

**Part I**  
**The Framework**

## About Part I

This first part explains the objectives of ODP and gives an initial introduction to the common concepts on which the framework is based. In particular, it describes the idea of viewpoints and how this idea is used to structure system specifications. It also introduces the taxonomy for supporting functions, which is used to establish a common vocabulary and, using human-computer interfacing as an example, indicates how planning for distribution interacts with other facets of system design.

Finally, notations are discussed, and a particular standardized profile for a UML-based notation is introduced. This notation will be used in the rest of the book.

Throughout the book, the problems being solved are put into context by a series of dramatized vignettes showing the everyday activities within the PhoneMob organization, a fictional company whose activities provide our running example. These are not, in general, essential to the technical points being made in each chapter, but you should certainly read the first episode, since it introduces the broad structure of the example to be used in the rest of the book.

# Chapter 1

---

## *What Is ODP About?*

Marcus Steinberg was a self-made man, and he had built his business on his ability to select a strong team. As he walked into the executive meeting room, he wished that events had not moved quite so quickly, but he would be a fool not to take advantage of the current state of the economy. He had snapped up the PhoneMob, a promising startup with a cash flow problem and a serious need to automate its working methods. Their business, providing a complete facilities management package to mobile phone owners, was close enough in concept to his current camera rescue empire for there to be some real opportunities for synergy, but PhoneMob would need sorting out first. Still, this acquisition would give him the platform he needed to make a real impact as the recovery kicked in.

Marcus knew the IT team he was about to address was technically solid, but he had begun to think that they needed a new approach. Recent projects seemed to have taken longer than expected, with too many last-minute snags and small misunderstandings. He liked a well-oiled machine, and their current performance grated. He had decided to call in some favours and had, as a result, been introduced to Alex Wren, a consulting enterprise architect whose new approach was, he had been told, finding favour in all the right places.

“Good, you’re all here.” He looked round the room and noticed his people had grouped themselves, as usual, into the primarily technical and the primarily business oriented. He also noticed the empty seats on either side of Wren; there was a job to be done there in getting Wren accepted. “You all know we have a big job on our hands in bringing the PhoneMob up to date, and I want to make this happen fast and efficiently. We need the relaunched organization to make a mark as a modern outfit, without losing its existing customers.”

“I know you already have a lot on your plate, and it would be unfair to saddle any of you with the responsibility for coordinating this,” — he wished this was his real reason — “so I have asked Alex Wren here to join us and help pull things together.” He could see from their faces that some of the business team would happily have done without this help. Eleanor Hewish, his volatile but ambitious CIO was staring directly at him, while the fastidious compliance officer, Ivor Davies, was looking critically at Wren, who still seemed relaxed and confident. The technical side seemed less threatened. Claire Moss, the lead system analyst, was still studying her papers, while Trevor Clark, the

configuration manager and Trudy McNeal of procurement were giving him their polite but guarded attention. Nigel Oliver, the infrastructure manager was sketching some sort of diagram on his pad, although Marcus couldn't make out whether it related to this meeting or to something else on Nigel's mind.

He turned towards the other member of the meeting, who was from outside his core team. "I'm assuming you already know Edward Faversham, from the PhoneMob; you must all have had some dealings with him during the negotiations. He is here to give us his experience of their current services and processes, and his view of the requirements for the merged system." Faversham shifted nervously and looked round the table, acknowledging the nods of recognition. "So, let's get down to business. Edward, perhaps you can give us a quick run-through of the main business units to get us started?"

Faversham flicked the remote control to display a picture of a dishevelled businessman jumping up and down on his mobile phone. "As you know," he said, "our mission is to take the hassle out of owning a mobile. We get the phone fixed if it breaks, and we provide a replacement if we can't get it going again straight away. We have outlets all over Europe, so we can help while you are travelling and return the repaired phone to you at your home base." He clicked, and the view changed to a map spattered with dots, although, Marcus thought, not that many, yet.

"To be honest," Edward said, "our main problem at present is that these sites are not that well integrated, but so far we have grown from a base where each local office franchise did its own thing, and had its own PC-style support, and we can't grow any more without a proper infrastructure."

