Documenting Software Architectures: Recommendations for Industrial Practice

David Garlan and João Pedro Sousa

October 2000
CMU-CS-00-169

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

An important issue for software system development is the documentation of architecture designs. In this report, we describe techniques for the architectural documentation of software-based systems in the context of development processes that use UML for software design. The architectural documentation is organized in four kinds of views: problem domain view, code view, run-time view and deployment view. We examine JavaPhone™ as a case study to illustrate the approach: what kinds of information are provided in each kind of view, what forms of notation should be used, what are their limitations, and what uses can be made of this documentation.

This material is based upon research sponsored by the DaimlerChrysler Corporation and by the Defense Advanced Research Projects Agency (DARPA) supported by the Air Force Research Laboratory under Contracts No. F30602-00-2-0616 and N66001-99-2-8918. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of DaimlerChrysler Corporation, DARPA, or the United States Air Force.

Keywords: software architecture, documentation, software frameworks, software architecture representation, component integration standards, JavaPhone, UML, formal specification.

1 Introduction

It is becoming widely recognized that well-designed software architecture is critical to the success of any complex software-related project. By exposing the key system design concerns, a properly designed architecture goes a long way towards guaranteeing that a system will satisfy its principal requirements and it helps insure system integrity as the system evolves over time.

Moreover, increasingly software-intensive systems are being developed by building on existing architectural frameworks – both those available as external standards, as well proprietary, domain-specific product-line architectures developed within a company. The cost savings that accrue from reusing such existing frameworks (and their associated implementations) is revolutionizing the world of software, which not long ago developed most of its software from scratch for each new project.

In order for an architectural design to be useful one must first be able to write it down so others can understand the design, build from it, analyze its properties, and maintain it. Such architectural documentation then becomes one of the principal artifacts of a software project, providing the basis for design reviews, implementation guidance, system evolution, and even testing.

But what exactly is the architecture of a system and how should one document it? In the past there has been no good answer to this question. Indeed, the issue of architectural description has been a major focus of research on software architecture over the past decade. During this time industry has had to make do with makeshift informal box-and-line descriptions, ad hoc adaptations of notations not originally intended for the purpose of architectural documentation, and out-of-date methods.

Thankfully over the past few years, considerable progress has been made. The essential issues of architectural design have become much clearer, as have the needs for documenting them effectively. Moreover a number of standard design notations have begun to emerge – particularly in the area of object-oriented and component-based design. While there is still no universally-accepted notational standard for architecture, it is now possible to provide basic guidance about minimal requirements and useful techniques for architectural documentation.

Of course the detailed nature of documentation for a particular system remains strongly influenced both by the needs to which it will be put, the organizational context in which it is to be used, as well as the resources a project is prepared to devote to the effort. Hence any recommendations for architectural documentation must be sensitive to local variation.

In this report we attempt to provide some guidance for architectural documentation in context of DaimlerChrysler's software-based systems and software development processes. We will do this by examining a case study to illustrate how one can go about documenting an architecture: what kinds of information must be provided, what forms of notation should be used, what uses can be made of that documentation.

The scope of this report is to provide a brief, although hopefully useful, guide to architectural documentation. In the brief period that we had to work through a case study, and to understand the needs of DaimlerChrysler, we could only provide an overview of the main features of the area. Additional elaboration would certainly be possible via a more extensive project. (See Section 6, which outlines possible future work.) Moreover, we confine ourselves to describing approaches consistent with current industrial practice of design documentation – most notably the use of the UML object modeling language. While we believe there are numerous opportunities to do much better using a new breed of architectural description languages and tools, space and time limitations of this report do not permit us to explore these avenues here – although we do give a brief indication of the kinds of benefits that can be realized using a more rigorous architectural specification language in Section 5.

The structure of this document is as follows: In the next two sections we briefly consider the purposes of software architecture, and the requirements for architectural documentation. Next we introduce the case study, JavaPhone™ that we will use to illustrate both the pitfalls of current documentation, and guidelines for improving the situation. Then we provide example documentation for four views of that system. Finally, we conclude by stressing the key points we have tried to make, and discuss possible future projects.

2 What is Software Architecture?

Before illustrating how to document an architecture, we must be clear about what an architecture is intended to describe. Considerable confusion can arise when the developers of an architecture are unsure about the essential purpose of architecture, and what should be exposed through architectural documentation.

While there are numerous definitions of software architecture, at the core of all of them is the notion that the architecture of a system defines its gross structure. This structure illuminates the top-level design decisions, including things such as how the system is composed of interacting parts, where are the main pathways of interaction, and what are the key properties of the parts. Additionally, an architectural description includes sufficient information to allow high-level analysis and critical appraisal.

By providing an abstract description of a system, the architecture exposes certain properties, while hiding others. Ideally this representation provides an intellectually tractable guide to the overall system, permits designers to reason about the ability of a system to satisfy certain requirements, and suggests a blueprint for system construction and composition. For example, an architecture for a signal processing application might be constructed as a dataflow network in which the nodes read input streams of data, transform that data, and write to output streams. Designers might use this decomposition, together with estimated values for input data flows, computation costs, and buffering capacities, to reason about possible bottlenecks, resource requirements, and schedulability of the computations. The architecture might also suggest ways to package the functionality as code units.

3 Software Architecture Documentation

Architectural documentation describes the structure of a system through one or more views, each of which identifies a collection of high-level components and relations among those components. A *component* is usually documented visually as some sort of geometrical object, and represents a coherent unit of functionality. The granularity of components will depend on the kind of documentation being developed: in some situations a component may be as large as a major subsystem; in others it might be as small as a single object class. Typically components represent system structures such as major modules, computational elements, and run-time processes.

The *relations* between components are documented visually using lines or adjacency. Typically such relations indicate what aspects of one component are used by other components, and how inter-component communication proceeds over time.

Different *views* are used to represent distinct aspects of a system, each view providing a model of some aspect of the system.[1] For example, as we will see, one architectural view might document the structure of a system as a layered description in which the components represent logical groupings of code, while another view might document the structure of a system in terms of its run-time configuration in which components represent communicating processes.

Different views, or models, are useful for different purposes. Deciding which views to use is one of the chief jobs of a software architect. Often the choice of views will depend strongly on the needs for design analysis. For example, a system with real-time scheduling constraints might use one or more views that expose process/task boundaries and indicate various properties such as periodicity and resource requirements. A system that is more centered on shared data, might devote views to describing the structure of the data space and the ways different components access that space.

Despite this variability there are typically at least four classes of views that are required to provide a reasonable set of architectural documents.

1. **Context-based views:** These indicate the setting in which the system is to be employed, and often identify the abstract domain elements that determine the system's overall requirements and business context.
2. **Code-based views:** These describe the structure of the code, indicating how the system is built out of implementation artifacts, such as modules, tables, classes, etc. Such views are particularly useful as a guide to implementation and maintenance. They can also be used to indicate boundaries of abstraction between differ-

[1] We use the term "model" to refer to a partial, abstract description of a system.

ent parts of the system, and between the system-under-construction other parts of the system that it uses or that use it. A special, but common, case of a code-based view is a layered diagram. By partitioning a system into layers, one can improve portability, modifiability, and ease of use via standard APIs. As we will see, the case study below makes heavy use of layered diagrams. Another common code-based view is a class diagram.

3. **Run-time views:** These describe the structure of the system in operation, indicating what are the main run-time entities and how they communicate between each other. Run-time views allow one to reason about behavioral properties and "quality attributes," such as run-time resource consumption, performance, throughput, latencies, reliability, etc.
4. **Hardware-based views:** These describe the physical setting in which the system is to run, indicating the number and kinds of processors and communication links. The information contained in these views is often combined with that in run-time views to derive system performance properties.

Independent of choice of view classes and specific models within each view class, there are some important observations that one can make about architectural documentation in general.

- *Current informal practice is not adequate.* Most architectural descriptions are ambiguous, inconsistent, misleading, and ineffective. Often this occurs because the authors are not clear about the nature of the view they are describing, or about the meaning of the elements in the views. Later we will illustrate a number of such problems, and indicate specific advice about how to avoid them.
- *Architecture is not low-level programming structure.* Architectural documentation's most important requirement is to *clarify* system structure. This can only be done via careful use of abstractions – hiding inessential details. Many architectural documentation efforts run adrift in a sea of detail, where the important issues are buried in low-level specifications.
- *Multiple views are usually needed.* As we noted earlier, different models are needed to cover the spectrum of issues addressed by a system's architecture. Many documentation efforts fail by attempting to combine too many issues into a single description. Again, we will see examples later.
- *Good architectures make use of architectural styles.* An architectural style provides a vocabulary for design together with constraints on how that vocabulary can be used. For example, a simple client-server style might involve components that are instances of clients, servers, databases, and user interfaces. These might be constrained topologically or otherwise. Styles should be documented, so it is clear to the reader what assumptions are being made about the vocabulary and constraints.
- *Many architectural designs are actually architectural frameworks.* In some cases an architectural description represents a particular system. However, more often, an architecture is designed to cover a family of systems that differ along certain dimensions. (The case study below is a good example of a framework.) A key to documenting architectural frameworks is to be clear what kind of variability is allowed.

- *There are many possible notations that one might use.* A notation for architectural documentation should be evaluated for satisfaction of the following properties
    - *Expressiveness*: the notation should be rich enough to express the semantic distinctions and decisions of the architecture.
    - *Precision*: the notation should be clear enough that one can spot inconsistencies and bad design choices.
    - *Understandability*: the notation should make it easy to see what the key design decisions are.
    - *Analytical utility*: Ideally, the notation should be formally analyzable.

4 The Example: JavaPhone™

4.1 Background

JavaPhone evolved between 1996 and 1999 as a vertical extension to Sun Microsystems' PersonalJava™ platform [PJv00]. In addition to Sun, a number of companies in the telecommunications industry contributed to this definition, including Dialogic, Lucent, Nortel, Novell, IBM, Intel and Siemens. In 1999, Sun decided to promote JavaPhone as an industry standard, and new collaborations were enlisted: the ETSI (European Telecommunications Standards Institute), Ericsson, Motorola, Nokia, Psion, Texas Instruments and Symbian. Version 1.0 of the standard proposal was released on March 22, 2000 after a period of public review [JPh00].

The key motivation behind JavaPhone is to enable application developers to deploy telecommunication-aware applications on any physical platform that supports an API complying with the JavaPhone specification. Conversely, suppliers of telecommunication platforms can market their solutions taking advantage of existing value-added applications - just by obtaining the latter off-the-shelf and deploying them on their newly developed devices. JavaPhone is, therefore, a specification of an architectural *framework* from which concrete systems will be derived.

The documentation that describes the JavaPhone standard provides one (informal) specification for a layer of software that isolates telecommunication-aware applications from the specifics of telephony integration platforms (see Figure 1.)

