Towards Architectural Information in Implementation (NIER Track)

Henrik Bærbak Christensen
Deparment of Computer Science
Aarhus University
Aarhus, Denmark
hbc@cs.au.dk

Klaus Marius Hansen
Department of Computer Science
University of Copenhagen
Copenhagen, Denmark
klausmh@diku.dk

ABSTRACT

Agile development methods favor speed and feature producing iterations. Software architecture, on the other hand, is ripe with techniques that are slow and not oriented directly towards implementation of costumers' needs. Thus, there is a major challenge in retaining architectural information in a fast-faced agile project. We propose to embed as much architectural information as possible in the central artefact of the agile universe, *the code*. We argue that thereby valuable architectural information is retained for (automatic) documentation, validation, and further analysis, based on a relatively small investment of effort. We outline some preliminary examples of architectural annotations in Java and Python and their applicability in practice.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures

General Terms

Design, Documentation

Keywords

Software architecture, agile methods, architectural reconstruction

1. INTRODUCTION

Agile methods have over a relatively short period had an enourmous impact on modern software development. As evident in the agile manifesto, emphasis is put on *Working software over comprehesive documentation* [3] which means the teams are highly focused on producing and delivering code that add business related features in each iteration. The *Clean code that works* mantra of Extreme Programming [4] highlights this dedication. Thus, it seems that architectural analyses, design, evaluation, and documentation may suffer severely in agile projects. Recently, arguments have been made that the apparent dichotomy is a false one [1], but still there is a tension between the traditional architecture design and documentation up-front idea on one hand, and the speed and incremental architecture building idea on the other.

Nevertheless architectural analysis, overview, and evaluation are hinged on architectural documentation on the right level of abstraction and with the proper viewpoint for the problem at hand. Deployment, dynamic, and static views of an architecture are vital tools of the architects for understanding and reasoning. Given the agile methods high focus on creating code over documentation the chances are high that architecture only exists in the quickly fading memory of programming pairs and forgotten architectural spikes. As noted by Woods [14]: *"Recovery of architecture from code inevitably struggles because the information is missing, not hidden..."* and thus the produced code is of little help.

Our contribution is an early proposal for using modern programming language annotations to embed architectural information directly into the implementation. While using annotations for embedding architectural information is arguably not a "new idea" (indeed it was proposed by Woods at ECSA 2010 and a variaty of ideas have previously been reported (see Section 4)) the novelty of our contribution is a perspective "from the code towards the architecture" whereas earlier work have had the opposite perspective, typically a strong ADL or model-driven approach that leaves traces in the code. Our perspective is to let developers and designers augment their code with the architectural constructs "as it happens" and is thus an inherently agile perspective. Working code is the important asset and we strive to get the most and best architectural information under this premise. We argue that a open-ended approach with little overhead has a better chance of being adopted in agile development while the amount of architectural information available is significantly increased.

2. EXAMPLES

Patterns. Architectural as well as design patterns have shown their strength in practical development as means to communicate and implement sound solutions to recurring problems. At the structural level, both architectural and design patterns dictate certain roles and protocols to software entities, like for instance the three roles of *model*, *view*, and *controller* and the connecting *observer protocol* of the MVC pattern. However, these roles and thus patterns are difficult to identify once they are implemented if no further clues are provided. As an example, most of the classic 23

ICSE '11, May 21–28, 2011, Waikiki, Honolulu, HI, USA

GoF [9] design patterns are structurally similar, for instance the class structure of Strategy and State are nearly identical though they are solutions to rather different problems, and therefore reverse engineering can only hint that some pattern may be present but never discern which of the two (or indeed many other patterns with similar structure) it is. Only the designer of the particular software knows.

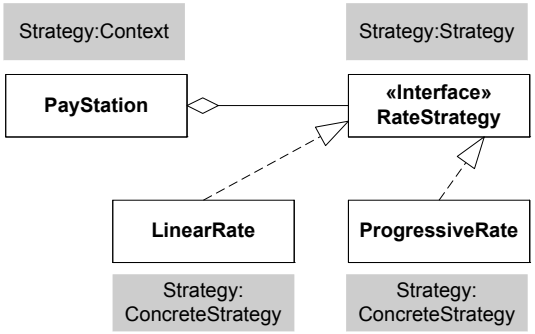

Figure 1: A Strategy based design with pattern roles shown.

