*18*

## DSLs and Software Architecture

*In this chapter we explore the relationship between DSLs and software architecture. In particular we establish the notion of an Architecture DSL (or ADSL), which is a DSL specifically created for a specific architecture. We first discuss ADSLs based on an extensive example, then discuss the conceptual details. The chapter also looks at embedding business logic DSLs, as well as at the role software components can play in the context of DSLs.*

*18.1 What is Software Architecture?*

*Definitions* Software architecture has many definitions, from various groups of people. Here are a few. Wikipedia defines software architecture in the following way:

The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships between them.

This is a classic definition that emphasizes the (coarse grained) structure of a software system (as opposed to the behavior), observable by analyzing an existing system[[1]](#footnote-1). A definition by

Boehm builds on this:

A software system architecture comprises a collection of software and system components, connections, and constraints, a

collection of system stakeholders’ need statements as well as a rationale which demonstrates that the components, connections, and constraints define a system that, if implemented, would satisfy the collection of system stakeholders’ need statements.

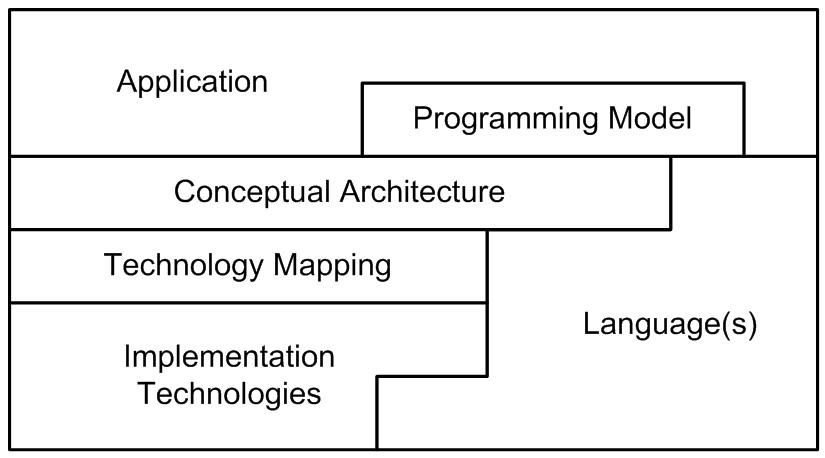
In addition to the structure, this definition also emphasizes the relevance of the stakeholders and their needs, and ties the structure of the system to what the system is required to do. Hayes-Roth introduce another aspect:

The architecture of a complex software system is its style and method of design and construction.

Instead of looking at the structures, they emphasize that there are different architectural styles and they emphasize the "method of design and construction"2. Eoin Woods takes it one step fur-

|  |  |
| --- | --- |
| ther:  Software architecture is the set of design decisions which, if made incorrectly, may cause your project to be cancelled.  He emphasizes the design decisions that lead to a given system. So he doesn’t look at the system as a set of structures, but rather considers the architecture as a process – the design |  |
| decisions – that leads to a given system. While I don’t disagree with any of these definitions – they all emphasize different aspects of the architecture of a software system – I would like to propose another one:  Software architecture is everything that needs to be consistent throughout a software system. |  |

|  |  |
| --- | --- |
| can help with achieving this consistency. |  |
| *Terminology* Fig. 18.1 looks at software architecture in more |  |
| detail. At the center of the diagram you can see the *conceptual architecture*. This defines architectural concepts from which concrete systems are built (we will come back to this in the next section). Examples could include: *task*, *message*, *queue*, *component*, *port*, or *replicated data structure*. Note that these concepts are independent of specific implementation technologies: it is the *technology mapping* that defines how the architecture concepts are implemented based on specific technologies4. The *programming model* is the way in which the architectural concepts are implemented in a given programming language5. |  |
| Applications are implemented by instantiating the architectural | implementation technologies change. |

This definition is useful because it includes structures *and* behavior; it doesn’t say anything about coarse-grained versus detailed3 and it implies that there needs to be some kind of process or method to achieve the consistency. As we will see, DSLs concepts, and implementing them based on the programming model.

*The Importance of Concepts* I want to reemphasize the importance of the conceptual architecture. When asking people about the architecture of systems, one often gets answers like: "It’s a Web service architecture", or "It’s an XML architecture" or "It’s a JEE architecture". Obviously, all this conveys is that a certain technology is used. When talking about architectures per se, we should talk about architectural *concepts* and how they relate to each other. Only as a second step should a mapping to one or more technologies be discussed. Here are some of these fundamental architectural concepts:

*Modularize* Break big things down into smaller things, so they can be understood (and potentially reused) more easily. Examples: procedures, classes, components, services, user stories.

*Encapsulate* Hide the innards of a module so that they can be changed without affecting clients. Examples: private members, facade pattern, components, layers/rings/levels.

*Contracts* Describe the external interface of a module clearly. Examples: interfaces, pre- and post-conditions, protocol state machines, message exchange patterns, published APIs.

*Decoupling* Reduce dependencies in time, data structure or contention. Examples: message queues, eventual consistency, compensating transactions.

*Isolate crosscuts* Encapsulate handling of cross-cutting concerns. Examples: aspect orientation, interceptors, application servers, exception handling.

As we will see in the following section, architecture DSLs can specify architectures unambiguously while emphasizing these fundamentals, instead of technologies.

### 18.2 Architecture DSLs

|  |  |
| --- | --- |
| the application architecture in the implementation language(s), automating the technology mapping7. Finally, the program- |  |
| ming model is defined with regards to the generated code plus possibly additional frameworks8.  I do not advocate the definition of a generic, reusable language such as the various ADLs, or UML (see below). Based on our experience, the approach works best if you define the ADSL in real time as you understand, define and evolve the conceptual architecture of a system9. The process of defining the language actually helps the architecture/development team to better understand, clarify and refine the architectural abstractions, as the language serves as a (formalized) ubiquitous language that lets you reason about and discuss the architecture.  *18.2.1 An Example ADSL*  This section contains an example of an Architecture DSL taken from a real system in the domain of airport management sys- |  |
| tems10. |  |
| *Background* The customer decided they wanted to build a new airport management system. Airlines use systems like these to track and publish information about whether airplanes have landed at airports, whether they are late, and to track the |  |
| technical status of the aircraft11. The system also populates |  |

An Architecture DSL (ADSL) is a language that expresses a system’s architecture directly. "Directly" means that the language’s abstract syntax contains constructs for all the ingredients of the conceptual architecture. The language can hence be used to describe a system on the architectural level without using low-level implementation code, but still in an unambiguous way6. Code generation is used to generate representations of

the online-tracking system on the Web and information monitors at airports. This system is in many ways a typical distributed system: there is a central data center to do some of the heavy number crunching, but there are additional machines distributed over relatively large areas. Consequently you cannot simply shut down the whole system, which leads to a requirement to be able to work with different versions of parts of the system at the same time. Different parts of the system will be built with different technologies: Java, C++, C#. This is not an untypical requirement for large distributed systems either. Often you use Java technology for the backend, and .NET technology for a Windows front end. The customer had decided that the backbone of the system would be a messaging infrastructure, and they were evaluating different messaging tools for performance and throughput.