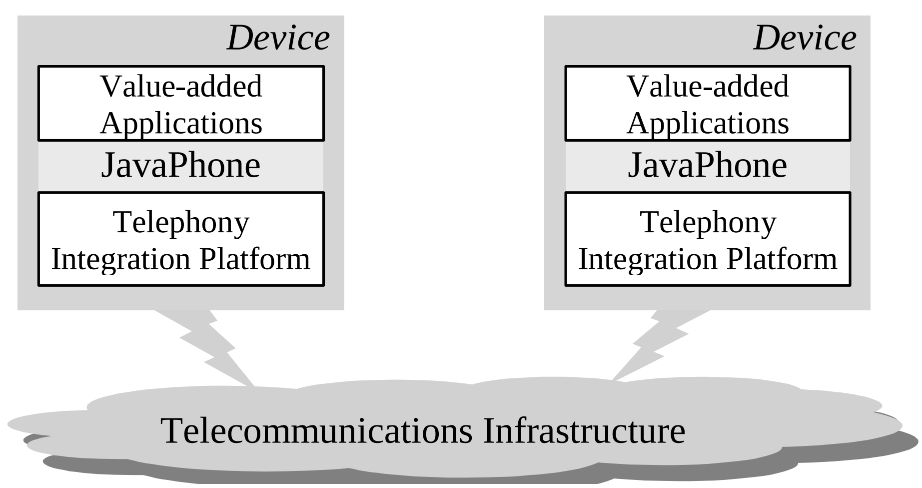

Figure 1: JavaPhone in a nutshell

4.2 Inside JavaPhone

JavaPhone is organized as a set of software packages, each supporting distinct, but complementary, functionality:

- Telephony, also known as **JTAPI** (Java Telephony API), composed of
  - **Core** package[2]
    basic functionality to make, answer and drop a call;
  - **Call Control** package
    extended features like conferencing and forwarding;
  - **Phone** package
    drivers for physical devices like phone buttons, display and ringer;
- **JTAPI Mobile** package
  mobile features like network selection and strength of signal;
- **Power Monitor** package
  estimation of available power;
- **Power Management** package
  management of device activity to minimize power consumption;
- **Network Datagram** package
  transport-independent addressing and delivery of messages;
- **Secure Sockets Layer** package
  secure communication over TCP/IP sockets;
- **Install** package
  version-aware installation of Java applications.

Leveraging the functionality commonly required in telecommunication devices, the standard also specifies interfaces for the following:

- **Address Book** package
  access to contact information on individuals and groups;
- **Calendar** package
  access to scheduling and task information;
- **User Profile** package
  access to the current user/owner's personal information.

A particular device is not required to support all of the packages above in order to be considered compliant with JavaPhone. Sun defines a number of possible subsets, or *pro-*

---

[2] In Sun's documentation the term JTAPI Core is sometimes used to represent the aggregation of the Core proper, Call Control and Phone packages.

files, for a device as consistent collections of supported packages.[3] Figure 2 reproduces the table from page 4 of [JPh00], which illustrates the wireless and screenphone profiles.

JavaPhone API	Wireless Profile	Screenphone Profile
Addressbook	Required	Required
User Profile	Required	Required
JTAPI Core	Required	Required
JTAPI Mobile	Required	Optional
Calendar	Required	Optional
Network Datagram	Required for connectionless transport	Optional
Power Monitor	Required	Optional
Power Management	Optional	Optional
Install	Optional	Optional

Figure 2: Device Profiles for Wireless and Screenphone

The architecture of JavaPhone is depicted by Sun primarily using variants of layered diagrams. Figure 3 and Figure 4 were taken from an online overview provided by Sun at http://web2.java.sun.com/products/javaphone/. In Figure 4 the purpose of the layers User Shell and Personal Java™ can be safely ignored for the purposes of this document.

[3] There are some dependencies among the packages offered by JavaPhone: for instance JTAPI Mobile depends of JTAPI Core in the sense that the former cannot be supported without the latter being supported as well. Generally, if package A depends on package B, a collection of packages supporting A is consistent only if it also supports B.

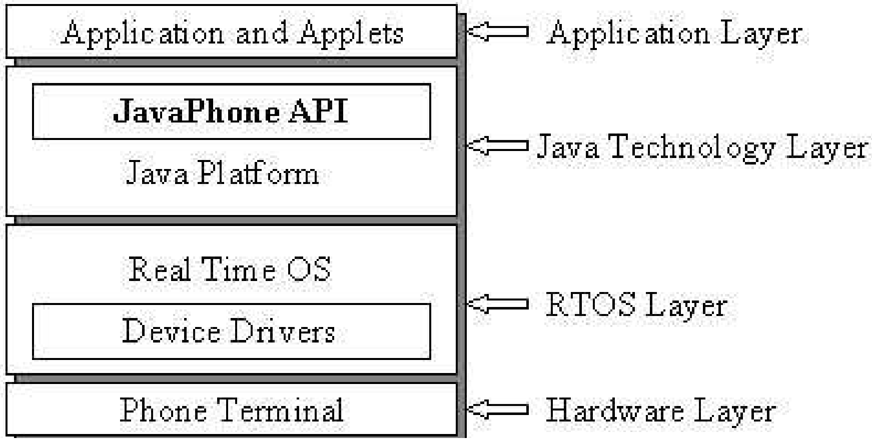

Figure 3: Layer diagram for a system incorporating JavaPhone

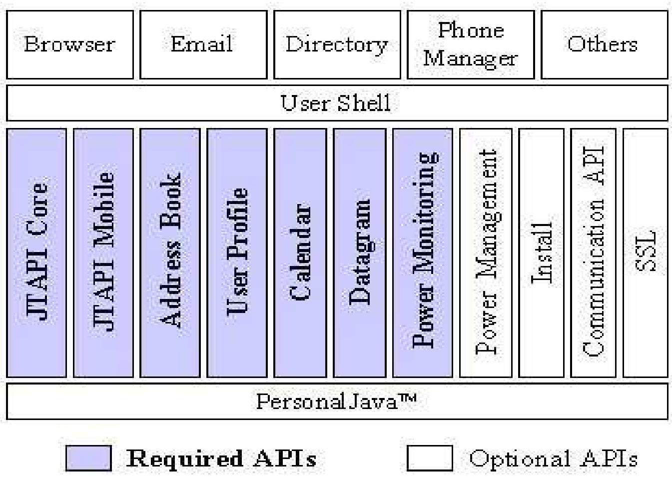

Figure 4: Detail of the JavaPhone layer, marking the wireless profile in color

One can identify a possible clustering of the software packages offered by JavaPhone. Take, for instance, the two packages pertaining to power management (Power Monitor and Power Management): there are relationships among the classes and interfaces defined by these two APIs, but they are isolated from the remaining packages in JavaPhone. The same is true for the software packages pertaining telephony. Given this natural association, and to simplify the diagrams showing the structure of JavaPhone without loss of consistency with Sun's definition, we will adopt the following clustering of packages:

- **Telephony**: JTAPI Core, Call Control, Phone and Mobile
- **Data Communication**: Network Datagram and Secure Sockets Layer
- **Application Management**: Install
- **Utilities**: Address Book, Calendar and User Profile

- **Power**: Power Monitor and Power Management.

4.3 JavaPhone as a case study for architecture representation

JavaPhone provides an adequate case study for architecture representation because, first, it describes an architectural framework – that is, a common set of architectural constraints to which the architecture of the concrete systems must comply. Second, it embodies the perspectives of a wide number of experts in the Software and Telecommunications industry and, third, the scope is manageable for the purposes of the case study. Furthermore, the uniformity of the patterns used in JavaPhone allows us to focus in the analysis of only the Telephony cluster and still cover the same interesting points from the architectural perspective that could be extracted from an exhaustive analysis of the case study.

There are, however, a number of drawbacks concerning the choice of JavaPhone as a case study for architecture representation. First, the JavaPhone framework describes the behavior of just one layer of software with few interactions among the components of that layer. Second, the relationships among components are only those common in the object-oriented approach: inheritance, method calling and event announcing. Third, the specification of the framework is not concerned with properties and tradeoffs that an architecture usually addresses, for instance:

- performance,
- reliability,
- tradeoffs between accuracy/fidelity of computation and response time or power consumption.

The somewhat narrow scope of the architectural issues brought up by the case study limits the scope of the recommendations concerning architecture representation that we can offer based on the current research.

5 Documenting the Architecture of JavaPhone

As we discussed in Section 2, we will describe JavaPhone using representations that are grouped into the following views:[4]

1. *Problem domain view:* Although software architecture is not about the representation of requirements, it is concerned with how the system to be built sits in the structure of the problem and how the system interacts with its environment. This issue has been approached from the structural and phenomenological perspective by Michael Jackson, and in a more directed use case perspective by Ivar Jacobson [Jack95, Jac+92].

[4] See [Kruc95, Ogu+00, You+99] for other examples based on views.

2. *Run-time view:* This view describes the principal run-time components, their interactions, and their properties. This view is the realm that Architecture Description Languages traditionally address [SG95, BCK98]. It is concerned with characterizing the run-time structure of a system, including the behavior of components, interaction protocols supported by connectors, the location of process boundaries, and analytical models for properties like throughput and reliability.

3. *Code view:* This view can be thought of as the link to software design – where one stops and the other starts is mostly a judgment call, based on how far the software architect wants to carry the abstract representations of code structure and behavior. Exactly which representations (layer diagram, class diagram, etc.) are most helpful to realize the code view depends on the architectural style of the system [SG95, RJB98].

4. *Deployment view:* This view describes the allocation of software to the deployment infrastructure. This is an area with relatively little attention from the research community [RJB98]. It is, however, essential for reasoning about properties like throughput and reliability [BCK98].

We will present the architectural representation of JavaPhone using each of the views above. The order in which we will do so is what seemed to us to be the natural order of explanation of JavaPhone, and should not be taken as an order of relative importance of the views.

For each view, first, we examine the aspects of the system that to be captured. Second, we identify a set of candidate representation techniques to realize the view. Since the ability to communicate effectively with the software engineering community is no less important than the ability to reason about the properties of a software system, the choice of the representation techniques that realize each of the views is targeted, as much as possible, to UML. Third, we examine how each of the representation techniques is used is current practice, namely in the JavaPhone specification, identifying underlying principles and potential problems. That insight will be used to evaluate the candidate documentation techniques with respect to the principles and aspects that are intended to be captured by the view.

5.1 Problem domain view

The goal of a problem domain view is to clarify the purpose, fit, and limitations of the system with respect to its environment. Also, this view has been found useful to identify commonly occurring patterns in the problem description – there are likely to exist proven solutions that will be cheaper to incorporate in the final system than solution rediscovery. As such, this view is concerned with:

- Principal parts (domains) of the problem space.
- Phenomena that occur in each domain.
- Sharing of phenomena (interactions) among domains.
- Identification of the domains corresponding to the system to be built.

- Framing of known configurations in the problem space.

In some application domains, such as in the development of safety critical software, people have successfully used formal notations to specify the problem space. However, for most systems a purely formal approach is frequently neither practical nor cost effective. At the other end of the spectrum in practice problem domains are typically approached using imprecise representation techniques, such as informal diagrams accompanied by explanatory prose.

Jackson's problem frame approach lives between these extremes: it strives for precision and effective communication without requiring strong formal skills. The work on use cases started by Jacobson (see [Jac+92]) and continued in the UML setting provides a nice complement of Jackson's approach for the description of typical interactions between domains [RJB98]. We note, however, that this kind of description, although appropriate to obtain a first cut of the system, or to discuss with users, does not deliver the full potential of the more formal notations used to express the architecture of software as presented in the following sections.

Although we cannot provide a complete explanation of Jackson's approach here, we will illustrate the use of domain diagrams for JavaPhone to indicate the basic principles and provide further insight into JavaPhone.

