Architecturally Evident Applications – How to Bridge the Model-Code Gap?

Oliver Drotbohm (odrotbohm@vmware.com) in collaboration with
Henning Schwentner (henning.schwentner@wps.com) & Stephan Pirnbaum (stephan.pirnbaum@buschmais.de)
February 2022

Abstract

Over the course of its lifetime, every non-trivial piece of software will significantly grow in complexity. The extent of that growth significantly affects the ability to evolve it to avoid having to replace it with a costly rewrite eventually. Thus, managing that complexity has been the topic of interest in the software community, and architectural and design pattern languages have been identified as a means to achieve that. But even if the conceptual models of an application use that language, a fundamental challenge remains: how to express those abstract concepts in the actual codebase?

This paper explores a novel approach that enables developers to explicitly express architectural and design concepts in application code, which ultimately enables:

- *Understandability* – By finding the architectural language in code, it is easier for developers to understand the code base, relate individual elements of it to the bigger picture and, ultimately, make architecture more accessible.
- *Documentation* – With abstract concepts present in the code base, we can extract documentation about it that is correct by definition and describes it at an architectural abstraction level.
- *Verification* – We can verify that our implementation adheres to the rules associated with the concepts that the individual elements of the code base implement at different levels of architectural abstraction.
- *Reduction of boilerplate code* – At the application boundaries, domain model elements have to be persisted into some data store or exposed to clients by APIs. Architectural concepts, directly expressed in those elements, allow transparently defaulting such mappings into popular implementation technologies.

This paper presents the fundamental idea in detail, as well as a Java library to express architectural and design concepts, and contrasts it to alternative approaches. It concludes with a presentation of the support of that library in a variety of associated integration technologies to implement the aspects described above.

Introduction

Bridging the gap between architectural patterns and code bases has been an ongoing challenge when writing long-living business software. We would like to present an approach to express these patterns directly in code by using programming-language-specific means and describe how that approach becomes an enabler to create code that is more expressive, more understandable, more correct and ultimately easier to change. The paper uses Java as an example because it is a very ubiquitous language in enterprise applications. However, the approach can be transferred to other languages, too.

Over the last 1.5 years, a prototypical implementation has been implemented in a cross-company collaboration effort between VMware, WPS Solutions (Hamburg), and BUSCHMAIS (Dresden). It can be found under a project named jMolecules on GitHub [jmolecules].

Fundamentally, we need a mechanism to express architectural artifacts in the codebase. In Java, two primary language constructs are great candidates to achieve this: annotations and types. We will have a detailed look at the pros and cons of each later. jMolecules currently provides annotations for the following architectural concepts: the Domain-Driven Design (DDD) building blocks described in [evans03], events and event listeners, and the parts of particular architectural styles, such as onion architecture [palermo08], layered architecture, and CQRS systems. The DDD and event concepts are also available as Java interfaces alternatively.

Developers can refer to the concept library in their application build files so that the architectural definitions become an inherent part of the code base. This results in more expressive code that has a more direct connection to the architectural model in the first place and, thus, supports understanding the implementation. The metadata available within the code enables extensive integration with external technology to verify the implementation against the model expressed in the code and extract architecture and developer documentation. To run on ubiquitous technical platforms (such as Spring Framework [spring]) and integrate seamlessly with persistence technology (such as the Jakarta Persistence API (JPA) [jpa] or commonly used serialization APIs like Jackson [jackson]), domain code usually has to be augmented with boilerplate code, like annotations or additional models which significantly increases the accidental complexity of applications.

The architectural concepts expressed explicitly, we can also derive and generate that additional code needed in reusable libraries. It can be directly implemented in either jMolecules or by the target technologies. The integration with verification tooling allows for detecting modeling and implementation problems early. The boilerplate code generation even prevents errors that would potentially be introduced in the step of projecting the architectural pattern into application code in the first place.

Motivation

Unlike prototypes, non-trivial business software will live over a long time. Thus, development teams spend the bigger part of their development work changing the software to accommodate changing requirements. This means that evolvability of software is crucial, and a prerequisite of that is that it is easy for developers to understand the system and its code. Both software architecture and design techniques play a key role in implementing these requirements.