While my customer knew many of the requirements and had made specific decisions about some architectural aspects, they didn’t have a well-defined conceptual architecture. It showed: when the team were discussing their system, they stumbled into disagreements about architectural concepts all the time because they had no *language* for the architecture. Also, they didn’t have a good idea of how to maintain the architecture over the 20 years of expected lifetime in the face of changing technologies.

*Getting Started* To solve this issue, an architecture DSL was developed. We started with the notion of a component. At that point the notion of components is defined relatively loosely, simply as the smallest architecturally relevant building block, a piece of encapsulated application functionality12. We also

|  |  |
| --- | --- |
| assume that components can be instantiated, making components the architectural equivalent to classes in object-oriented programming. To enable components to interact with each other, we also introduce the notion of interfaces, as well as ports, which are named communication endpoints typed with |  |
| an interface. Ports have a direction13 (**provides**, **requires**) as |  |
| well as a cardinality. Based on this initial version of the ADSL, we could write the following example code14: |  |

|  |
| --- |
| **component** DelayCalculator { **provides** aircraft: IAircraftStatus **provides** managementConsole: IManagementConsole **requires** screens[0..n]: IInfoScreen  }  **component** Manager { **requires** backend[1]: IManagementConsole  }  **component** InfoScreen { **provides** default: IInfoScreen  }  **component** AircraftModule { **requires** calculator[1]: IAircraftStatus } |

|  |  |
| --- | --- |
| We then looked at instantiation. There are many aircraft, each running an **AircraftModule**, and there are even more **Info-** |  |
| **Screen**s. So we need to express instances of components15. |  |

We also introduce connectors to define actual communication paths between components (and their ports).

**instance** dc: DelayCalculator **instance** screen1: InfoScreen **instance** screen2: InfoScreen **connect** dc.screens **to** (screen1.default, screen2.default)

Decisions about pooling and redundant physical instances had not been made yet.

*Organizing the System* At some point it became clear that in order to not get lost in all the components, instances and connectors, we need to introduce some kind of namespace. It became equally clear that we’d need to distribute things to different files:

|  |
| --- |
| **namespace** com.mycompany { **namespace** datacenter {  **component** DelayCalculator { **provides** aircraft: IAircraftStatus **provides** managementConsole: IManagementConsole **requires** screens[0..n]: IInfoScreen  }  **component** Manager { **requires** backend[1]: IManagementConsole } } **namespace** mobile {  **component** InfoScreen { **provides** default: IInfoScreen  }  **component** AircraftModule { **requires** calculator[1]: IAircraftStatus }  }  } |

It is also a good idea to keep component and interface definitions (essentially type definitions) separate from system definitions (connected instances), so we introduced the concept of compositions, which make a group of instances and connectors identifiable by a name:

|  |
| --- |
| **namespace** com.mycompany.test {  **composition** testSystem { **instance** dc: DelayCalculator **instance** screen1: InfoScreen **instance** screen2: InfoScreen  **connect** dc.screens **to** (screen1.default, screen2.default)  }  } |

*Dynamic Connectors* Of course in a real system, the **DelayCalculator** would have to dynamically discover all the available **InfoScreen**s at runtime. There is not much point in manually describing those connections. So, we specify a query that is executed at runtime against some kind of naming/trader/lookup/registry infrastructure[[2]](#footnote-2). It is re-executed every 60 seconds to find the **InfoScreen**s that have just come on line.

|  |
| --- |
| **namespace** com.mycompany.production { **instance** dc: DelayCalculator  // InfoScreen instances are created and started in other configurations **dynamic connect** dc.screens **query** { type = IInfoScreen status = active  }  } |

|  |
| --- |
| **interface** IAircraftStatus { **oneway message** reportPosition(aircraft: ID, pos: Position )  **request**-**reply message** reportProblem { **request** (aircraft: ID, problem: Problem, comment: String) **reply** (repairProcedure: ID)  }  } |

A similar approach can be used to address load balancing or fault tolerance. A static connector can point to a primary as well as a backup instance. Or a dynamic query can be reexecuted when the currently used instance becomes unavailable. To support registration of instances with the naming or registry service, we add additional syntax to their definition. A registered instance automatically registers itself with the registry, using its name (qualified through the namespace) and all provided interfaces. Additional parameters can be specified, and the following example registers a primary and a backup instance for the **DelayCalculator**:

|  |
| --- |
| **namespace** com.mycompany.datacenter {  **registered instance** dc1: DelayCalculator { **registration parameters** {role = primary}  }  **registered instance** dc2: DelayCalculator { **registration parameters** {role = backup} }  } |

*Interfaces* So far we hadn’t really defined what an interface is. We knew that we’d like to build the system based on a messaging infrastructure. Here’s our first idea: an interface is a collection of messages, where each message has a name and a list of typed parameters[[3]](#footnote-3). After discussing this notion of

interfaces for a while, we noticed that it was too simplistic. We needed to be able to define the direction of a message: does

it flow in or out of the port? More generally, which kinds of message interaction patterns are there? We identified several; here are examples of **oneway** and **request-reply**[[4]](#footnote-4):

We talked a long time about various message interaction patterns. After a while it turned out that one of the core use cases for messages was to push updates of various data structures out to various interested parties. For example, if a flight was delayed because of a technical problem with an aircraft, then this information had to be pushed out to all the **InfoScreen**s in the system. We prototyped several of the messages necessary for "broadcasting" complete updates, incremental updates and removal of data items. And then it hit us: we were working with the wrong abstraction!

*Data Replication* While messaging is a suitable *transport* abstraction for these things, architecturally we’re really talking about replicated data structures. It basically works the same way for all of those structures:

* You define a data structure (such as **FlightInfo**).
* The system then keeps track of a collection of such data structures.
* This collection is updated by a few components and typically read by many other components.
* The update strategies from publisher to receiver always include full update of all items in the collection, incremental updates of just one or a few items, and removal of items.

Once we understood that in addition to messaging, there’s this additional core abstraction in the system, we added this to our Architecture DSL and were able to write something like the following. We define data structures and replicated items. Components can then publish or consume those replicated data structures. We state that the publisher publishes the replicated data whenever something changes in the local data structure.