Figure 5 depicts the context of operation of JavaPhone, much as did Figure 1. Here the semantic primitives are the *Domain*, represented by a box, and the indication of *sharing of phenomena* between domains, represented by a line between the domains. There is no special meaning to the thickness or color of lines, unless noted. A domain may itself be decomposed into interacting sub-domains, represented as a diagram enclosed inside the box for the original domain.

In Figure 5, we identify three replicas of the Device domain interacting with a Telecommunications Infrastructure domain. Devices contain an Applications domain, a Telephony Integration Platform and may contain a JavaPhone domain that sits between the two former. The Telephony Integration Platform is always responsible for the interactions with the Telecommunications Infrastructure and represents the device's specific hardware and software dedicated to telecommunications. Applications are unaware of the details of the Telephony Integration Platform when JavaPhone sits between them. Of course, we are interested in the structure and behavior of devices that include JavaPhone, although they will often coexist with other devices that do not include JavaPhone.

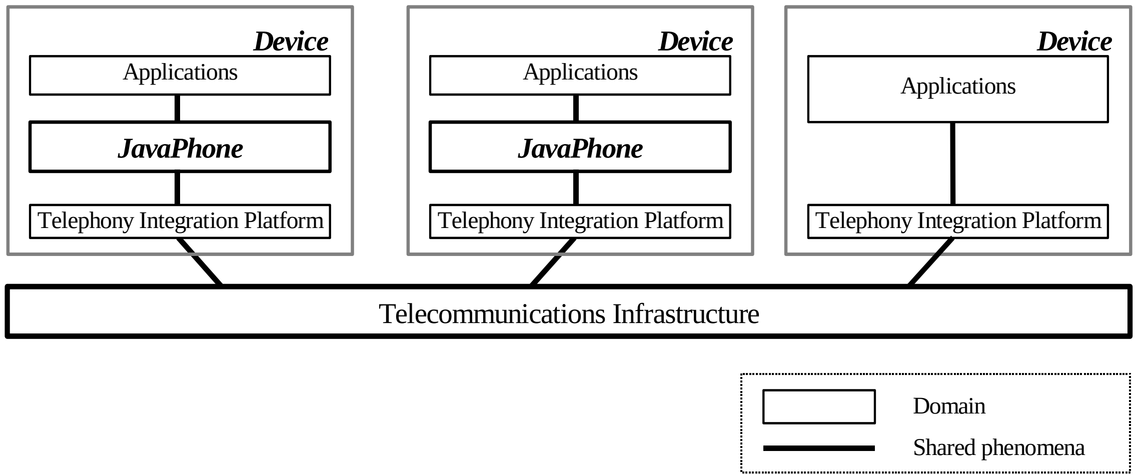

Figure 5: Problem domain of JavaPhone

Figure 6 focuses on a single device and "frames" the kind of problem that we are facing as an example of one of Jackson's dozen predefined problem frame. Specifically, JavaPhone achieves a *connection*, or adaptation, between telecommunications-aware applications and the telecommunications platform realized in the device. In Jackson's terminology this becomes an instance of the "Connection Frame". Jackson identified several other problem frames (Information System, Control, Workpieces, etc.). Thinking about the frame(s) that is (are) applicable to the problem at hand helps identify the dominant characteristics of the problem and, ideally, helps direct the system's architect to known architectural patterns. Again, the architectural notations that we cover in the next sections flesh out this first cut at the system's structure and mechanics.

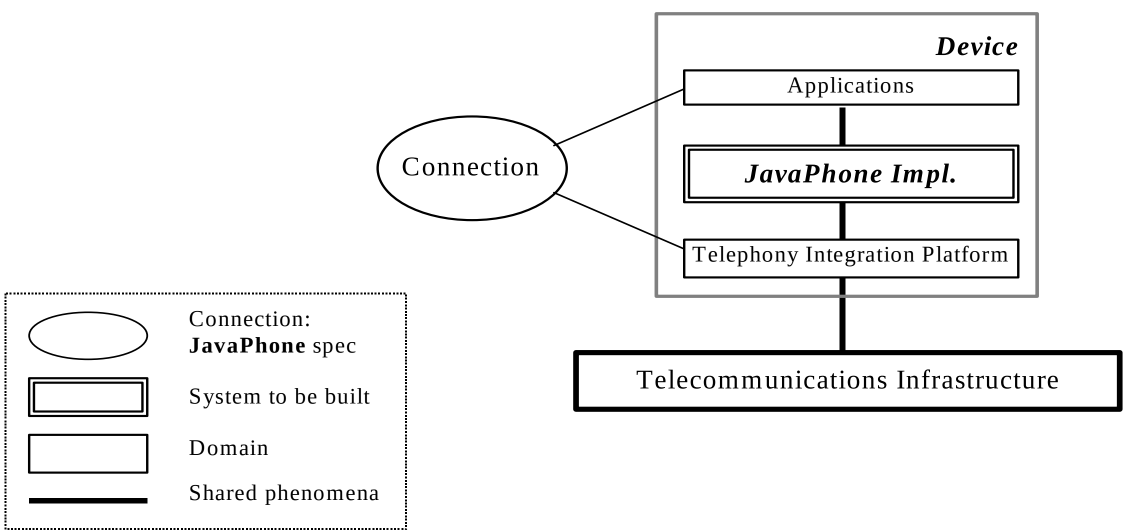

Figure 6: Framing JavaPhone – connection frame

Figure 7 focuses on the internal structure of JavaPhone, where we recognize sub-domains corresponding to the API grouping that we established in Section 4.2. Note that there is no sharing of phenomena among these sub-domains of JavaPhone.

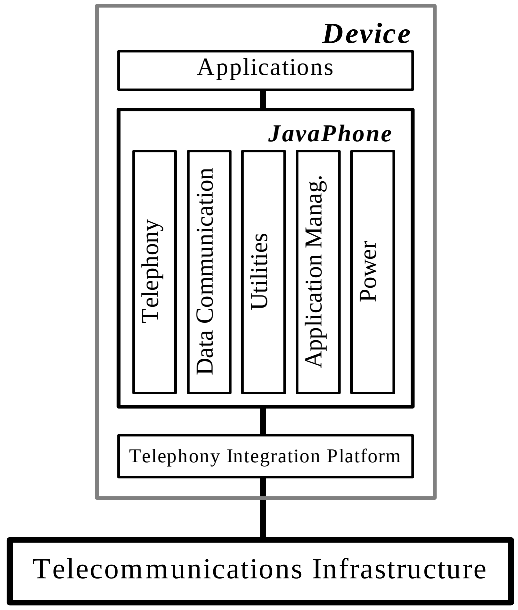

Figure 7: Detailing the domain view of JavaPhone

Figure 8 shows a more interesting diagram: the detail of the Telephony domain. The telephony Core domain describes the relationships between Addresses (e.g. phone numbers), Terminals (e.g. phones), the abstraction for the telecommunication Providers (to whom calls are requested), and the Calls themselves.[5] Also note that whereas the Call-Control and Mobile domains interact with the whole Core domain (that is, potentially with every sub-domain inside the Core), the Phone domain interacts specifically with the Terminals sub-domain of the Core.

[5] Sections 5.2.2 and Figure 16 contain more details on the internals of the Telephony domain.

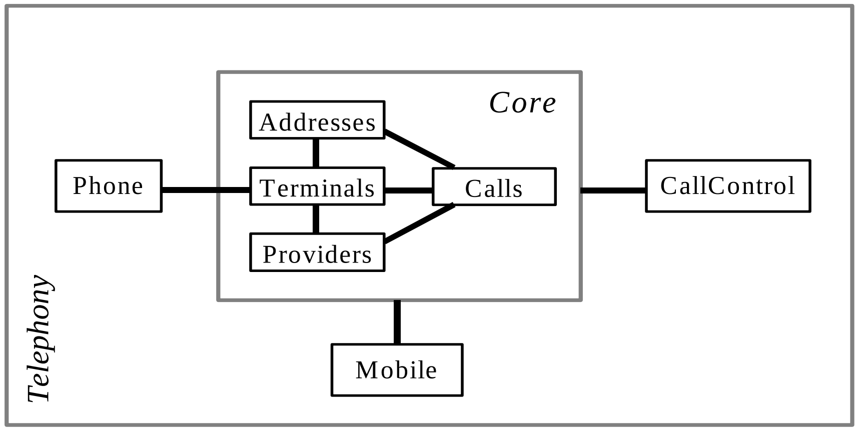

Figure 8: Detailing the Telephony domain of JavaPhone

How can diagrams like the ones in Figure 5 through Figure 8 be supported in a UML setting? We examine two alternatives: package diagrams and class diagrams. Figure 9 shows a package diagram corresponding to the domain diagram in Figure 8. As shown, domains are represented as packages and domain interaction as dependency.

While this serves to capture the main elements of the domain and the existence of relations between them, there are several warnings that must be heeded. First, one must be careful not to interpret the problem space decomposition (into domains) as a commitment to code organization: one domain may end up implemented as several code packages and vice-versa. Second, domain interaction is intentionally neutral with respect to directionality and final representation of the interactions themselves, whereas package dependency is directional and has defined semantics for access to services and configuration control. In other words, one should make sure that a package diagram representing the problem domain is interpreted as a problem domain view, rather than a package diagram.

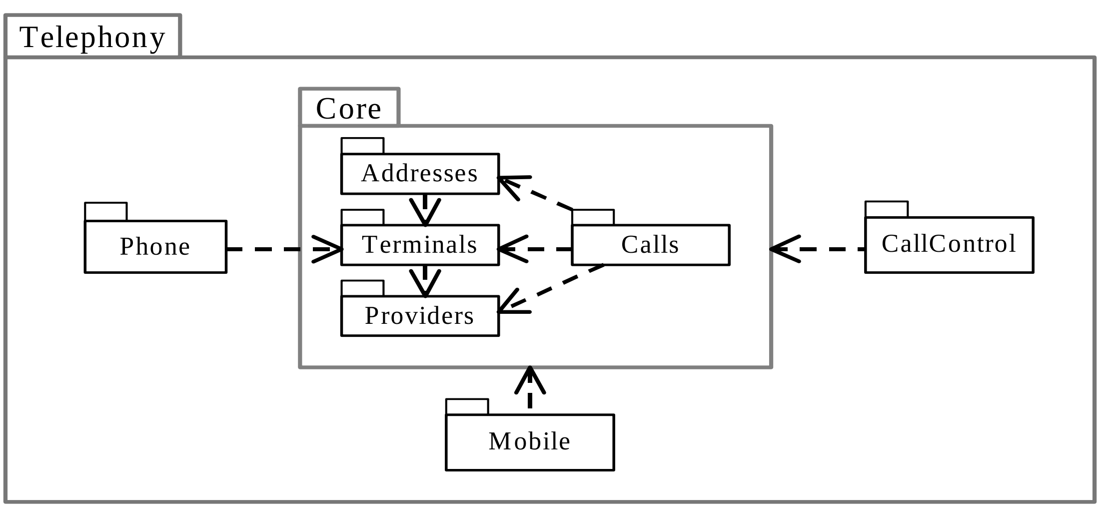

Figure 9: Package diagram representing the domain view of Telephony

Taking an alternative approach, Figure 10 shows the class diagram corresponding to the domain diagram in Figure 8. Domains are represented as object classes and domain interaction as associations. If we wish to consistently represent every domain as an object class (and every interaction as an association), we can use the aggregation association to stand for domain decomposition: in Figure 10, the Core aggregates Addresses, Terminals, Providers and Calls.