Pattern languages, such as the Domain-Driven Design building blocks, have been established to describe concepts and rules assigned to those to manage complexity. While these patterns have been in successful use for a while, a fundamental challenge exists in the gap between the architectural design and the actual implementation code. A gap that causes challenges when developers need to modify the software.

The individual architectural concepts cannot be immediately found in the code base, especially if the logical concept is reflected in multiple source code artifacts. For example, a DDD *Aggregate* usually consists of a root entity type (plus 0 to n other entities and value objects) and a corresponding repository interface. Thus, solely looking at the root entity type does not let the developer know whether it is an aggregate root or not, making it difficult to distinguish important model elements from supportive ones.

Implicitly scattering architectural concepts around the codebase introduces the risk of the implementation not following the rules implied by the concepts, which can cause significant complexity and effort if detected too late. Traditionally, developers have resorted to either naming conventions or references to platform-specific elements in the code base in combination with custom extension and configuration of architecture verification tooling to mitigate that risk. This comes at the expense of significant extra work, which often causes those verification approaches not to be applied for short-term convenience.

Related Work

Using pattern languages in software architecture and the influence of that on the maintainability of systems has been extensively discussed in [lilienthal17]. Lilienthal identifies the use of modularity, hierarchization, and patterns as variants of *chunking*, a concept established in cognitive psychology to help the brain deal with complexity, in turn described in detail in [hermans21].

The model-code gap was first discussed in detail in [fairbanks12, p. 169ff]. Fairbanks hints at the fact that architectural models primarily consist of *intensional* model elements (see [eden-kazman] for details), which are particularly hard to transfer into code. In contrast to *extensional* elements, to which source code elements can easily correspond (a logical purchase order module being represented by a build unit named *purchase order*), intensional ones, such as the DDD building blocks pattern language, require implementation conformance to the rules implied by the concepts. This has been traditionally achieved by using naming conventions accommodated by tooling that developers had to customize themselves to the conventions used.

A prepared architectural vocabulary as proposed in this paper applicable to the codebase lets tooling integration be prepared and used by application developers without additional effort and, thus, reduces the cognitive load imposed on them.

ArchJava

In [archjava], Jonathan Aldrich describes a language extension to express the architectural concept of a component within Java. It would need a dedicated compiler or a pre-compilation step for the original Java compiler to work, as well as explicit support in IDEs, such as IntelliJ IDEA, Eclipse and VS Code. This presents a significant obstacle in widespread adoption and is likely one reason the approach never really gained significant traction.

jMolecules avoids those challenges by using the Java language as is and, instead, adds the semantic elements by using existing language means, such as annotations, types, and generics.

Model-Driven Architecture

In the early 2000s, Model-Driven Architecture MDA [mda] has tried to connect architectural descriptions and implementation. It was driven by the Object Management Group (OMG) and centered around UML and dedicated tooling to transform models into executable code. Just as ArchJava, it is not widely applied these days as it doesn't align well with developer's everyday workflows.

Stereotype Annotations in Spring Framework

Architectural stereotypization of application code has also been the subject of the annotation-based component model of Spring Framework [spring]. The generic *@Component* annotation is extended by (for example) *@Controller*, @Service, and *@Repository*. Those annotations serve a descriptive purpose but also act as anchors to apply technical functionality to the classes to which they are applied. For example, classes annotated with *@Repository* are subject to Spring's exception translation, which converts persistence technology-specific exceptions into Spring's common *DataAccessException* type hierarchy and, thus, lets client code abstract from the technical detail of the persistence technology used.

The downside of that approach is that using these annotations effectively creates a dependency on Spring Framework. While annotations constitute a very weak kind of dependency (they are not even required during the compilation of code that uses them), it might be desirable to avoid it in the first place, especially in domain code. Also, the set of annotations available is completely driven by the framework's needs and does not aspire to cover architectural pattern languages in their entirety.

Design

We will showcase the effect of working with architecturally evident code based on an example in the e-commerce domain. We model an *Order* DDD *Aggregate Root*. It consists of *LineItem Entities*. The *Order* refers to a *Customer* which is an *Aggregate Root* in turn. The assignment of the stereotypes of the pattern language lets us understand that the *Order* applies business rules and constraints to the list of *LineItem*s.