However, the **InfoScreen** only needs an update every 60 seconds (as well as a full load of data when it is started up).

|  |
| --- |
| **struct** FlightInfo { from: Airport to: Airport scheduled: Time expected: Time  }  **replicated singleton** flights { flights: FlightInfo[]  }  **component** DelayCalculator {  **publishes** flights { **publication** = **onchange** } |

}

**component** InfoScreen {

**consumes** flights { **update** 60 }

}

This is much more concise compared to a description based on messages. We can automatically derive the kinds of messages needed for full update, incremental update and removal, and create these messages in the model using a model transformation. The description reflects much more clearly the actual architectural intent: it expresses better what we want to do (replicate data) compared to a lower-level description of how we want to do it (sending around update messages)19.

*Interface Semantics* While replication is a core concept for data, there is of course still a need for messages, not just as an implementation detail, but also as a way to express architectural intent20. It is useful to add more semantics to an interface, for example to define valid sequencing of messages. A well-known way to do that is to use protocol state machines. Here is an example. It expresses the fact that you can only report positions and problems once an aircraft is registered. In other words, the first thing an aircraft has to do is register itself.

|  |
| --- |
| **interface** IAircraftStatus { **oneway message** registerAircraft(aircraft: ID ) **oneway message** unregisterAircraft(aircraft: ID ) **oneway message** reportPosition(aircraft: ID, pos: Position )  **request**-**reply message** reportProblem { **request** (aircraft: ID, problem: Problem, comment: String) **reply** (repairProcedure: ID)  }  **protocol initial** = new {  **state** new { registerAircraft => registered  }  **state** registered { unregisterAircraft => new reportPosition reportProblem  }  }  } |

.

Initially, the protocol state machine is in the **new** state where the only valid message is **registerAircraft**. If this is received, we transition into the **registered** state. In **registered**, you can either **unregisterAircraft** and go back to **new**, or receive a **reportProblem** or **reportPosition** message, in which case you will remain in the **registered** state.

*Versioning* We mentioned above that the system is distributed geographically. This means it is not feasible to update all the software for all parts of the systems (e.g., all **InfoScreen**s or all **AircraftModule**s) at the same time. As a consequence, there might be several versions of the same component running in the system. To make this feasible, many non-trivial things need to be put in place in the runtime system. But the basic requirement is this: you have to be able to mark versions of components, and you have to be able to check them for compatibility with older versions. The following piece of code expresses the fact that the **DelayCalculatorV2** is a new implementation of **DelayCalculator**. **newImplOf** means that no externally visible aspects change, which is why no ports and other externally exposed details of the component are declared21.

|  |
| --- |
| **component** DelayCalculator {  **publishes** flights { **publication** = **onchange** }  }  **newImplOf component** DelayCalculator: DelayCalculatorV2 |

To evolve the externally visible signature of a component, one can write this:

|  |
| --- |
| **component** DelayCalculator {  **publishes** flights { **publication** = **onchange** }  }  **newVersionOf component** DelayCalculator: DelayCalculatorV3 {  **publishes** flights { **publication** = **onchange** } **provides** somethingElse: ISomething } |
| The keyword **newVersionOf** allows us to add additional provided ports (such as the **somethingElse** port) and to remove | | |  |
| required ports22. | | |  |

This wraps up our case study[[5]](#footnote-5). In the next subsection we recap the approach and provide additional guidance.

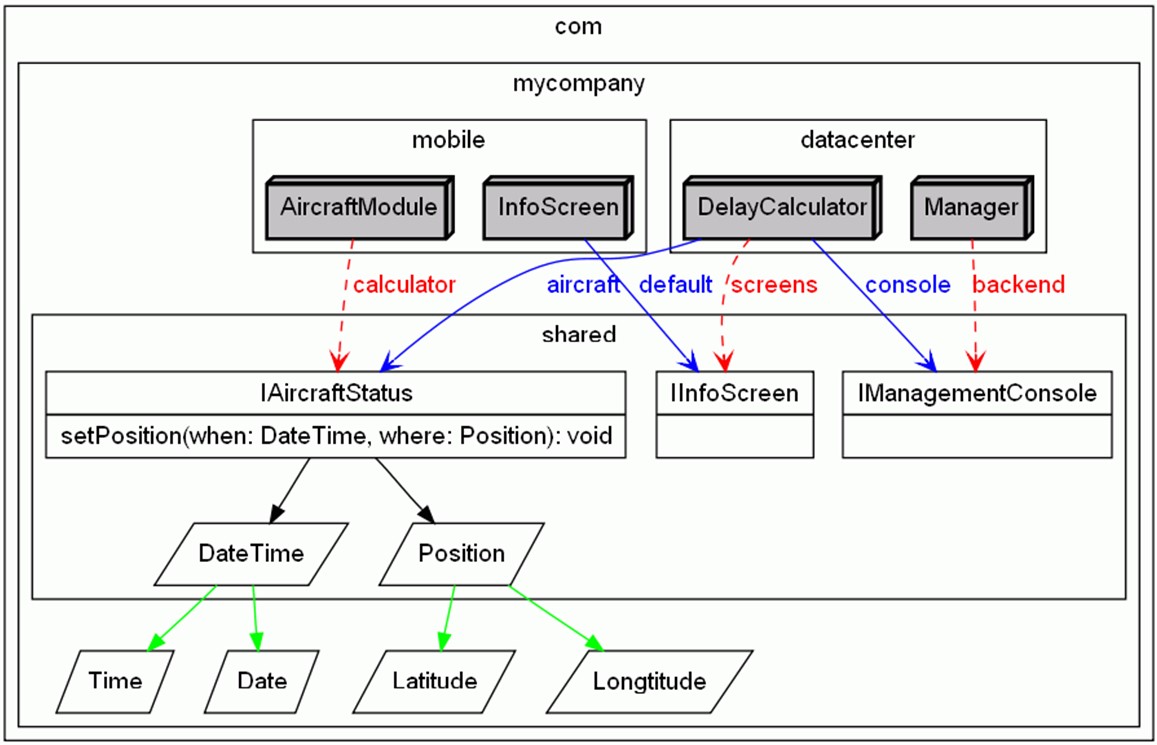
#### 18.2.2 Architecture DSL Concepts

*What we did in a Nutshell* Using the approach shown here, we were able to quickly get a grip of the overall architecture of the system. All of the above was actually done in the first day of the workshop. We defined the grammar, some important constraints, and a basic editor (without many bells and whistles). We were able to separate what we wanted the system to do from how it would achieve it: all the technology discussions were now merely an implementation detail of the conceptual

ports or remove any of the provided ports, since that would destroy the "plug-in compatibility". Constraints make sure that these rules are enforced on the model level.