As with the representation of domain diagrams by package diagrams, a few issues of potential miscommunication arise. First, a domain will probably not end up as an single object class in the final system. Second, class associations typically indicate multiplicity, unlike domain interactions – it is easy to imagine one using a tool supporting the UML notation to start specifying the multiplicity of associations in what is supposed to be a domain diagram.

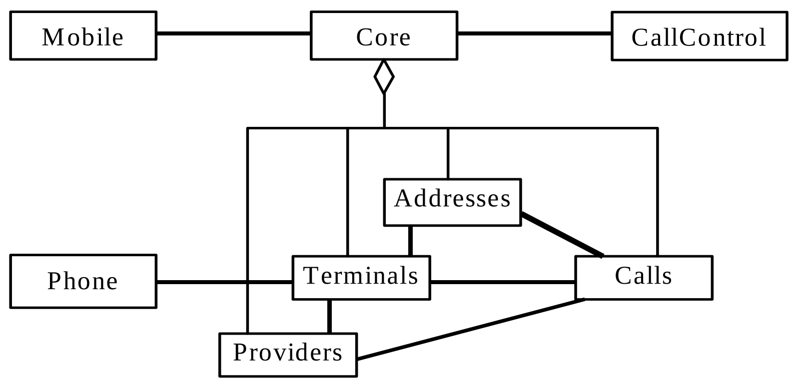

Figure 10: Class diagram representing the domain view of Telephony

5.2 Code views

Given the description of JavaPhone outlined above (Sections 4.1 and 4.2), we would naturally expect layer diagrams to be particularly relevant to represent the overall code structure. Also, since the JavaPhone specification is very much focused on the object-oriented style, class diagrams are the natural complementary way to capture the class-oriented organization of JavaPhone.[6]

We go over layer diagrams and class diagrams in the next two sub-sections. Although layer diagrams are often used, they are typically misused in ways that lead to miscommunication and even inconsistencies. We will examine a few examples of such problems in the documentation of JavaPhone itself and then show how we believe a layered view

[6] Strangely, however, Sun's documentation does not include any class diagrams, but only listings of the classes and interfaces that constitute the JavaPhone API. The relations between those classes and interfaces were obtained by us through examination of the definitions of the methods and the prose accompanying each interface.

should be presented. With respect to class diagrams, of course the situation is quite different: the fundamentals of class diagrams are relatively well-understood and explained by many other authors [RJB98]. As a matter of reference, however, we present a partial class diagram for the Telephony component.

5.2.1 Layer diagrams

Layer diagrams are typically used in situations where the structure of a system's code can be partitioned into a linear sequence of logical modules, with certain visibility restrictions between those modules. The restrictions allow one to improve portability and maintainability, since the limit the effect of changes to the code.

As for the problem domain view, the first step is to establish what aspects of an architecture a layer diagram should capture. We recommend that the following three items are paramount.

- The system and its surrounding environment.
- Dependency/isolation between layers.
- Interaction mechanisms between layers.

Usually dependency relations are represented by adjacency of layers and the relevant interaction mechanisms represented either graphically, by some sort of convention, or in textual notes. A key issue for understanding a given layer diagram is what is the nature of the access (or visibility) restrictions. In particular, can a given layer access *only* the services of the layer immediately below it? Or can it also access services of any layer at a lower level? Ideally one would hope to find a definitive, uniform answer to those questions. Unfortunately, in practice this is often not the case, as we will now illustrate.

Figure 11 reproduces Figure 2 appearing on Sun's page 7 of [JTAPI00].[7] What does adjacency mean? Suppose that we assume that a layer can only access services provided by the lower layer immediately adjacent. Then we would erroneously conclude that applications access the Java Run-Time layer only through the Java Telephony API (JTAPI part of JavaPhone).

On the other hand, if we assume that a layer can accesses, and therefore depend on, *all* of the layers below it, we would erroneously conclude that applications can depend on vendor specific telephony integration platforms, like Sun's XTL or Lucent's TSAPI.

Hence, for the diagram at hand, we must conclude that there is *no* consistent semantics for adjacency. This makes the diagram close to useless.

[7] The role of color in this figure in not clarified anywhere in Sun's documentation.

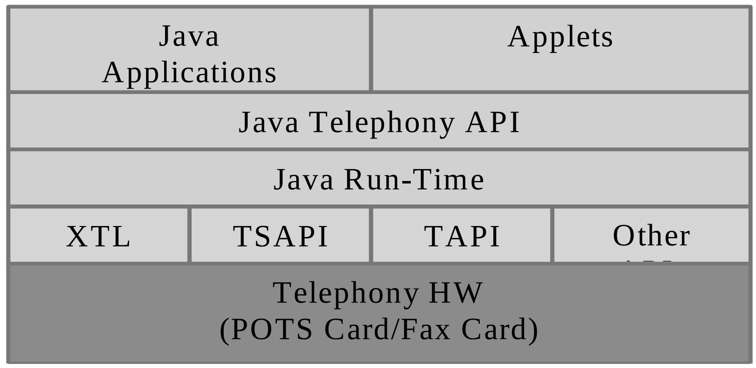

Figure 11: Layer diagram for the desktop configuration of JavaPhone

Take as another example Figure 12, that reproduces Figure 1 on page 7 of [JTAPI00]. It appears to be a layer diagram that combines aspects from the run-time view. Here, the arrows between layered boxes (presumably devices) denote some kind of remote access. Interpreting the diagram, one may be led to believe that the JTAPI (Java Telephony API) layer on the top box need not be aware of the specifics of the telephony integration platforms (like Sun's XTL or Lucent's TSAPI). Wrong again: in this case the Java Run Time and Remote Telephony Server layers serve as a low-level access mechanism between the layers for JavaPhone and the telephony integration platforms.

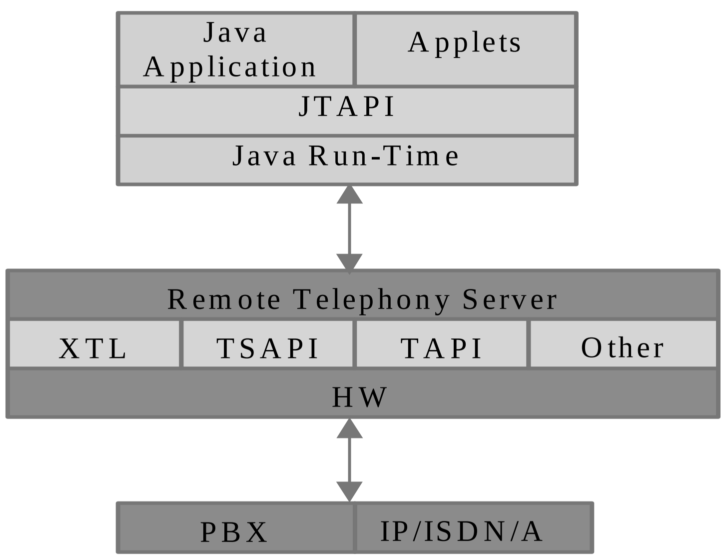

Figure 12: Layer diagram for the network configuration of JavaPhone

From examining examples like these, we can extract a set of principles to guide the documentation of architectures through layer diagram views:

a) represent only the layers that are relevant to the problem
b) a layer represents a virtual machine

c) a layer should depend only on layer(s) immediately below
d) indicate clearly the system to be built

Figure 13 shows a layered representation of the system and its environment as Figure 11 and Figure 12 attempted to capture it, but now according to principles (a) to (d) above.

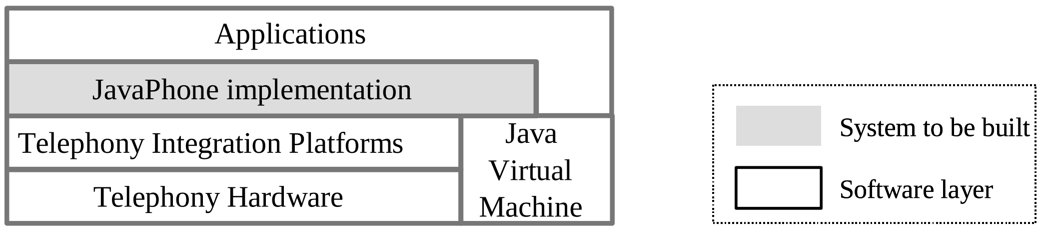

Figure 13: Top-level layer diagram for JavaPhone

One of the features of most layer diagrams is the use of substructure within a layer. Here again, current practice is typically ambiguous and inconsistent in the meaning of such decompositions. For example, the more detailed layer diagram, like the one on Figure 4, confronts the reader with questions like: "does JTAPI Mobile depend on JTAPI Core?" (it does) or "does Power Monitoring depend on Datagram?" (it doesn't) or "what are the interaction mechanisms between layers?"

One possibility to answer this kind of questions is to introduce explicit representation of software interaction mechanisms in code view diagrams. Although code views are, in general, not concerned with the interaction protocols, the interaction mechanisms represent abstractions that constrain the code itself.[8] Figure 14 shows one such diagram that we created, corresponding to the contents of Figure 4, but enriched by an indication of interaction mechanisms that otherwise must be painstakingly inferred from the textual specification of JavaPhone.

[8] For instance, code based on direct method invocation presents a different set of constraints than code based on indirect method calling (or event broadcast).

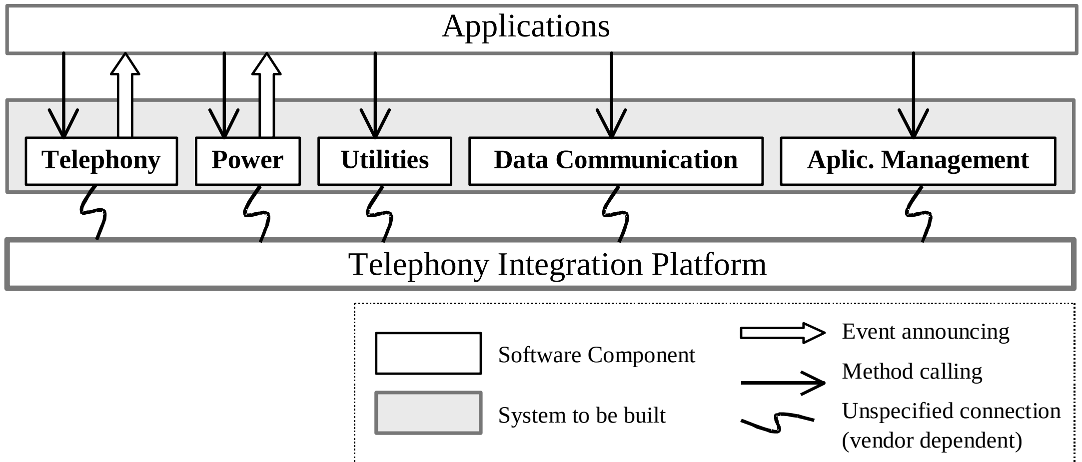

Figure 14: Detailed layer diagram for JavaPhone

Note, that in this diagram it is now much easier to infer important relationships. For instance, the Telephony and Power components both accept method calling and announce events to the Application layer. On the other hand, the Data Communication component does not announce events. A consequence is that applications must use polling to test for completion of data transmission and reception.