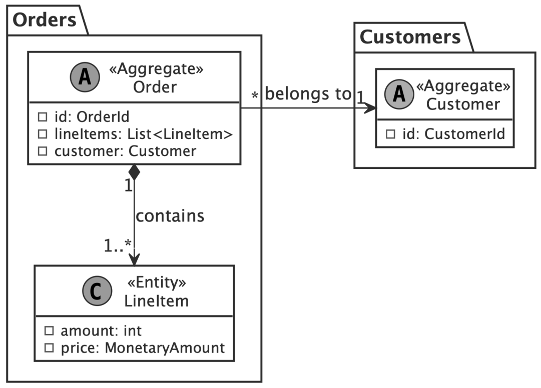


Fig.1 – An (invalid) Order model

Expressing Architectural Patterns in Code

jMolecules allows transferring the patterns expressed in the design into the code. To do that, the application developer would depend on the JARs that correspond to the architectural vocabulary they would like to use. The DDD one can be referred to as follows:

```
<dependency>
  <groupId>org.jmolecules</groupId>
  <artifactId>jmolecules-ddd</artifactId>
  <version>…</version>
</dependency>
```

Declaring the jMolecules DDD module as a project dependency

The referenced JAR contains both annotations and interfaces to assign DDD vocabulary to individual source code elements. The annotation-based flavor looks something like the following code:

```
// All annotations from org.jmolecules.ddd.annotation

@AggregateRoot
class Order { … }

@Entity
class LineItem { … }

@AggregateRoot
class Customer { … }
```

The architectural concepts via jMolecules' DDD annotations

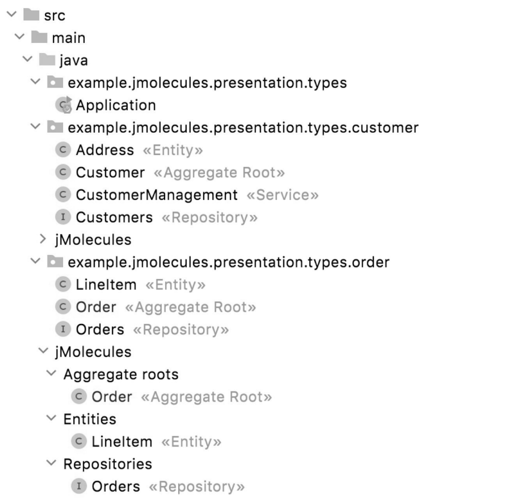
Alternatively, the interface-based variant allows declaring even more details about the intended relationships:

```
// All super types from org.jmolecules.ddd.types

class Order
  implements AggregateRoot<Order, OrderId> { … }

class LineItem
  implements Entity<Order, LineItemId> { … }

class Customer
  implements AggregateRoot<Customer, CustomerId> { … }
```

The architectural concepts by using jMolecules' DDD interfaces

Not only do we assign stereotypes to our domain model implementation, we also declare that (for example) the *LineItem* logically belongs to the *Order* aggregate through the first generic type parameter of *Entity*. We do not have to inspect the *Order* class to understand that. In other words, individual model elements develop some contextual gravity, so that reasoning about them becomes significantly easier as the amount of source code elements that must be understood to capture the context is reduced.

That architectural metadata can be introspected, as the *jMolecules* IntelliJ IDEA prototype shows below. It extracts the classes' roles and decorates the project overview tree accordingly. It can even provide a dedicated tree node to group model elements per stereotype.



```
src
  main
    java
      example.jmolecules.presentation.types
        Application
      example.jmolecules.presentation.types.customer
        Address «Entity»
        Customer «Aggregate Root»
        CustomerManagement «Service»
        Customers «Repository»
      jMolecules
      example.jmolecules.presentation.types.order
        LineItem «Entity»
        Order «Aggregate Root»
        Orders «Repository»
      jMolecules
        Aggregate roots
          Order «Aggregate Root»
        Entities
          LineItem «Entity»
        Repositories
          Orders «Repository»
```

Fig.2 – IntelliJ IDEA using the architectural metadata present in the source code to distinguish model classes

Verifying Implementation Against Architectural Definitions

Expressing architectural concepts directly within code also lets us reason about whether the model satisfies the rules implied by the concepts expressed as stereotypes. In our particular case, according to the relationship rules described in [vernon13, p. 359ff], the reference from the *Order* to the *Customer* needs to be modeled as an identifier reference, not the fully mapped aggregate.