Unfortunately, efforts like that completely miss the point. We have not experienced much benefit in shoehorning an architecture description into the (typically very limited, as well as too generic) constructs provided by predefined languages – one of the core activities of the approach explained is this chapter is the process of actually building *your own language* to capture your system’s specific conceptual architecture.

|  |  |
| --- | --- |
| descriptions given here24. We also achieved a clear and unam- |  |
| biguous definition of what the different architectural concepts mean. The generally nebulous concept of *component* has a formal, well-defined meaning in the context of this system.  The approach discussed in this chapter therefore recommends the definition of a formal language for your project’s or system’s conceptual architecture. You develop the language as the understanding of your architecture grows. The language therefore always resembles the complete understanding about your architecture in a clear and unambiguous way.  *Component Implementation* By default, architecture DSLs are incomplete; component implementation code is written manually against the generated API, using well-known composition techniques such as inheritance, delegation or partial classes.  However, there are other alternatives for component implementation that do not use a GPL, but instead use formalisms that are specific to certain classes of behavior: state machines, business rules or workflows. You can also define and use a domain-specific language for certain classes of functionality in a specific business domain. If such an approach is used, then the code generator for the application domain DSL has to act as the "implementor" of the components (or whatever other architectural abstractions are defined in the architecture DSL). The code generated from the business logic DSL must fit into the code skeletons generated from the architecture DSL. The code composition techniques discussed for incomplete DSLs in |  |
| Section 4.5.1 can be used here25. |  |
| *Standards, ADLs and UML* Describing architecture with formal languages is not a new idea. Various communities recommend using Architecture Description Languages (ADLs) or the Unified Modeling Language (UML). However, all of those approaches advocate using existing, *generic* languages for specifying architecture, although some of them, including UML, |  |
| can be customized to some degree26. |  |

* We generate an API against which the implementation is coded. That API can be non-trivial, taking into account the various messaging paradigms, replicated state, etc. The generated API allows developers to code the implementation in a way that does not depend on any technological decisions: the generated API hides those from the component implementation code. We call this generated API, and the set of idioms to use it, the *programming model*.
* Remember that we expect some kind of component container or middleware platform to run the components, so we also generate the code that is necessary to run the components (including their technology-neutral implementation)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *Visualization* In this project, as well as in many other ones, we have used textual DSLs. We have argued in this book why textual DSLs are superior in many cases, and these arguments apply here as well. However, we did use visualization to show the relationships between the building blocks, and to communicate the architecture to stakeholders who were not willing to dive into the textual models. Figure 18.2 shows an example, created with Graphviz.  .   |  |  | | --- | --- | | *Code Generation* Now that we have a formal model of the conceptual architecture (the language) and also a formal description of system(s) we are building – i.e., the models defined |  | | using the language – we can exploit this by generating code27: |  | |

|  |  |
| --- | --- |
|  |  |
| It is of course completely feasible to generate APIs for several target languages (supporting component implementation in various languages) and/or generating glue code for several target platforms (supporting the execution of the same component on different middleware platforms). This nicely supports |  |
| potential multi-platform requirements29. |  |
| Another important point is that you typically generate in several phases: a first phase uses type definitions (components, data structures, interfaces) to generate the API code so you can write the component implementation code. A second phase generates the glue code and the system configuration code. Consequently, it is often sensible to separate type definitions |  |
| from system definitions into several different viewpoints30: these |  |
| are used at different times in the overall process, and also often created, modified and processed by different people.  In summary, the generated code supports an efficient and technology independent implementation and hides much of the underlying technological complexity, making development more efficient and less error-prone.  *What Needs to be Documented?* I advertise the above approach as a way to formally describe a system’s conceptual and application architecture. So, this means it serves as some kind of documentation, right? Right, but it does not mean that you don’t have to document anything else. The following things |  |
| still need to be documented31: |  |
| *Rationales/Architectural Decisions* The DSLs describe *what* your architecture looks like, but it does not explain *why* it looks the way it does. You still need to document the rationales for architectural and technological decisions. Note that the grammar of your architecture DSL is a really good baseline for such a documentation. Each of the constructs is the result of architectural decisions. So, if you explain for each grammar element why it is there (and why other alternatives have not been chosen) you are well on your way to documenting the important architectural decisions. A similar approach can be used for the application architecture, i.e. the programs written with the DSL. |  |

on the implementation technology of choice. We call this layer of code the *technology mapping* code (or *glue code*). It typically also contains the configuration files for the various platforms involved28.

*User Guides* A language grammar can serve as a well-defined and formal way of capturing an architecture, but it is not a good teaching tool. So you need to create tutorials for your users (i.e., the application programmers) that explain how to use the architecture and the DSL. This includes what and how to model (using your DSL) and also how to generate code and how to use the programming model (how to fill in the implementation code into the generated skeletons).

### 18.3 Component Models

As I have hinted at already, I think that component-based software architectures are extremely useful. There are many (more or less formal) definitions of what a components is. They range from a building block of software systems, through something with explicitly defined context dependencies, to something that contains business logic and is run inside a container.

Our understanding (notice we are not saying we have a real definition) is that a component is the smallest architectural building block. When defining a system’s architecture, you typically don’t look inside components. Components have to specify all their architecturally relevant properties declaratively (using meta data, or models). As a consequence, components become analyzable and composable by tools. Typically they run inside a container that serves as a framework to act on the runtime-relevant parts of the meta data. The component boundary is the level at which the container can provide technical services such as as logging, monitoring or fail-over. The component also provides well-defined APIs to its implementation to address cross-cutting architectural concerns such as locking[[6]](#footnote-6).

I don’t have any specific requirements for what meta data a component actually contains (and hence, which properties are described). The concrete notion of components has to be defined for each (system/platform/product line) architecture separately: this is exactly what we do with the language approach introduced above.

Based on my experience, it is almost always useful to start by modeling the component structure of the system to be built. To do that, we start by defining what a component actually is – that is, by defining a meta model for component-based development. Independent of the project’s domain, these meta

book *Server Component Patterns*. While the EJB-based technology examples are no longer relevant, many of the patterns identified and discussed in the book still are. The book was not written with a MDD/DSL background, but the patterns fit very well with such an approach.

models are quite similar, with a set of specific variation points. We show parts of these meta models here to give you a head start when defining your own component architecture.