So, why not let the developer state the use of architectural and design patterns directly in the code by stating the pattern and the roles? As an example, consider a parking lot pay station that has to be configured for several municipalities' different pricing policies. The architect decides to use the Strategy pattern resulting in the design shown in Figure 1. In the figure, the three roles of the Strategy pattern (context, strategy, concrete strategy) as defined by the GoF book are marked in gray. To preserve these architectural intentions, the developer can annotate classes and interfaces directly in the source code. In Java, this could look like shown in Figure 2. The @Pattern annotation associates a type definition and declares this type to belong to a certain role (role) of a certain pattern (name) and defines its membership of a certain group (id). The latter is of course important as usually there will be many applications of Strategy in a system and as one type can participate in many patterns. Thus the annotated code fragment above is essentially an encoding of the mini architecture in Figure 1.

```
@Pattern( name = "Strategy",
          role = "Context",
          id   = "PriceCalculation" )
public class PayStation { ... }

@Pattern( name = "Strategy",
          role = "Strategy",
          id   = "PriceCalculation" )
public interface RateStrategy { ... }

@Pattern( name = "Strategy",
          role = "ConcreteStrategy",
          id   = "PriceCalculation" )
public class LinearRateStrategy
  implements RateStrategy { ... }
```

Figure 2: Encoding the architecture as annotations.

Often a given type participates in several design patterns at the same time. This is in particular true in object-oriented frameworks. However, Java annotations do not support multiple instances of the same annotation being associated with the same type, but @Pattern annotations can easily be wrapped in a list, like in Figure 3 in which the PayStation type now also plays the Subject role of the Observer pattern. Also methods that are central for a pattern's protocol can be annotated, like method notifyObservers() is marked as the notify method of the Subject. This source code can be processed by Java annotation processors which can validate correctness of the use of the patterns as well as document it. Figure 4 shows a class diagram generated with yUML[1] with emphasis on the PayStation class. Pattern names and roles are added as keywords in the UML syntax as yUML is quite restrictive in what can be put into the class box. While inferior in graphical quality to the hand-drawn Figure 1—it is certainly quicker to make!

```
@Patterns( {@Pattern( name = "Observer",
                      role = "Subject",
                      id="PaystationObserver" ),
            @Pattern( name = "Strategy",
                      role = "Context",
                      id="PriceCalculation" )})
public class PayStation { ... }
  @PatternMethod( type = "notify" )
  public void notifyMyObservers() { ... }
}
```

Figure 3: Multiple pattern annotation.

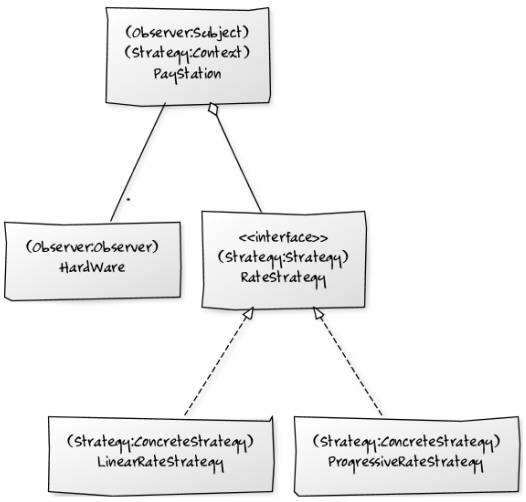

Figure 4: Generated yUML.

Heterogeneous Architectures. Software systems are (most) often written using different programming languages and technologies. An example could be an Android application (written in Java) connected to a REST-based [8] backend (written in Python). This poses problems for architectural description and for architectural reconstruction in particular. One problem is that components are typically not statically linked, but rather communicate via, e.g., TCP/IP sockets.

Assuming that the languages/technologies used support annotations in some form, we may annotate components in

[1]http://yuml.me/

order to be able to reconstruct architecture.