Now that we have illustrated (Figure 13 and Figure 14) the principles behind the layer diagrams and how these principles reflect the aspects that we expect to see captured in a layer diagram, the third step is to identify candidate representations in UML. An obvious candidate is, again, the UML package diagram. Next, we evaluate how well this kind of representation supports the principles for the layer diagram that were identified in the earlier.

The most serious potential problem with using package diagrams is a mismatch between what we are trying to express in a layer diagram and the usual semantics for UML package diagrams. For instance, people may take the depicted software components as a commitment to final code organization (in packages): although it seems useful to represent the JavaPhone layer as a package in the package diagram in Figure 15, there is no corresponding package definition in Sun's specification. Another example is the direction of the representation of the event announcing interaction mechanism between the Telephone and Power components and the Applications layer: we chose to keep the direction shown in the layer diagram in Figure 14. However, if we think in terms of UML dependency, we would expect to see the arrow in the opposite direction – it is the Applications layer that depends on the definitions of the events in the Telephony and Power components.

Another practical problem is the potential cluttering of visual information when several concepts are mapped into the same representation. For instance, from Figure 14 to Figure 15, the several interaction mechanisms end up being represented as stereotyped dependencies. It may also be harder to mark clearly which components belong in the system to be built and which are part of the environment (e.g. the Application package.) In Figure 15, the component corresponding to the system to be built is marked with the stereotype <<system>>.

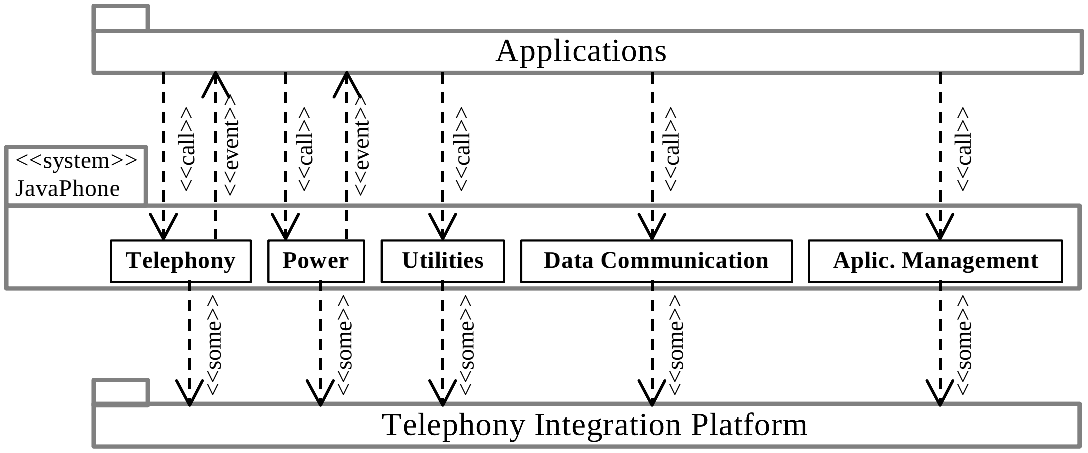

Figure 15: Package diagram corresponding to Figure 14

Still, we believe that using package diagrams to capture layer views of the code is by far the best alternative of the notations available in standard UML.

5.2.2 Class diagrams

Class diagrams are a natural way to capture the object-oriented organization of JavaPhone. Since the fundamentals of class diagrams are relatively well understood and explained by many authors [RJB98], we take the presentation of a class diagram for the Telephony component as an opportunity to explain the structure of JavaPhone in more detail. This insight into JavaPhone will be useful when we specify some of the interaction protocols in Section 5.3.

Figure 16 shows the principal interfaces of the Core package:

- Provider – abstraction for a provider of telecommunication services as delivered by the Telephony Integration Platform in Figure 13. If a concrete device has access to, say, a telephone connection and an IP channel, there will be two distinct objects that implement the Provider interface,[9] one with the characteristics of a phone service provider and the other with the characteristics of an IP service provider.
- Address – logical address of a telecommunication device, say, a phone number. In a concrete implementation of JavaPhone, there may be more than one Address object being managed by particular provider. The set of such Address objects are

[9] Whether or not these distinct objects are instances of the same class or of distinct classes is unspecified by JavaPhone and, as such, is a decision for the vendor of each implementation of JavaPhone.

included in the *domain* of the provider. For instance, the phone numbers are included in the domain of the provider of phone services and are returned by the operation Provider.getAddresses.

- Terminal – abstraction for a physical telecommunication endpoint. In a concrete implementation of JavaPhone, there may be more than one Terminal object being managed by a particular provider object. The set of such Terminal objects is included in the *domain* of the provider and is returned by the operation Provider.getTerminals. Addresses and Terminals within a provider's domain may be associated in a many-to-many fashion.
- Call – representation of an actual call requested by the telecommunications provider.
- Connection – there are as many Connection objects associated with a Call object as there are addresses involved in the call (normally two; more for conference calls).
- TerminalConnection – association between the (logical) connection and the physical endpoint that actually takes the call. (Recall that there may be more than one endpoint associated with an address.)

None of the interfaces above exposes constructor methods to the Applications layer – the management of the objects that implement such interfaces is made at the level of the concrete implementation of JavaPhone.

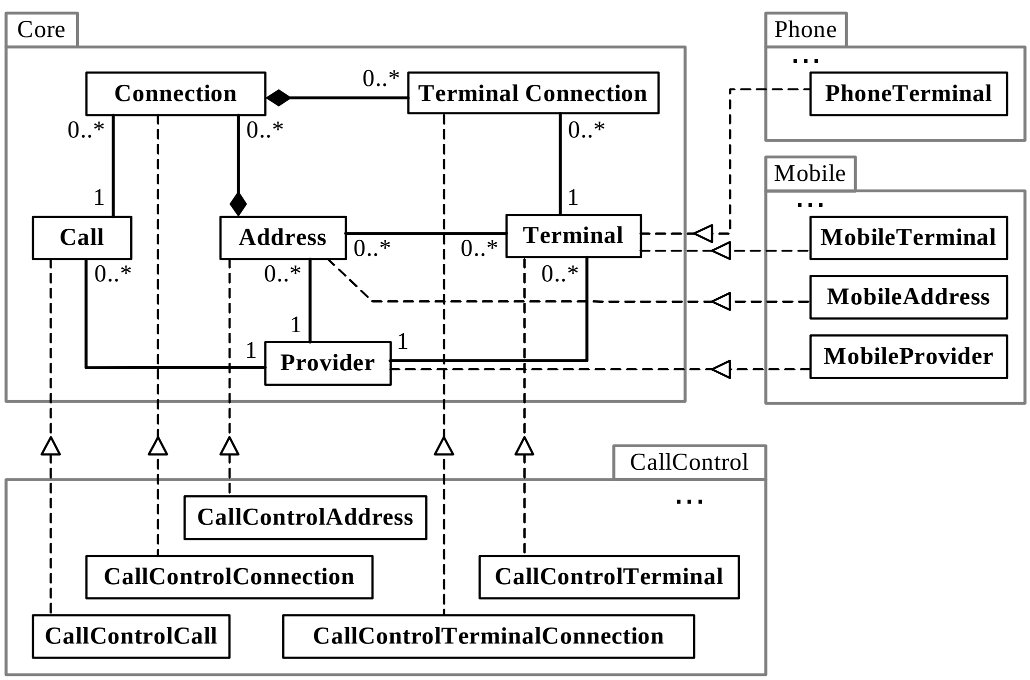

Figure 16: Class diagram for the JTAPI Core in the Telephony component

Recall that JavaPhone specifies an architectural framework and not the architecture of a concrete system. Therefore, it mostly specifies interfaces rather than prescribing actual implementations given by object classes.[10] Consequently, all the "classes" in Figure 16 actually refer to interfaces in the specification of JavaPhone and should have been affixed with the stereotype <<interface>>. We omitted the stereotype to reduce the cluttering of the diagram. A concrete system will have vendor-specific class definitions implementing each of the interfaces specified in JavaPhone.

From the Phone, Mobile and CallControl packages, Figure 16 shows only the interfaces that refine (or in Java terminology, *extend*) interfaces in the Core. Note that interface refinement is indeed the only kind of relationship among the several Telephony packages that is specified by JavaPhone.

JavaPhone follows the event-observer design pattern to realize the event announcing interaction mechanism depicted in Figure 14 (see Section "The Java Telephony Observer Model" in [JTAPI00]). As such, for each of the principal interfaces above there is an extra interface for the particular events that may be announced by the implementations of the principal interface, as well as another interface for a generic observer of such events. They will be implemented by as many object classes in the Applications layer as needed to be alerted by the occurrence of the events. It is the principal interface that defines the operations to register and remove an observer.

For instance, Figure 17 shows the definitions corresponding to the Terminal interface: there is an interface definition TerminalEvent and another TerminalObserver. Note the operations that are defined in each interface: the operation terminalChangedEvent in TerminalObserver is guaranteed to be called by the concrete implementation of JavaPhone, receiving a list of the events that occurred in the terminal since the last call. Note also how the class diagram fails to capture this important abstraction: the presence of the interaction mechanism has to be inferred from the explanatory prose and analysis of the names of some entities in the class diagram. If we had drawn every interface definition in Figure 16 it would be very hard to recognize both the structure of the core in terms of principal interfaces and their associations and even more so the presence of a well-defined interaction mechanism. Compare this with the clarity of Figure 14.

---

[10] There are very few object-classes (as opposed to interfaces) specified in JavaPhone, exceptions being associated with the factory design pattern. For instance, in the Telephony package, there is only one class, JtapiPeerFactory, that ultimately allows an application to obtain references for objects of vendor-specific classes that implement the Provider interface.

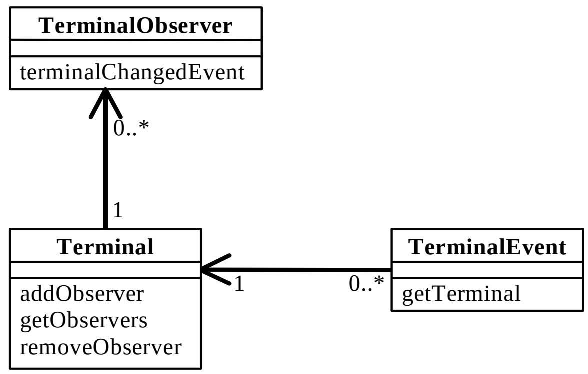

Figure 17: Class diagram for the interfaces defining the event-observer pattern for Terminal

5.3 Run-time view

Capturing the run-time view of a system is the main concern of Architecture Description Languages (ADLs), such as Acme and Wright [GMW00, All97]. The basic notions of ADLs are that of *component*, a cohesive locus of computation, and of *connector*, a description of the interactions between the attached components. Historically the paramount aspects captured by such run-time views have been:

- Architectural style – defines the types of components and their behavior, as well as the types of connectors and the interaction protocols they support. Examples of architectural style are: blackboard, pipe-and-filter, and object-oriented [SG95]. ADLs vary in the degree of formality of their descriptions of component behavior and interaction protocols, as well as in their support for definition and reuse of architectural styles.
- Configuration of the system – identifies the components that constitute the system and how they are interconnected. The identification of the possible process boundaries is also important since it constrains the interaction mechanisms that can be used to realize the connectors as well as the interaction protocols themselves (e.g. mutual exclusion, two-phase commit, etc.)
- Analytical models – support the analysis of tradeoffs for architectural properties of the given configuration [BCK98]. For example, such models enable the software architect to make informed tradeoffs between buffer size and throughput, code maintainability and performance, precision of computation and power consumption.