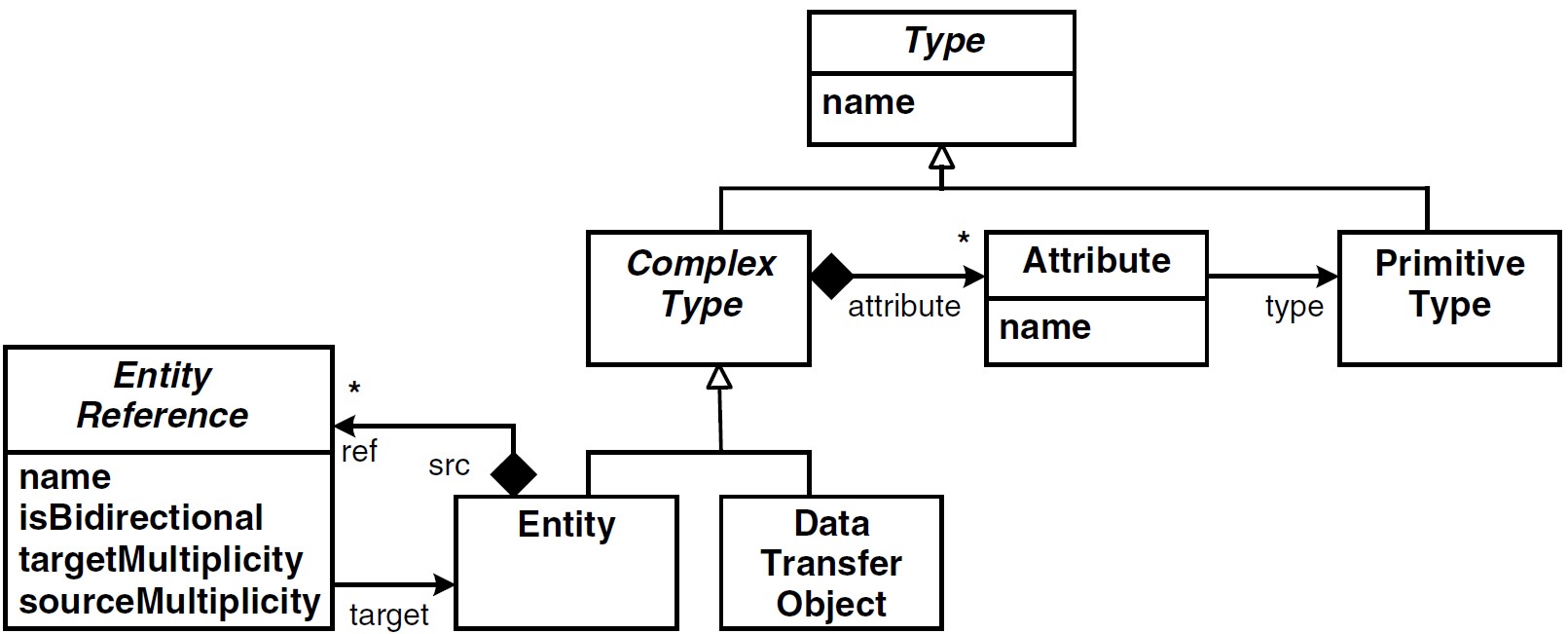
#### 18.3.1 Three Typical Viewpoints

It is useful to look at a component-based system from several viewpoints (Section 4.4). Three viewpoints form the backbone of the description.

*The Type Viewpoint* The Type viewpoint describes component types, interfaces and data structures33. A component pro-

vides a number of interfaces and references a number of required interfaces (often through ports, as in the example above). An interface owns a number of operations, each with a return type, parameters and exceptions. Fig. 18.3 shows this.

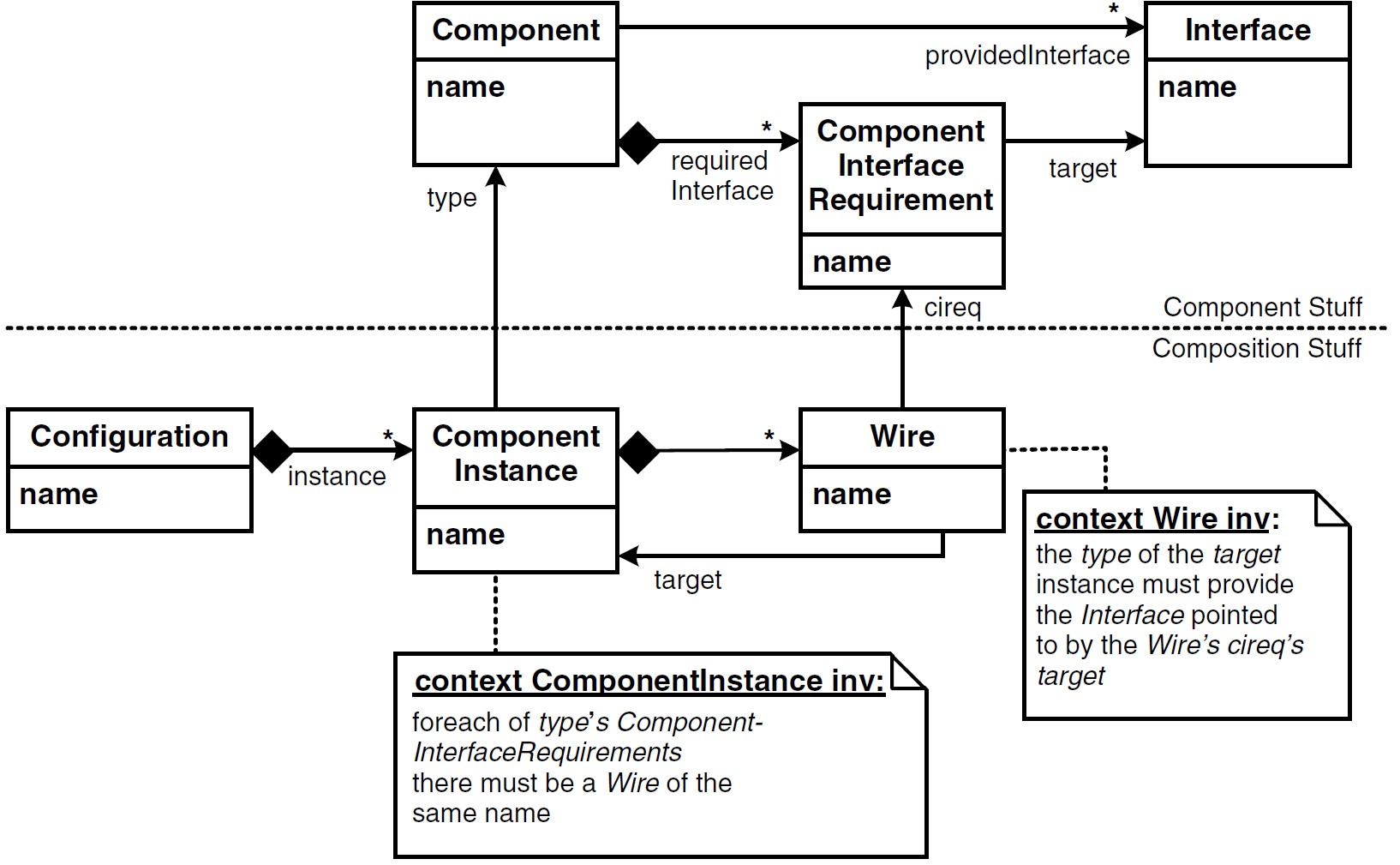
|  |
| --- |
| To describe the data structures with which the components work (the meta model is shown in Fig. 18.4), we start out with the abstract concept **Type**. We use primitive types as well as complex types. A **ComplexType** has a number of named and typed attributes. There are two kinds of complex types. Data transfer objects are simple (C-style) **struct**s that are used to exchange data among components. Entities have a unique ID and can be persisted (this is not visible from the meta model). Entities can reference each other and can thus build more complex data graphs. Each reference specifies whether it is navigable in only one or in both directions. A reference also specifies the cardinalities of the entities at the respective ends, and whether the reference has containment semantics. |