In the following listing, a Python Backend component references a Frontend component not written in Python:

```
@Component(references=["Frontend"])
class Backend():
    ...
```

The Frontend component is defined in a Java class. The Frontend component furthermore composes a Controller component:

```
@Component(external_references={"Backend"},
           composes=Controller.class)
public class Frontend extends Activity {
    ...
}
```

Processing this, yields the following RIGI [13] output:

```
Component Controller
Component Frontend
Composes Frontend Controller
HasPort Controller ControllerUpdate
Component __main__.Frontend
References __main__.Frontend Backend
```

Given the RIGI output, we may apply a Relational Markup Language (RML; [5]) script which results in the output in Figure 5.

```
PRINT "YUML http://yuml.me/diagram/class/";
FOR component1 IN References(x,_) {
  FOR component2 IN References(component1,x) {
    PRINT "[<<Component>>\n", component1,
      "]->[<<Component>>\n", component2, "],";
  }
}
FOR component1 IN Composes(x,_) {
  FOR component2 IN Composes(component1,x) {
    PRINT "[<<Component>>\n", component1,
      "]++->[<<Component>>\n", component2, "],";
  }
}
PRINT "", ENDL;
```

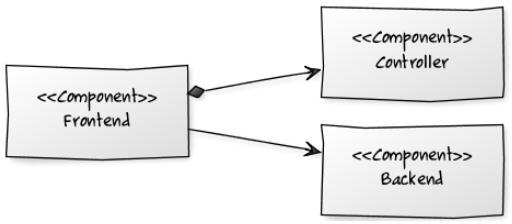

Figure 5: A UML view of components.

Similarly, we can do a typecheck of the annotated components (and connectors), checking, e.g., if no attribute in a component has a non-referenced/non-composed Component as type:

```
TypeOf(c,f,t) := HasField(c,f)
  & FieldHasType(f,t);
IntegrityError(c,f,t) := TypeOf(c,f,t)
  & Component(t)
  & !Composes(c,t)
  & !References(c,t);
PRINT ["HasIntegrityError"] IntegrityError(c,f,t);
```

3. DISCUSSION

What can architecture annotated source code provide that existing techniques, like ADLs, model-driven architecture, and so forth, can not? While this is a valid question, it is actually not the relevant question. The relevant question is, in our opinion, whether the "return on investment" is worthwhile in an agile context where architctural design is made fast and inspired by the daily and weekly feedback that the code-oriented development process provides. In such an environment, the success lies in how easy it is to express the architectural design. When the mantras are speed and simplicity the notion of starting an ADL modeling tool seems contradictory to most real-world development teams. However, it does seem feasible that once the pair programming team decides "let us use a Publish-Subscribe pattern here", "a layered architecture will increase modifiability", or "a ping/echo tactic is the proper choice to increase availability", they state this decision right in the code by annotating the involved types and methods. It is fast, and even without any further tool support, it provides important architectural information to other teams and developers that may pass by these code fragments.

So, returning to the first question: "what can it provide that other techniques cannot" the answer is theoretically "nothing", but its low cost may just be the important aspect that makes it cheap enough to actually get that architeture information that today only exists in the minds of the developers in many agile development projects today.

Many proposed techniques in software architecture research and pratice have a unspoken premise of architecture being the first-class citizen and the code second-class. Implementation is generated from ADLs or models are decorated with code fragments in more or less cumbersome ways. However, in most real world projects the code is the first-class citizen because, as Bertrand Meyer stated in the 1980's *"once everything is said, software is defined by code."* Working software pays the bills. Our suggestion is to keep the code as the first-class citizen and then get as much architecture information as possible under this premise.

Once the architecture information is embedded, we can point to a number of benefits:

- *Architecture documentation in code.* Tactics, styles, and patterns are often documented anyway in JavaDoc or similar comments. Using a community-accepted format would just move unstructured documentation into a structured and better supported format rather than pose an additional investement in time. Thus even without supporting tools, it would be an improvement in communication across teams.
- *Generation of views.* Visual notations such as various UML diagrams or ADL-based descriptions could be generated. For instance, a class diagram with associated pattern roles like Figure 1 can be generated and embedded into JavaDoc web pages or similar.
- *Validation of well-formedness.* A tool can verify that the patterns are well formed, style topology is maintained, etc. For instance, a tool can validate that all three roles of the Strategy (context, strategy, concrete strategy) are associated with abstractions, that

no more than one strategy role is defined for a pattern with a given identity, etc.

- *External tools.* The annotations can be processed to generate e.g. RIGI data and thus a system's architecture can be processed by a number of analysis tools.
- *Run-time verification.* Main stream languages like Java and C# have the ability to retain the annotations at run-time. This can be employed to build tools to verify architectural protocols and other run-time aspects.

Of course, the approach has limitations. Some architecturally important information does not seem like candidates for embedding in source code. For instance architectural requirements or quality attribute scenarios are cross-cutting concerns that do not associate well to language constructs like types, packages, or methods.