Architectural frameworks not only choose an architectural style, by identifying precisely the types of components and connectors, but also constrain the possible system configurations to particular topologies and the use of particular architectural properties.

JavaPhone specifies an API that stands between Applications and Telephony Integration Platforms (Figure 1), that is, it specifies both the generic topology of the system and the behavior of the connection between the two components (Figure 6). As Figure 11 and

Figure 12 suggest, there are possible variants in the way to realize the generic topology, but the behavior of the JavaPhone connection remains the same, regardless.

Like many architectural frameworks, JavaPhone leaves concerns about architectural properties to the specific implementations of the framework. Therefore, we cannot use JavaPhone to illustrate the definition and use of analytical models. The following subsections focus on the representation of, first the run-time view of the system configuration and, second details of the interactions among components.

5.3.1 System configuration

Figure 18 shows the run-time view corresponding to Figure 5. Recall that, since JavaPhone's concern is to represent an architectural framework rather than a concrete system, both figures illustrate the possible configurations of a device inside a system rather than a fixed system configuration with three devices.

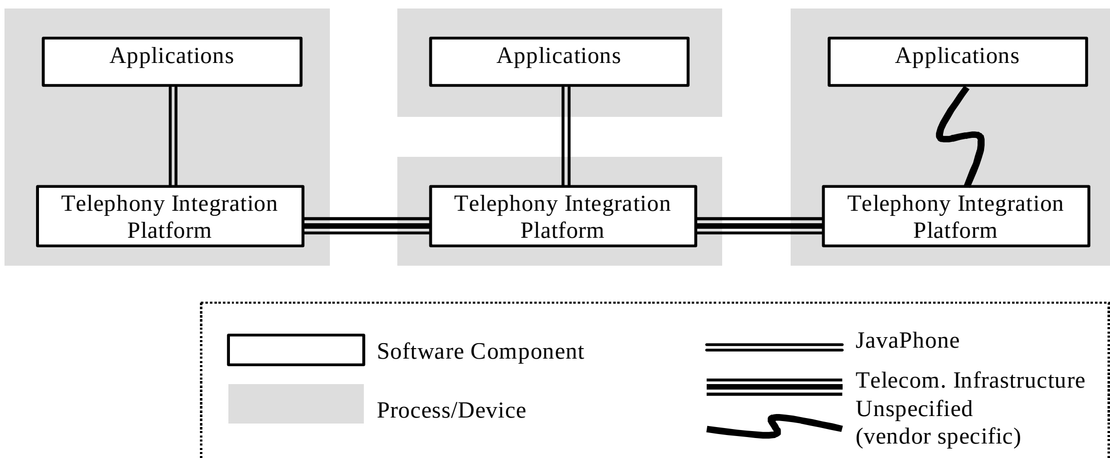

Figure 18: Possible device configurations for JavaPhone implementations

The choice between representing a particular piece of software as a component or as a connector in the run-time view is mostly one of usefulness of the abstraction.[11] The Applications and Telephony Integration Platform appear as two loci of computation, one containing the logic that serves the purposes of the user and the other wrapping the physical devices that are used in the communication. The Telecommunications Infrastructure is best represented as a connector, transporting data across between remote devices. As for JavaPhone, its concern is to assure that the Applications reach the services of the Telephony Integration Platforms in a way that is independent of the vendor-specific variations in which those services are delivered. JavaPhone is thus also represented as a connector.

[11] More on this can be found in [SG99].

Furthermore, the JavaPhone connector has to make sure the applications reach the telecommunication services seamlessly, regardless of being in the same processing context, Figure 18 left, or in a different processing context from the Telephony Integration Platform, Figure 18 center. Figure 18 right is shown just to illustrate that the devices that include JavaPhone are nevertheless able to communicate with devices that do not include JavaPhone, the difference being the connector (API) that the applications use to access the services of the Telephony Integration Platform.

Note how the run-time view separates the concerns of specifying the interaction protocols, in the JavaPhone connector, from defining the process boundaries. That the implementations of JavaPhone will have to deal with communication within the same process and across processes is a consequence of the architectural statements above. Note also that it is not an issue for the architectural framework *how* the implementations will deal with the problem (or even if the same implementation should handle the two situations, or, if each situation requires a different implementation of the same interaction protocol).

Let us now focus on the candidate representation techniques to capture the run-time view of a system or architectural framework more precisely. Architecture Description Languages and the tools that support them are obvious candidates. Figure 19 shows the Acme representation corresponding to Figure 18.[12] The most obvious advantage of this representation is that the boxes and lines in it stand exactly for what we expect them to: components and connectors. Acme provides placeholders for the description of the behavior of components and of the interaction protocols supported by connectors.

The most important advantage of this representation is that one can write formal descriptions for the behaviors and protocols and then use appropriate tools to check properties of interest. In [AGI98] we used this kind of architectural representation to examine the soundness of a protocol for distributed simulation and were able to detect several flaws and potential sources of trouble. In [SG99] we did the same for another architectural framework: Enterprise JavaBeans™, from Sun MicroSystems, and detected a potential source of protocol deadlock.

This kind of architectural representation is also useful during system maintenance or reconfiguration. For instance, upon a change of the internal behavior of a component, we can check whether the new behavior is still compatible with the interaction protocol, besides rechecking the whole system for preservation of the desired responses.

---

[12] The legend with Acme's symbology is not part of an Acme diagram and is shown here just for clarity. A detailed explanation of Acme can be found in [GMW00].

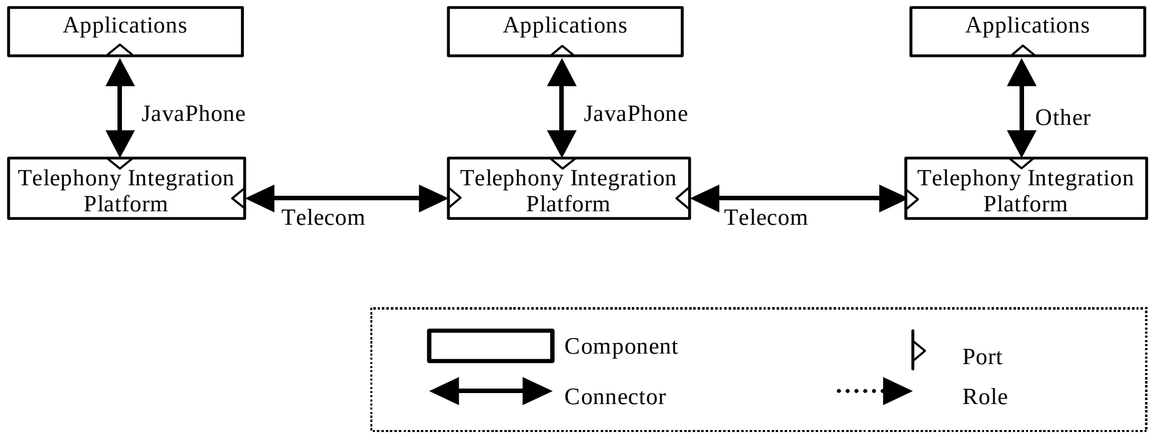

Figure 19: Acme representation of the JavaPhone configurations

Finally, note that there is no explicit way to represent the process boundaries, as such, in Acme. We can, of course, stretch the notation, as we have been doing systematically for UML, and represent each processing context as a component. Then, the part of the system inside each processing context would be represented as a subsystem using the same kind of notation as in Figure 19.

Figure 20 shows a representation of Figure 18 using a UML package diagram. Here, we adopt the convention of representing processing contexts as packages, components as classes and connectors as stereotyped associations. This representation, unlike an ADL proper, misses the support for the formal verification of properties.

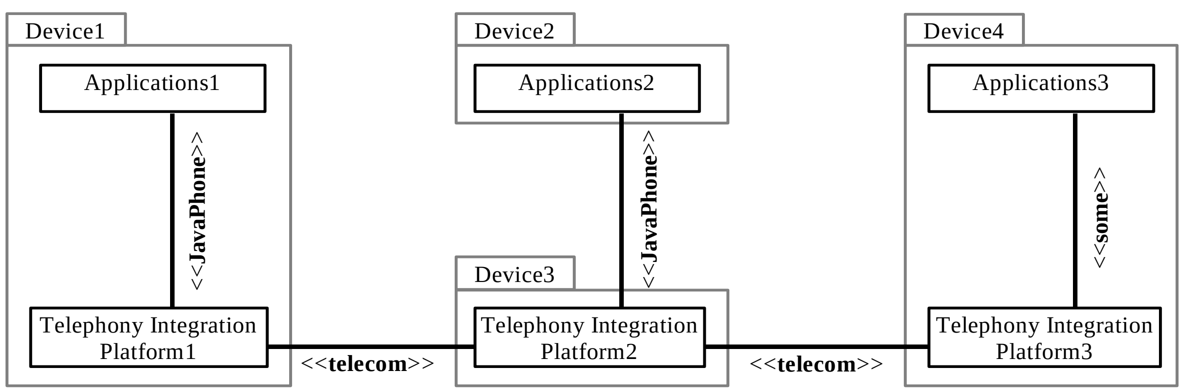

Figure 20: UML representation of the JavaPhone configurations

Besides that, we identified as a nuisance the fact that boxes in a package (or class) diagram are understood as the definition of a specification unit (a package or an object class) instead of as an occurrence of a processing unit (like a component, or an object instance by that matter.) Therefore, in order to represent several occurrences of the same component, we must name them differently, e.g. by numbering them. For instance, Applications1 to Applications3 in Figure 20.

So far, we have discussed the run-time view at a level of granularity where components correspond to code modules (or, in Java terminology, packages or collections of packages.) These are well identified at compile-time and remain stable during program execution. However, sometimes an architecture or architectural framework needs to provide run-time descriptions of certain critical components at a smaller level of granularity. JavaPhone is no exception: it prescribes the behavior of interfaces that will be implemented by objects in a concrete system.

For instance, Figure 21 shows the run-time view of JavaPhone (actually, the Telephony Core) in the called device during a two way connection.[13] In the depicted example, there is one local address, a2, associated to two terminals, t2 and t3. From our perspective, the representation of the remote address, a1, needs only to be associated with the representation of the remote terminal that originated the call, t1, regardless of how many terminals are actually associated with a1 in the originating device. There are two logical connections: c2, corresponding to the near end, and c1, representing the remote end. The terminal connection objects ct1 to ct3 represent the state of the associations between the logical connection and each of the physical endpoints. In this case, if t3 picks the call, ct3 will go active and ct2 will go passive for the duration of the phone call. The lines in Figure 21 represent the run-time connections between the objects. Note that execution units corresponding to the calls and connections will complete and then disappear in response to the action of the users.