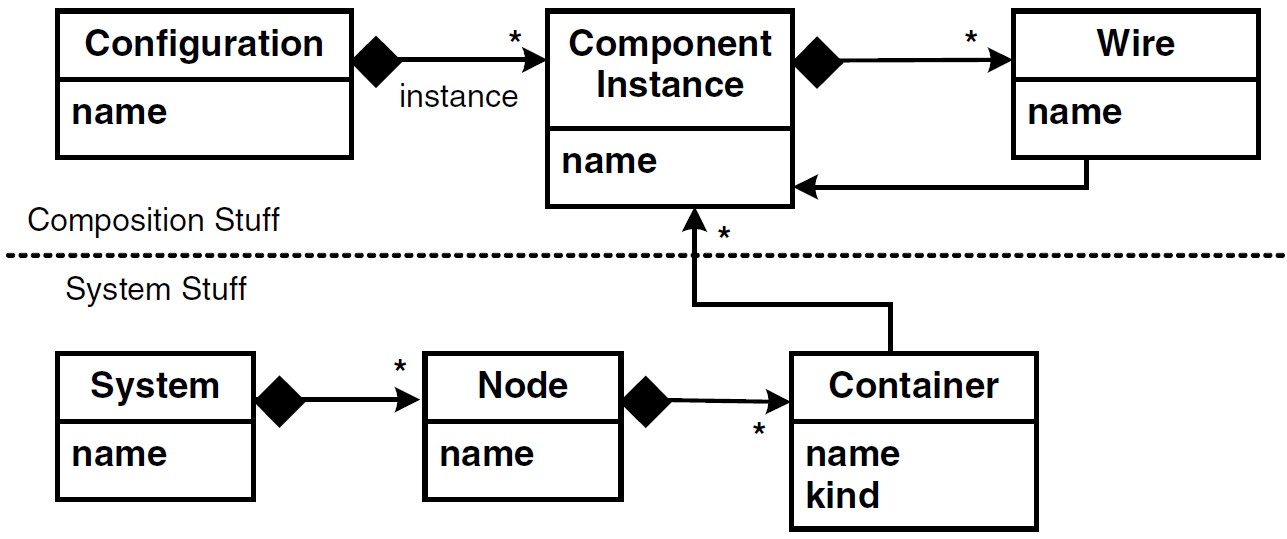
*The Composition Viewpoint* The Composition viewpoint, illustrated in Fig. 18.5, describes component instances and how they are connected34. Using the *Type* and *Composition* view-

points, we can define logical models of applications. A **Configuration** consists of a number of **ComponentInstance**s, each referencing their type (from the *Type* viewpoint). An instance has a number of wires (or connectors): a **Wire** can be seen as an instance of a **ComponentInterfaceRequirement**. Note the constraints defined in the meta model:

* For each **ComponentInterfaceRequirement** defined in the instance’s type, we need to supply a wire.
* The type of the component instance at the target end of a wire needs to provide the interface to which the wire’s component interface requirement points.



|  |  |
| --- | --- |
| *The System Viewpoint* The system viewpoint describes the |  |
| system infrastructure onto which the logical system defined with the two previous viewpoints is deployed (Fig. 18.6), as well as the mapping of the Composition viewpoint onto this execution infrastructure35. |  |

.

A system consists of a number of nodes, each one hosting containers. A container hosts a number of component instances. Note that a container also defines its kind – representing technologies such as OSGi, JEE, Eclipse or Spring. Based on this data, together with the data in the Composition viewpoint, you can generate the necessary "glue" code to run the components in that kind of container, including container and remote communication configuration code, as well as scripts to package and deploy the artifacts for each container.

You may have observed that the dependencies among the viewpoints are well-structured. Since you want to be able to define several compositions using the same components and interfaces, and since you want to be able to run the same compositions on several infrastructures, dependencies are only legal in the directions shown in figure 18.7.

