responsibilities (does a class override a specific method and in that method does a call to a particular method in another class), usage restrictions (package X cannot use package Y). In this way, we can check whether the actual software conforms (to some extent) to an architecture. Given the fact that the tool is fast and easy to use, checking software for conformance can become a routine part of the development cycle.

Acknowledgements/Related work

RevJava was/is developed by the author. Various people gave valuable input, such as Ronald Haentjens Dekker, Erik Aalbrecht, Willem van den Ende and Marc Evers. Of course this work was and is inspired by efforts of others. Roel Wuijts (together with others) at the Free University of Brussels works on SOUL, which explores architectural conformance checking of Smalltalk software. John Brant and others (University of Illinois, Urbana Champaign) worked on Smalltalk Lint, a style checker for Smalltalk. Boris Bokowsky (Free University of Berlin) worked on source level-checking of Java software (Barat).