4. RELATED WORK

Krahn and Rumpe [10] proposed to support architectural refactoring through source code annotation in Java 5. Their annotation were related to structure (ports, roles, parts) and construction (adding and removing structure), their goal being to synchronize external architectural descriptions and source code. While their work shared motivation with us (supporting keeping architectural information up-to-date during development/evolution), there appear to be no realization of their proposal.

Like Krahn and Rumpe, Mens et al. [11] recently focused on "architectural co-evolution", i.e., that software implementation, design, and, architecture should evolve together when one artefact/description evolves. Mens et al. particularly pointed to that preserving critical behavioral properties (e.g., specified by behavioral views on the architecture). While they focus on model-driven engineering in contrast to us, an interesting area of future work is likewise to which extent we can support reconstruction of behavioral views by annotations and thus use that information in evolution.

Various related work follow the strategy of using a model-driven approach to support architectural conformance. Cirilo et al. [7], e.g., instatiate product lines based on Java annotations while Buchgeher and Weinreich [6] use Service Component Architecture[2]-related annotations in connection with architecture models in an Eclipse-based environment.

Finally, various approaches have created language extensions to support architectural conformance. ArchJava [2] is a Java-based example of a language extension supporting embedded architecture definition and conformance checking. Ubayashi et al. [12] proposed "archfaces" as an intersection of design and implementation with which a collaboration of components can be described and connected to implementation via an aspect-oriented mechanism. In contrast to this work, we propose an approach in which existing source code can be annotated with architectural information.

5. SUMMARY AND STATUS

So far our work is in a preliminary state. Experimental annotation processers have been developed for Java and Python. The Java version can validate well-formness for a small subset of design patterns and can moreover generate architecture diagrams using yUML for documenting design pattern usage directly in JavaDoc pages and component/-connector relationships. Also a processor has been developed that generates RIGI relations. Furthermore, we have designed and realized annotations also for Python, allowing architecture documentation to be reconstructed referencing Python components from Java components and vice versa.

In summary, we tentatively argued that embedding architectural information in source code through annotations could with relatively little effort support architectural activities in agile software development. While we have provided examples of the usefulness of such an approach, it remains future work to establish this fully.

[2]http://www.osoa.org

6. REFERENCES

[1] P. Abrahamsson, M. A. Babar, and P. Kruchten. Agility and Architecture: Can They Coexist? *IEEE Software*, March/April 2010.

[2] J. Aldrich, C. Chambers, and D. Notkin. ArchJava: connecting software architecture to implementation. In *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*, pages 187–197. IEEE, 2005.

[3] Agile manifesto. www.agilemanifesto.org, 2001.

[4] K. Beck. *Extreme Programming Explained–Embrace Change, 2nd ed.* Addison-Wesley, 2005.

[5] D. Beyer, A. Noack, and C. Lewerentz. Efficient relational calculation for software analysis. *IEEE Transactions on Software Engineering*, 31(2):137–149, 2005.

[6] G. Buchgeher and R. Weinreich. Connecting architecture and implementation. In *On the Move to Meaningful Internet Systems: OTM 2009 Workshops*, pages 316–326. Springer, 2009.

[7] E. Cirilo, U. Kulesza, and C. Lucena. A product derivation tool based on model-driven techniques and annotations. *Journal of Universal Computer Science*, 14(8):1344–1367, 2008.

[8] R. Fielding and R. Taylor. Principled design of the modern Web architecture. *ACM Transactions on Internet Technology (TOIT)*, 2(2):115–150, 2002.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[10] H. Krahn and B. Rumpe. Towards Enabling Architectural Refactorings through Source Code Annotations. In *Proceedings der Modellierung*, volume 22, page 24, 2006.

[11] T. Mens, J. Magee, and B. Rumpe. Evolving software architecture descriptions of critical systems. *Computer*, 43(5):42–48, 2010.

[12] N. Ubayashi, J. Nomura, and T. Tamai. Archface: a contract place where architectural design and code meet together. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 75–84. ACM, 2010.

[13] K. Wong. Rigi user's manual, version 5.4.4, 1998.

[14] E. Woods and N. Rozanski. Unifying software architecture with its implementation. In *ECSA '10: Proceedings of the Fourth European Conference on Software Architecture*, pages 55–58, New York, NY, USA, 2010. ACM.