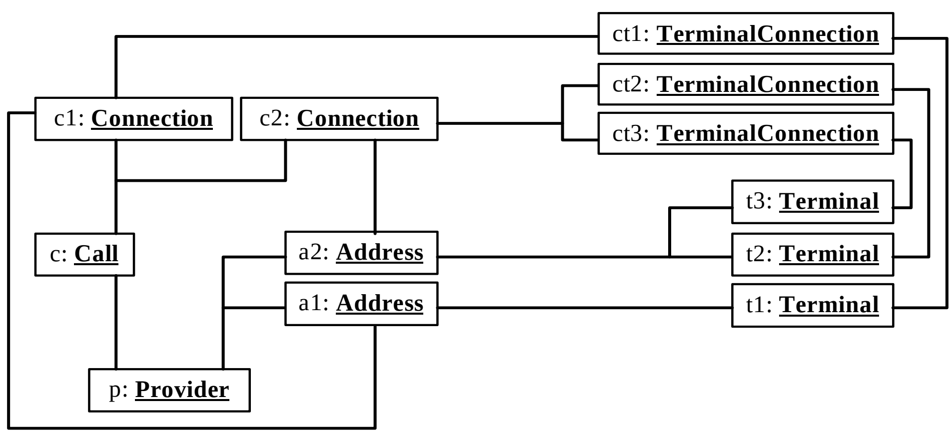

Figure 21: Run-time view of the objects that realize a two-way connection

Now, in the object-oriented architectural style, classes are explicitly identified at compile time and the rules for potential connection are given by the existence of associations (refer to Figure 16). However, the rules that determine which or how many execution units (objects) will be created to realize a given computation, or how they will actually be connected, is determined only during the execution of the methods at run-time. Neither

[13] Refer to Section 5.2.2 for a short explanation of the purpose of the interfaces in Telephony Core. Note that the types that qualify the objects in Figure 21 designate object-classes in a concrete implementation of JavaPhone that implement the corresponding interface definitions in the JavaPhone standard.

ADLs like Acme or Wright, nor any of the diagrams in UML capture this kind of dynamic run-time view. Therefore, the representation of run-time views should remain at a higher level than what Figure 21 attempts to capture, at least with respect to capturing the system configuration.

Nevertheless, an essential part of the run-time view is to capture how the execution units interact. For that, we often need to refer to finer grain units than those we can precisely capture as components in the run-time view. In the next section we give an example of how that can be achieved.

5.3.2 Interaction protocols

Figure 22 reproduces Figure 7 on page 16 of [JTAPI00]. It shows the same scenario as captured in Figure 21, now from the perspective of the state transitions of the objects involved in a two-way call. The time intervals on the vertical axis are of arbitrary length. To the left of the diagram, the vertical lines labeled Terminal 1 and Address stand for the objects representing the originating end of the call. To the right, the vertical lines labeled Address and Terminal 2/Terminal 3 stand for the objects representing the receiving end of the call. The horizontal lines between Call and Address, on either side, stand for the state transitions of the Connection object representing the originating end (to the left) and the receiving end (to the right). Likewise, the horizontal lines between Address and Terminal, on either side, stand for the state transitions of the TerminalConnection objects representing the originating end (to the left) and the receiving end (to the right). So, in timeframe 1, both Connection objects are in the *idle* state. Then the originating side goes to the *connected* state and the receiving side goes through the sequence of states: *inprogress*, *alerting*, *connected* and finally, *disconnected*. Note that on the receiving side, in timeframe 5, when one of the terminals, presumably Terminal 2, picks up the call, the corresponding TerminalConnection goes to the *active* state while the other goes to the *passive* state.

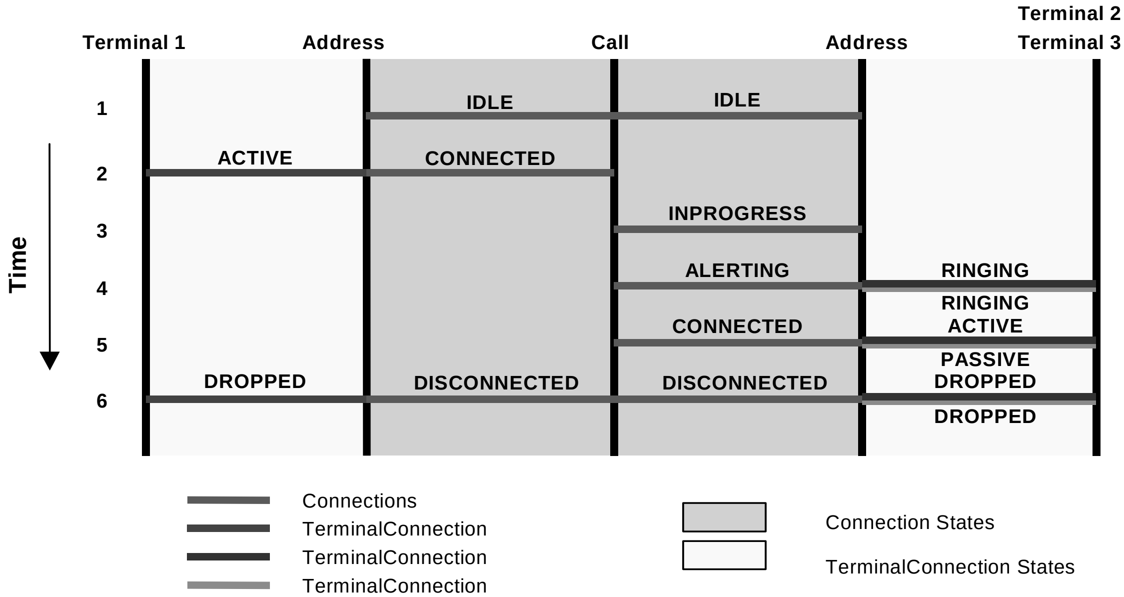

Figure 22: Timing diagram for the connection of a two-way call

Now, in JavaPhone both the originating device and the receiving device build a representation of the call using instances of Call, Connection, etc., as above. Are both representations identical? If not, to which end of the call does the diagram in Figure 22 correspond? The documentation provided by Sun is not clear about these questions.[14]

But more important than the misuse of a representation technique are the characteristics of the technique itself: Figure 22 captures the state transitions of the objects involved in a given computation, not the interactions that lead to those transitions. Often the latter perspective is more useful, especially in an object-oriented setting, where one object should not depend on the representation of the internal state of another, but rather on the operations (interface) it supports.

Sun documents the permissible protocols of interaction between an application and JavaPhone by supplying a sample code example. Figure 23 reproduces the main program of the code example in pages 18-19 of [JTAPI00]. This code illustrates an application that originates a phone call from address "4761111" to address "5551212” (these are just example addresses). The sequence of interactions captured in Figure 24 was built by examining the code example: once the application has a reference to a provider object, it obtains, from the provider, a reference to the Address object that represents the originating address. Then it calls the Address object to obtain a list of the terminals associated

[14] We believe it corresponds to the receiving end of the call, since the representation is aware of the existence of two terminals associated to the called address. From the originating perspective, the representation should not need to be aware of which, or even how many, terminals are associated with the called address until one of them picks up the call and becomes, from the caller’s perspective, the answering terminal.

with the originating address and it picks one of them arbitrarily. Next, the application asks the provider to create a Call object (which is created in the idle state). A new object is created of an application-defined class (named Observer in this example) that implements the interface CallObserver. This object is registered as an observer of the newly created Call object. Finally, the application asks the Call object to establish a connection between the originating address, using the (in this case arbitrarily) picked terminal, and the destination address (addr2). This request is directed to the Telephony Integration Platform that feeds the progress on establishing the connection back to JavaPhone. Such information is wrapped (by implementation-specific code within JavaPhone) as standard JavaPhone events and feedback to all the registered observers of the Call object.

Figure 25 shows the sequence diagram corresponding to the code example for an application that receives a phone call, in pages 21-22 of [JTAPI00]. First, the application calls the Provider object to obtain a reference for the Terminal object that represents the terminal to be monitored. That Terminal object supports a method that registers a (newly created) call observer. Such a call observer receives events pertaining to calls directed to an address associated with the monitored terminal. In the example, the Observer object receives a *ringing* event, signaling an incoming call. The Observer creates a TerminalConnection object within a new thread and requests it to answer the call.[15]

Both Figure 24 and Figure 25 are sequence diagrams as defined by the UML and are arguably more effective in illustrating one sequence of interactions than showing code snippets. However, questions remain such as: Is the illustrated order of requests to JavaPhone the only order permitted? Can the application interleave requests to other services in JavaPhone? The representation techniques that are available in UML capture only example scenarios of interaction – they are unable to express general rules for interaction protocols.

[15] The documentation provided by Sun does not clarify the purpose of creating the TerminalConnection object within a new thread, nor the consequences of not doing so.

```
import javax.telephony.*;
import javax.telephony.events.*;

public static final void main(String args[]) {

    /*
    * Create a provider by first obtaining the default implementation of
    * JTAPI and then the default provider of that implementation.
    */
    Provider myprovider = null;
    try {
        JtapiPeer peer = JtapiPeerFactory.getJtapiPeer(null);
        myprovider = peer.getProvider(null);
    } catch (Exception excp) {
        System.out.println("Can’t get Provider: " + excp.toString());
        System.exit(0);
    }

    /*
    * We need to get the appropriate objects associated with the
    * originating side of the telephone call. We ask the Address for a list
    * of Terminals on it and arbitrarily choose one.
    */
    Address origaddr = null;
    Terminal origterm = null;
    try {
        origaddr = myprovider.getAddress("4761111");

        /* Just get some Terminal on this Address */
        Terminal[] terminals = origaddr.getTerminals();
        if (terminals == null) {
            System.out.println("No Terminals on Address.");
            System.exit(0);
        }
        origterm = terminals[0];
    } catch (Exception excp) {
        // Handle exceptions;
    }

    /*
    * Create the telephone call object and add an observer.
    */
    Call mycall = null;
    try {
        mycall = myprovider.createCall();
        mycall.addObserver(new MyOutCallObserver());
    } catch (Exception excp) {
        // Handle exceptions
    }

    /*
    * Place the telephone call.
    */
    try {
        Connection c[] = mycall.connect(origterm, origaddr, "5551212");
    } catch (Exception excp) {
        // Handle all Exceptions
    }
}
```