We can use architectural tools like ArchUnit [archunit] or jQAssistant [jqassistant] to verify the arrangement based on the additional metadata present in the code base and even the compiled bytecode. For the former, an API exists in the jMolecules Integrations artifact (*org.jmolecules.integrations: jmolecules-archunit*) that allows verifying the domain model structure by using a simple test case:

```
@AnalyzeClasses(packagesOf = …)
class ArchVerificationTests {

  @ArchTest
  void verifyModel(JavaClasses classes) {
    JMoleculesDddRules.all().check(classes);
  }
}
```

The architectural concepts by using jMolecules' DDD interfaces

When run during the build or in the IDE, this test would fail on a 1:1 implementation of the model shown in Figure 1, because that declared the relationship to the *Customer* aggregate as a to-object reference.

Extracting Architecturally Relevant Documentation

The architectural metadata present in the code lets tooling inspect it and distinguish source code artifacts that represent architecturally relevant concepts from ones that are implementation details and, thus, lift that kind of encapsulation into documentation artifacts. Figure 3 shows a UML component diagram derived from the codebase that uses *jMolecules' @Module* annotation to describe three logical modules.

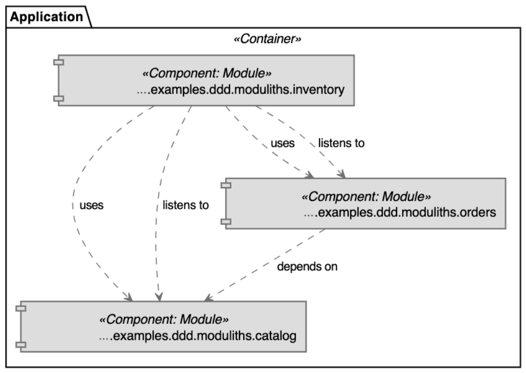
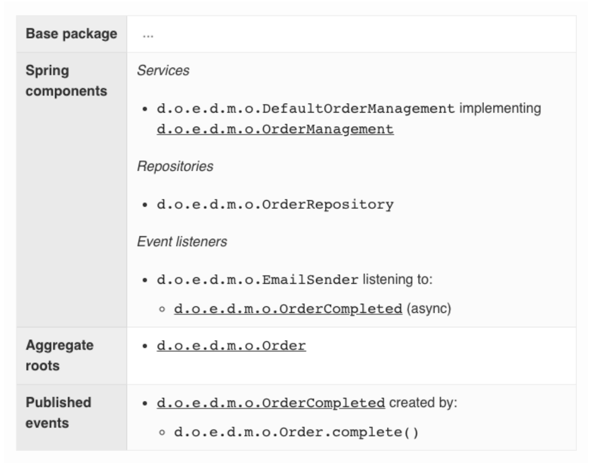


Fig.3 – A UML component diagram derived from the architectural metadata present in the code base

The documentation integration can now further inspect the code contained in these modules, analyze their dependencies, and detect different characteristics of the modules' relationships: *uses* describes a simple type dependency, *depends on* describes a component dependency required at bootstrap time (effectively implemented as Spring bean reference), and *listens to* describes a dependency established by the presence of an event listener interested in an event published by the module depended on.

Starting with this, a next step would be to investigate the interesting aspects of each individual module: Which publicly available application components do they expose? What events do the modules emit and consume? What are the primary aggregate roots to look at to understand the module's internal working?



Base package	...
Spring components	*Services* • d.o.e.d.m.o.DefaultOrderManagement implementing d.o.e.d.m.o.OrderManagement *Repositories* • d.o.e.d.m.o.OrderRepository *Event listeners* • d.o.e.d.m.o.EmailSender listening to: ◦ d.o.e.d.m.o.OrderCompleted (async)
Aggregate roots	• d.o.e.d.m.o.Order
Published events	• d.o.e.d.m.o.OrderCompleted created by: ◦ d.o.e.d.m.o.Order.complete()

Fig.4 – A Module Canvas generated from the architectural metadata present in the code base

Because of the architectural roles assigned to the source code elements, the documentation integration can detect all of this and create a so called Module Canvas (referring to the Bounded Context Canvas [bccanvas]), a textual representation of the module's provided and required interface. Given that the architectural metadata is an intrinsic part of the source code, the documentation generated is correct by definition. We have seen how the presence of architectural metadata in the codebase can help to create useful documentation situated at levels three and four of Simon Brown's C4 model [c4].