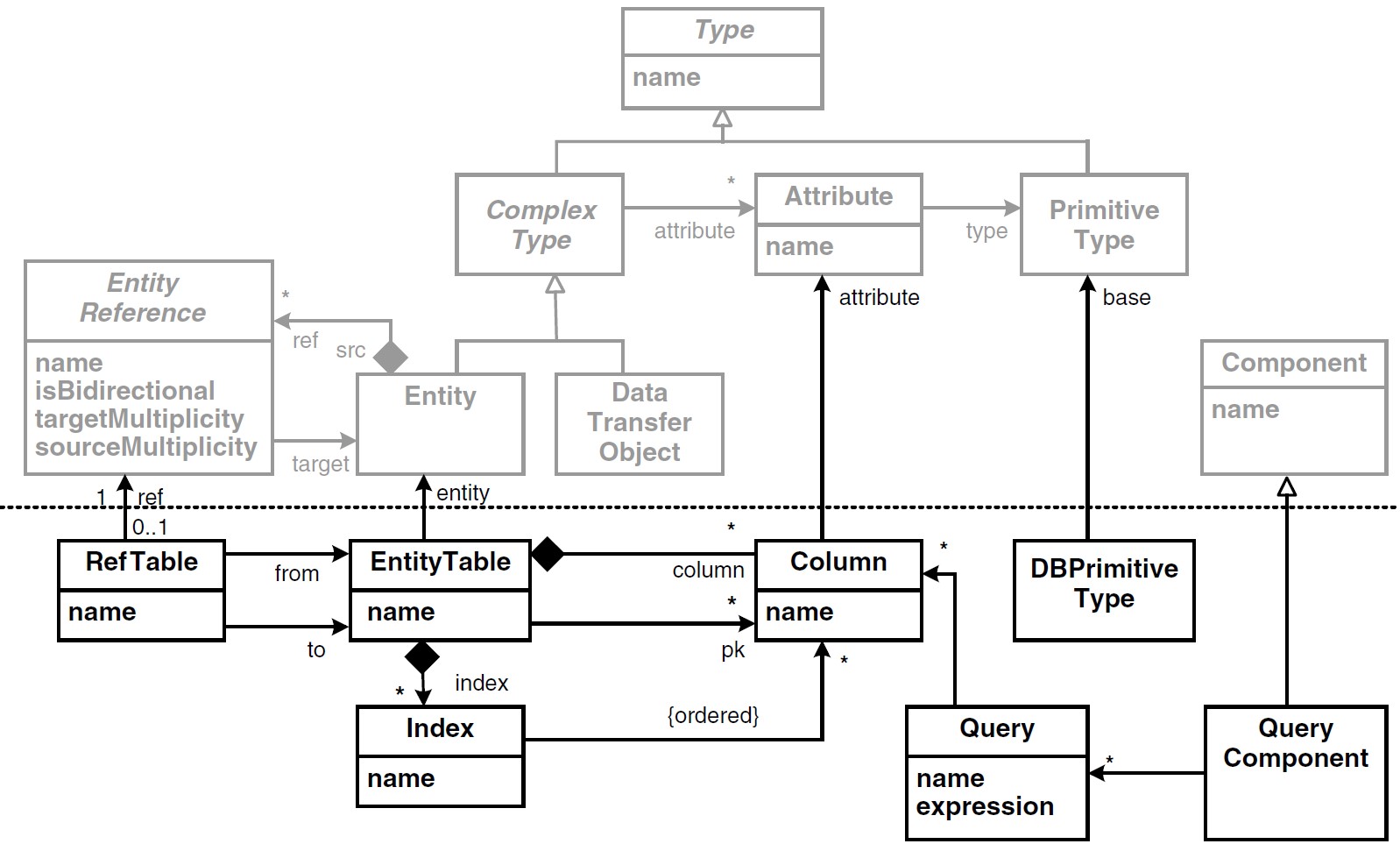
.

#### 18.3.2 Aspect Models

The three viewpoints described above are a good starting point for modeling and building component-based systems. However, in many cases these three models are not enough. Additional aspects of the system have to be described using specific aspect models that are arranged "around" the three core viewpoint models. The following aspects are typically handled in separate aspect models:

* Persistence
* Authorization and Authentication (for enterprise systems)
* Forms, layout, page flow (for Web applications)
* Timing, scheduling and other quality of service aspects (especially in embedded systems)
* Packaging and deployment
* Diagnostics and monitoring

The idea of aspect models is that the information is not added to the three core viewpoints, but rather is described using a separate model with a suitable concrete syntax. Again, the meta model dependencies are important: the aspects may depend on the core viewpoint models and maybe even on one another, but the core viewpoints must not depend on any of the aspect models. Figure 18.8 illustrates a simplified persistence aspect meta model.



|  |  |
| --- | --- |
| *18.3.3 Variations*  The meta models described above cannot be used in exactly this way in every project. Also, in many cases the notion of |  |
| what constitutes a *Component* needs to be adapted or extended. As a consequence, there are many variations of these meta models. In this section we discuss a few of them36. |  |

*Messaging* Instead of operations and their typical call/block semantics, you may want to use messages, together with the well-known message interaction patterns. The example system in this chapter used messaging.

*No Interfaces* Operations could be added directly to the components. As a consequence, of course, you cannot reuse the interface’s "contracts" separately, independently of the supplier or consumer components. You cannot implement interface polymorphism.

*Component Kinds* Often you’ll need different kinds of components, such as domain components, data access components, process components or business rules components. Depending on this component classification, you can define the valid dependency structures between components (e.g., a domain component may access a data access component, but not the other way round)37.

*Layers* Another way of managing dependencies is to mark each component with a layer tag, such as domain, service, GUI

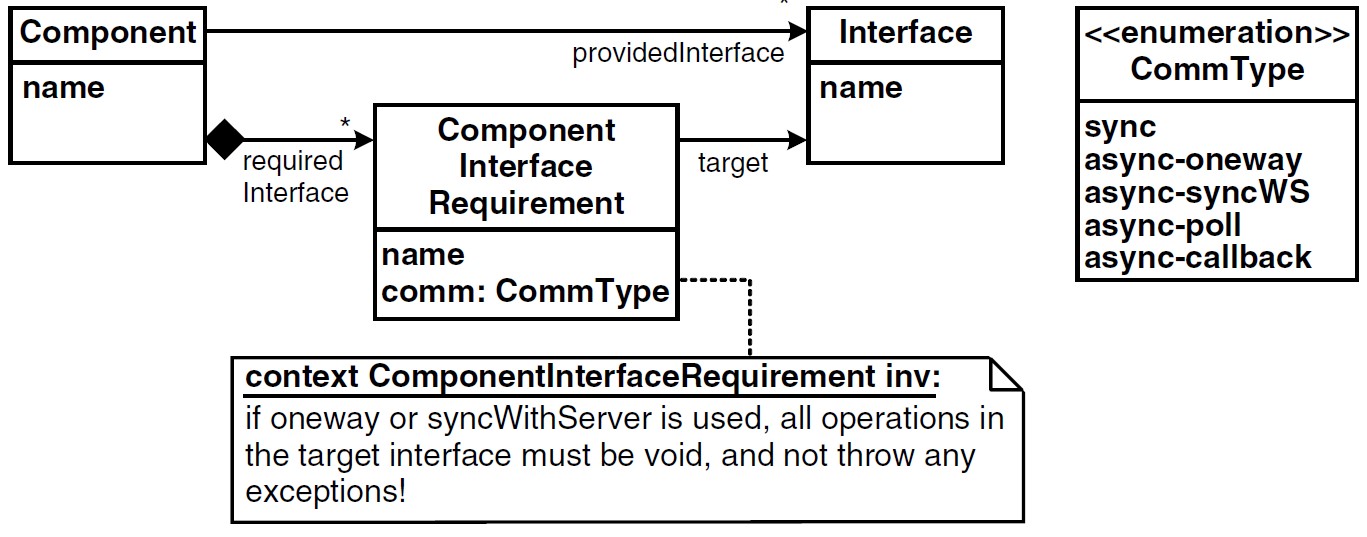
or facade, and define constraints on how components in these layers may depend on each other.