Figure 23: Coding example for outgoing telephone call

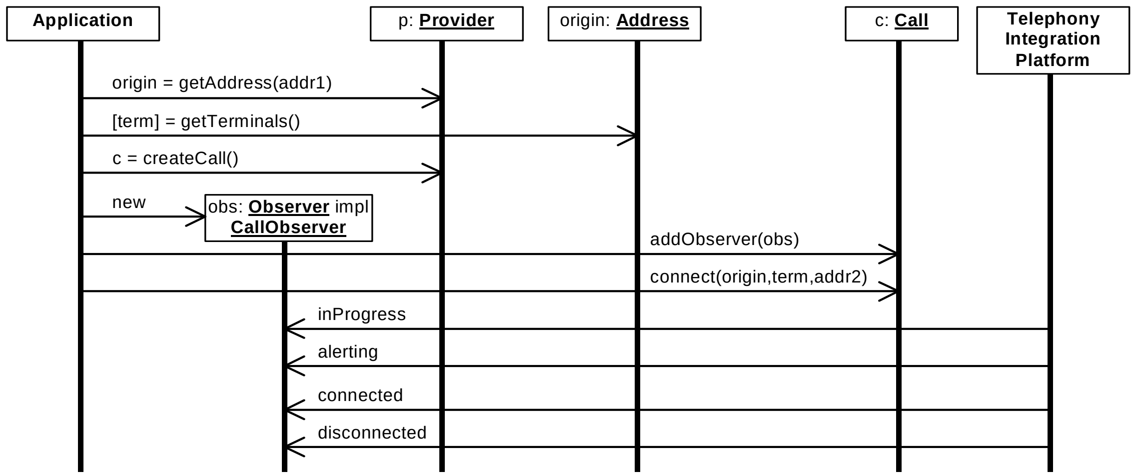

Figure 24: Application initiating a call between addresses *addr1* and *addr2*

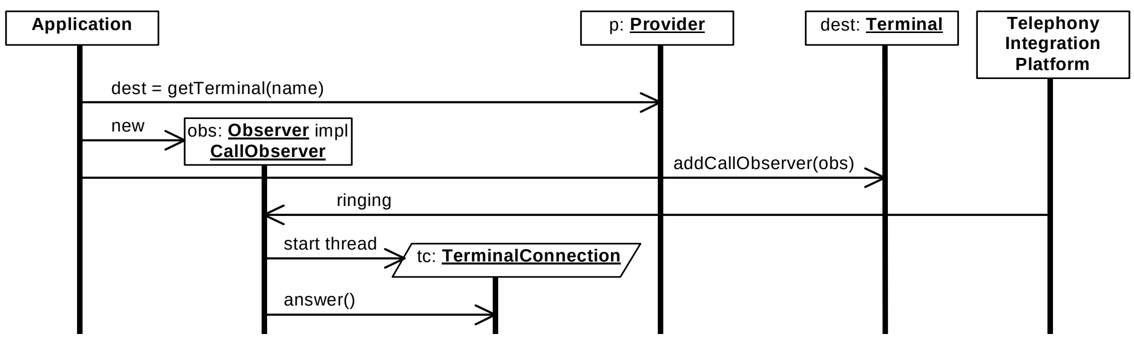

Figure 25: Application monitoring a terminal with id *name*

In contrast, consider the Wright ADL, which uses a variant of Hoare's CSP to describe the protocols that are supported by a connector [All97]. Figure 26 shows what a Wright description of JavaPhone as a connector would look like. Note that we chose to structure the interface that JavaPhone exposes to the application in such a way that it reflects JavaPhone's structure.[16] Thus, we recognize the roles Provider, Address, Terminal, Call, Connection and TerminalConnection corresponding to so many interfaces defined by JavaPhone. Furthermore, since Wright associates a behavior description with each role, we can capture the known rules for the behavior of each interface in JavaPhone. The description for the role Provider in Figure 26 is just an example: it allows the attached component to choose among any of the available services with no constraints of sequencing. The informal <...> is not part of the language and is used here to denote a piece of description that is not shown for brevity.

[16] The same principle can be used when describing the architecture of a final system and not just an architectural framework that defines a collection of interfaces.

Note also the definition of a role where the Telephony Integration Platform attaches, TelephonyIntgrPlatform and an undetermined number of identical roles for call observers, defined using Wright's parameterization mechanism. The "glue" defined for the JavaPhone connector describes the rules for the joint behavior of all the roles (corresponding to the interfaces exposed by JavaPhone) and thus defines the interaction protocols supported by Javaphone.

```
Connector JavaPhone

    Role Provider =  getAddress → Provider
                   □ createCall → Provider
                   □ <...>
    Role Address  = <...>
    Role Terminal = <...>
    Role Call     = <...>
    Role Connection = <...>
    Role TerminalConnection = <...>
    Role Observer (x:CallObserver) = <...>
    Role TelephonyIntgrPlatform = <...>

    Glue{Obs} =   Originate{Obs}
              □ Receive{Obs}
              □ Call.RemoveObserver(x) → Glue{Obs}-x

    Where Originate{Obs} =  Provider.getAddress
                         → Address.getTerminals
                         → Provider.createCall
                         → Call.addObserver(x)
                         → Call.connect → ObserveOrig{Obs}+x

    Where ObserveOrig{Obs}
          = TelephonyIntgrPlatform.inProgress
               → ‖ x:Obs ● Observer(x).inProgress; ObserveOrig{Obs}
          □ TelephonyIntgrPlatform.alerting
               → ‖ x:Obs ● Observer(x).alerting; ObserveOrig{Obs}
          □ TelephonyIntgrPlatform.connected
               → ‖ x:Obs ● Observer(x).connected; ObserveOrig{Obs}
          □ TelephonyIntgrPlatform.disconnected
               → ‖ x:Obs ● Observer(x).disconnected; Glue{Obs}


    Where Receive{Obs} = <...>
```

Figure 26: Example of JavaPhone protocols in Wright

For the reader interested in the technical details, we note that the definition of the glue process and its auxiliary definitions are parameterized with the set of call observers.[17] This set is reduced of x after the application invokes Call.RemoveObserver(x). In the auxiliary process definition that describes the behavior of JavaPhone while originating a call, $\text{Originate}_{\{\text{Obs}\}}$, we can recognize the same sequence of possible invocations that were captured in Figure 24. After the invocation of Call.Connect, JavaPhone observes the

[17] [SG99] contains an introduction to Wright that is sufficient to read through a specification like the one on Figure 26.

events originated in the Telephony Integration Platform, as described in the process definition ObserveOrig$_{\{Obs\}}$. After recognizing an event in the role TelephonyIntegrPlatform, the connector broadcasts the corresponding event to all registered observers: parallel triggering of a self-initiated event (underlined) in every role Observer(x), for every x in the set of registered observers.

So, what is the essential difference between a description like the one in Figure 26 and the sequence diagrams in Figure 24 and Figure 25, or any collaboration diagrams we might draw for the same scenario? The Wright specification does not define one example scenario. Rather, it defines exactly which call sequences are permitted and which are not; which permitted sequences can be interleaved and which cannot. Wright has precise semantics and the tools that support it enable running the specification through other tools like FDR [FDR92]. With FDR, we can explore the consequences of the specification, checking if desired behaviors hold, making sure there are no circumstances where the interaction protocols may deadlock.

We believe it is fine to use UML sequence diagrams or collaboration diagrams to represent typical or particularly important interaction sequences, and to use those representations for communication, especially during the early stages of development. However, these representations could be improved significantly using a more precise specification of the interactions, as argued above.

In the case studies that we have conducted so far, the limitations of architectural description as discussed with respect to Figure 21 were never a problem. As exemplified here, we were able to describe the interactions of execution units that are smaller than the defined components by capturing them as interacting structures inside the components and connectors (ports, roles, auxiliary process definitions) and eventually using Wright parameterization mechanisms to refer to some number of replicas of a given structure.

5.4 Deployment view

The deployment view adds infrastructure elements, both computing and communications, to the run-time view. The vocabulary of the deployment view includes the components and connectors identified in the run-time view: components now sit in computing structures and connectors that run across those structures are supported by communication media. The formal aspect of the description of behaviors and interaction protocols fades in this view, changing the focus of attention to the aspects related to analytical models. The latter are now added of physical properties like speed of computation, bandwidth, mean time between failures and available power. In heavily distributed applications, reasoning about the properties of topological alternatives may also be of interest.

The representation techniques that are commonly used for the deployment view are informal and provide little support for analytical models. Given that the deployment view adds to the run-time view, it seems important to keep a graphical consistency between the two, e.g., components should be represented the same way, since they *are* the same. An alternative is to build on top of whatever representation we choose for the run-time view, noting some of the components as being devices, and with the actual software components represented as the formers internal structure.

The Acme ADL supports placeholders for noting the components and connectors with attributes that can be used in the analytical models. Examples of the definition and use of analytical models are found in [BCK98]. Standard work on UML also has good examples of documenting this view, using UML deployment diagram notations [RJB98].

6 Summary and Future Work

In this report we have explored the needs for architectural documentation in the context of a particular architectural framework: JavaPhone. The primary aim was to indicate that it is possible to do a much better job of architectural description using notations that are clearly described (both graphically and textually), separating concerns (into separate views), and providing semantic guidelines for interpreting each documented view. We also looked at ways of using UML to provide documentation for the views. In each case we tried to point out the strengths, weaknesses, and pitfalls of using object notations like UML to do this [RJB98]. Finally, we gave a flavor of two languages for architecture description that are more directly aimed at expressing architectural concerns (Acme and Wright).

As a concluding comment, we should mention that many of these topics are closely related to on-going work at Carnegie Mellon's Software Engineering Institute and also in the School of Computer Science [Cle+01].

7 References

[AGI98] Robert Allen, David Garlan, and James Ivers, Formal Modeling and Analysis of the HLA Component Integration Standard, *Proceedings of the Sixth Intl. Symposium on the Foundations of Software Engineering* (FSE-6), Nov. 1998.

[All97] Robert Allen. A Formal Approach to Software Architecture. PhD thesis, CMU, School of Computer Science, January 1997. CMU/SCS Report CMU-CS-97-144.

[BCK98] Len Bass, Paul Clements and Rick Kazman, *Software Architecture in Practice*, Addison-Wesley, 1998.

[Cle+01] Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Robert Nord and Judy Stafford, *Software Architecture Documentation in Practice*, to be published by Addison Wesley, 2001.

[FDR92] *Failures Divergence Refinement: User Manual and Tutorial*, 1.2fi. Formal Systems (Europe) Ltd., Oxford, England, 1992.

[GMW00] David Garlan, Robert T. Monroe and David Wile. Acme: Architectural Description of Component-Based Systems, *Foundations of Component-Based Systems*, pages 47-68. Cambridge University Press, Gary T. Leavens and Murali Sitaraman, editors, 2000.

[Jac+92] Ivar Jacobson, Magnus Christerson, Patrik Jonsson and Gunnar Overgaard, *Object-Oriented Software engineering: a Use Case Driven Approach*, Addison-Wesley, 1992.

[Jack95] Michael Jackson, *Software Requirements & Specifications*, Addison-Wesley, 1995.

[JPh00] Sun Microsystems, *JavaPhone™ API Specification*, Version 1.0, March 22, 2000, http://web2.java.sun.com/products/javaphone/.

[JTAPI00] Sun Microsystems, *Java Telephony API Specification*, Version 1.0, March 22, 2000, http://web2.java.sun.com/products/javaphone/.

[Kruc95] Phillipe Kruchten, The 4+1 View Model of Architecture, *IEEE Software*, Vol. 12, No. 5, pages 42-50, November 1995.

[Ogu+00] Michael Ogush, Derek Coleman and Dorothea Beringer, *A Template for Documenting Software and Firmware Architectures*, Hewlett-Packard, http://www.architecture.external.hp.com/index.htm.

[PJv00] Sun Microsystems, *PersonalJava™ Application Environment Specification*, http://web2.java.sun.com/products/personaljava/.

[RJB98] James Rumbaugh, Ivar Jacobson and Grady Booch, *The Unified Modeling Language Reference Manual*, Addison-Wesley, 1998.

[SG95] Mary Shaw and David Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, 1996.

[SG99] João Pedro Sousa and David Garlan, Formal Modeling of the Enterprise JavaBeans™ Component Integration Framework, *Proceedings of FM'99 – World Congress on Formal Methods in the Development of Software Systems*. Lecture Notes in Computer Science, vol. 1709, pages 1281-1300. Springer Verlag, Wing, Woodcock and Davies, editors, 1999.

[You+99] R. Youngs et al, A Standard for Architecture Description, *IBM Systems Journal*, Vol. 38 No. 1, 1999.