Reducing boilerplate code

In applications written the traditional way, architectural concepts, such as *Aggregate Root*s, are present in the code – but not explicitly. They manifest in the form of a particular application of technology, such as persistence frameworks. Assume the example model will be persisted by using JPA. The *Order* class would be as follows:

```
@Entity
@NoArgsConstructor(force = true)
@EqualsAndHashcode(of = "id")
@Table(name = "SAMPLE_ORDER")
class Order {

  private final @EmbeddedId OrderId id;

  @OneToMany(
    cascade = CascadeType.ALL, orphanRemoval = true)
  private List<LineItem> lineItems;
  private CustomerId customerId;

  Order(Customer customer) {
    this.id = OrderId.of(UUID.randomUUID());
    this.customerId = customer.getId();
  }

  @Value
  @RequiredArgsConstructor(staticName = "of")
  @NoArgsConstructor(force = true)
  static class OrderId implements Serializable {
    private static final long serialVersionUID = …;
    private final UUID orderId;
  }
}
```

A traditionally implemented Order aggregate persisted with JPA

In this example, we use Lombok [lombok] to eliminate the boilerplate code needed to implement the *equals(…)* and *hashCode()* methods suitable for entities. The need for a no-argument constructor and the identifier having to implement *Serializable* is implied by JPA. It pollutes the domain code with technical details, hindering both the understandability and the evolvability of the codebase. Also, most of the JPA-specific annotations found here are actually an expression of the class' aggregate nature.
By using the *jMolecules* abstractions, this code can be decluttered to describe the architectural concepts of the class explicitly:

```
@Table(name = "SAMPLE_ORDER")
class Order implements AggregateRoot<Order, OrderId> {

  private final OrderId id;
  private List<LineItem> lineItems;
  private Association<Customer, CustomerId> customer;

  Order(Customer customer) {
    this.id = OrderId.of(UUID.randomUUID());
    this.customer = Association.forAggregate(customer);
  }

  @Value(staticConstructor = "of")
  static class OrderId implements Identifier {
    private final UUID orderId;
  }
}
```

An Order implementation using jMolecules interfaces

Order now implements *AggregateRoot*, which lets us derive the required implementations of *equals(…)* and *hashCode()*, as well as the default constructors. The relationship to the *Customer* is modeled by using the *Association* type and needs a custom JPA *AttributeConverter* to be registered to persist properly. *OrderId* has become an *Identifier*, and we know we would have to map it by using *@EmbeddableId* and let it implement *Serializable*. We can detect that *LineItem* is an *Entity* belonging to *Order* and default its mapping to *@OneToMany(…)* with full cascading and orphan removal. The only JPA-specific element still left is the *@Table* annotation, because we need to customize that as *Order* is a reserved name in SQL. This shows that we moved from scattering the domain class with verbose technical detail describing the default mappings to only having to declare necessary deviations from those defaults.

All these implementation details can be added to the code by generating the necessary annotations and boilerplate code via a *ByteBuddy* [bytebuddy] plugin that is plugged into the compilation step of the project build.

```
<plugin>
  <groupId>net.bytebuddy</groupId>
  <artifactId>byte-buddy-maven-plugin</artifactId>
  <version>…</version>
  <executions>
    <execution>
      <goals>
        <goal>transform</goal>
      </goals>
    </execution>
  </executions>
  <dependencies>
    <dependency>
      <groupId>org.jmolecules.integrations</groupId>
      <artifactId>jmolecules-bytebuddy</artifactId>
      <version>…</version>
    </dependency>
  </dependencies>
</plugin>
```

An Order implementation that uses jMolecules interfaces

The integration into the compilation turns the *Order* class shown above into a type persistable by JPA as is. The plugin inspects the project's classpath to decide which technical integrations to apply and has the necessary projection steps encoded in its implementation. This results in domain code that is less cluttered with technical details and architectural patterns that are explicitly assigned.