*Configuration Parameters* A component might have a number of configuration parameters – comparable to command line arguments in console programs – that help configure the behavior of components. The parameters and their types are defined in the type model, and values for the parameters can be specified later, for example in the models for the Composition or the System viewpoints.

*Component Characteristics* You may want to express whether a components is stateless or stateful, whether they are threadsafe or not, and what their lifecycle should look like (for example, whether they are passive or active, whether they want to be notified of lifecycle events such as activation, and so on).

*Asynchronicity* Even if you use operations (and not messaging), it is not always enough to use simple synchronous communication. Instead, one of the various asynchronous communication paradigms, such as those described in the Remoting Patterns book[[7]](#footnote-7), might be applicable. Because using these

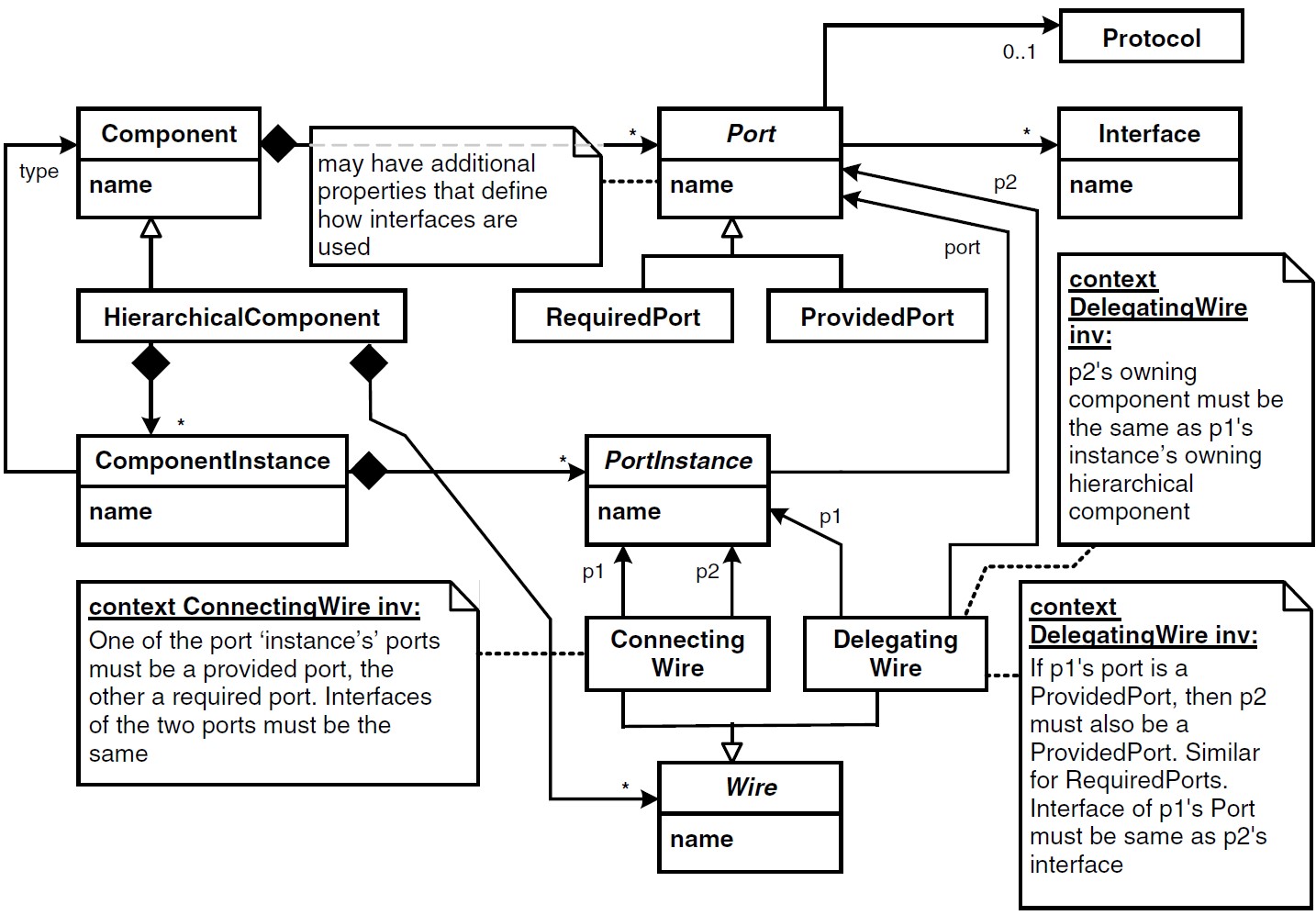
paradigms affects the APIs of the components, the pattern to be used has to be marked up in the model for the *Type* viewpoint, as shown in Fig. 18.9. It is not enough to define it in the Composition viewpoint39.

.

|  |  |
| --- | --- |
| *Events* In addition to the communication through interfaces, you might need (asynchronous) events using a static or |  |
| dynamic publisher/subscriber infrastructure40. |  |
| *Dynamic Connection* The Composition viewpoint connects component instances statically. This is not always feasible. If dynamic wiring is necessary, the best way is to embed the information that determines which instance to connect to at runtime into the static wiring model. So, instead of specifying in the model that instance **A** must be wired to instance **B**, the model only specifies that **A** needs to connect to a component with the following properties: it needs to provide a specific interface, and for example offer a certain reliability. At runtime, the wire is "deferenced" to a suitable instance using an instance |  |
| repository41. |  |

*Hierarchical Components* Hierarchical components, as illustrated in figure 18.10, are a very powerful tool. Here a component is structured internally as a composition of other component instances. This allows a recursive and hierarchical decomposition of a system to be supported. Ports define how components may be connected: a port has an optional protocol definition that allows for port compatibility checks that go beyond simple interface equality. While this approach is powerful, it is also non-trivial, since it blurs the formerly clear distinction between Type and Composition viewpoints.

*Structuring* Finally, it is often necessary to provide additional means of structuring complex systems. The terms *busi-*

.

*ness component* or *subsystem* are often used. Such a higher-level structure consists of a set of components (and related artifacts such as interfaces and data types). Optionally, constraints define which kinds of components may be contained in a specific kind of higher-level structure. For example, you might want to define a business component to always consist of exactly one facade component and any number of domain components.

1. - [↑](#footnote-ref-1)
2. . [↑](#footnote-ref-2)
3. [↑](#footnote-ref-3)
4. . [↑](#footnote-ref-4)
5. ) [↑](#footnote-ref-5)
6. e [↑](#footnote-ref-6)
7. 4 [↑](#footnote-ref-7)