Future Work

While *jMolecules* is already covering a broad set of architectural concepts, there are a few areas of potential further experimentation. The architectural vocabulary can be extended further to include other high-level pattern languages. Also, using the metadata during runtime could be interesting to fuel (for example) observability tools with logical module interaction (tracing) and gathering metrics about event publication and other concerns. Another idea is porting the approach into different programming languages. Initial drafts for that exist for .NET and PHP (see [xmolecules]), but the broad catalog of technology integration already existing for Java is currently missing.

Evaluation and Conclusion

In this paper, we have shown a novel approach to expressing architectural patterns in source code and, thus enabling developers to narrow the gap between an architectural design and the code that implements it. Its fundamental simplicity is its greatest strength: By using existing programming- language-specific means, the entry barrier to apply the technique in the developers' everyday work is very low. The code keeps working as is but benefits from optional integration into IDEs and the build system. The source code becomes more expressive and enables tooling to extract significant parts from it to process it into higher-level documentation that supports understandability of the software system. Also, less boilerplate code is needed to map the source code onto particular integration technologies and frameworks, as the expression of the architectural concepts lets the mapping be encoded in code generators and applied automatically at build time. The latter might be considered a drawback as it introduces a level of indirection between the source code in the IDE and what is executed. That's why the boilerplate code generation is optional integration available to those who are willing to buy into the tradeoff.

jMolecules allows writing code better suited for long-term maintenance and, thus, avoiding software degradation that usually eventually results in costly rewrites.

References

[jmolecules] jMolecules – https://www.jmolecules.org

[evans03] E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*, Addison Wesley, 2003.

[palermo08] J. Palermo, Onion Architecture – Part I, website, 2008. https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1

[jpa] Jakarta Persistence API – https://en.wikipedia.org/wiki/Jakarta_Persistence

[jackson] Jackson (Java JSON library) – https://github.com/FasterXML/jackson

[lilienthal17] C. Lilienthal, *Sustainable Software Architecture - Analyze and Reduce Technical Debt*, 2nd ed. dpunkt Verlag, 2017.

[hermans21] F. Hermans, *The Programmer's Brain – What every programmer needs to know about cognition*, 1st ed. Manning, 2021.

[fairbanks12] G. Fairbanks, *Just Enough Software Architecture - A Risk-Driven Approach*. Marshall & Brainerd, 2012.

[eden-kazman03] A. H. Eden and R. Kazman, *Architecture, Design, Implementation*, International Conference on Software Engineering, 2003.

[archjava] ArchJava, https://projectsweb.cs.washington.edu/research/projects/cecil/www/Internal/hype/att-0476/01-part

[mda] Model-Driven Architecture – https://en.wikipedia.org/wiki/Model-driven_architecture

[spring] Spring Framework – https://www.spring.io

[vernon13] V. Vernon, *Implementing Domain-Driven Design*, Addison Wesley, 2013.

[archunit] ArchUnit – https://www.archunit.org

[jqassistant] jQAssistant – https://jqassistant.org

[bccanvas] Bounded Context Canvas – https://github.com/ddd-crew/bounded-context-canvas

[c4] C4 Model – https://c4model

[lombok] Project Lombok – https://projectlombok.org

[bytebuddy] ByteBuddy – https://www.bytebuddy.net

[xmolecules] xMolecules umbrella project – https://www.xmolecules.org

All internet resources were checked in January, 2022.

Appendix

Acknowledgements

Jens Schauder and Dr. Dave Syer for feedback. Jay Bryant for copy editing.

Glossary

Aggregate – A group of related entities (see *Entity*) over which a set of business and consistency rules have to be enforced.

Aggregate Root – The root entity of the aggregate.

C4 Model – A hierarchical set of models and corresponding kinds of diagrams to describe

CQRS – Command Query Responsibility Segregation.

DDD – Domain-Driven Design.

Entity – A domain concept that, contrary to *Value Object*s, has identity and lifecycle.

IDE – Integrated Development Environment.

JPA – Jakarta Persistence API, the default object-relational mapping standard.

SQL – Structured Query Language. A language to interact with relational databases.

UML – Unified Modeling Language.

Value Object – A domain concept that doesn't have identity, which means that same values can be represented by the same underlying instance (a zip code, for example).