He clicked again, revealing a complicated block diagram on the screen, and pointed to a set of paths highlighted in red. "This," he said, "is the work scheduling and logistics. That's where there is the main scope for automation. This other area over here covers the liaison with the handset suppliers, and at present is mostly person-to-person negotiation; and this is the corporate relationship management, which is one of the areas we need to put on a much firmer basis. Down here are the charging and account management processes. No, it's the logistics that we should concentrate on here, at least initially." He paused and considered his chart, as if seeing it for the first time. "I guess the main difference from what you are used to is the complexity of the data management involved."

Eleanor interrupted. "What's the problem?" she asked sharply. "It looks simple enough." "Oh no," he said, "far from it. Think about the investment a user has in the content of his mobile phone. The SIM is bound to the number, and can be swapped, but the contact lists and the message and call history are needed to keep things moving, and all that data is potentially sensitive."

This brought Nigel in. "So your service engineers have access to sensitive customer data? Isn't that a security problem?"

“Well, at present we don’t provide the flexibility customers want, but data privacy certainly will be an issue in what you are planning now. Let’s look at the most common use case,” Edward said. “A corporation takes out a contract with us to support its salespeople. One of its representatives is in Stuttgart, and his phone keypad fails. He visits our Stuttgart office, and we put his phone into a test harness to bypass the keypad and download the state into a courtesy unit; we swap the SIM into this new unit, and the salesman can go on his way. Before he leaves we estimate the time to repair, and he tells us that by then he will be in Berlin. We commit to deliver the repaired phone to his Berlin hotel.”

“What about if there is some extra delay?” Eleanor asked. “Suppose there was a deeper problem requiring a return to the manufacturer that didn’t show up until you had done the keyboard replacement?” “Well, then we send an SMS giving a revised estimate. He can then reply to the SMS, or access our website, to confirm the new time.” Claire leaned forwards. “But he might have returned home by then.” “Yes, but he can change the delivery address if necessary. He can also do that if his own plans change.” Claire was not satisfied. “What happens if the phone is already in transit to Berlin when he makes the change?” “Well, our logistics support has to cope with that. We do a confirmation check before the phone leaves the local depot, and the couriers are instructed to ensure the customer has not yet checked out before they complete the delivery.”

“The loan phone is returned using a prepaid wallet; the user’s information was copied to the repaired phone before it is dispatched. After all this, the corporate account is billed for the work.” Nigel frowned. “What about the state in the loan phone?” “Well,” Edward spread his hands. “If the user needs complete synchronization, he can come in to the local office to make the exchange; that also means we can do the SIM swap for him. But we find people are generally happy to make separate records if they know they are using a loan phone. We can e-mail the state back to them after return of the phone if they want.”

Marcus looked around the group. “Well, any comments on what we have heard so far?” Nigel looked up. “It seems to me that there is a huge piece of design work to do here,” he said. “We need to look at a number of different infrastructure use cases, and it will take my team some time to understand the application structure, plus the security aspects, and the user interface issues, just for starters.” “Hang on,” interrupted Eleanor, “we need a design the business side can understand; we have got to keep it simple for them, or we can’t have a meaningful discussion about the processes they need.” Ivor frowned. “But we need to check over precisely that detail to pick up inconsistencies that would give us compliance problems later.” Trudy nodded. “And we need to think from the start about whether our corporate standards and procedures will be suitable. We need to have all the detail there,” she

said. “But look here,” said Claire, “we can’t add all that into the application design, or we will lose sight of the application architecture. It will be a mess. And we don’t need the application and infrastructure teams to duplicate each other’s work.”

Marcus banged the table. “Hold on now. You are all talking at once. Alex, you haven’t spoken yet. What’s your take on what you have just heard?”

Alex stood up and walked over to the flipchart. “I think you have all made good points.” He said. “This is a very large design, and there are conflicting requirements, needing different focuses and levels of detail. The best way to handle it is to divide and conquer; you should identify the key stakeholders, and then describe the problem from their points of view. If you can identify the right stakeholders, the different views can be largely independent, and so they can be worked on in parallel. The framework I want you to work with is called ODP, and it stresses five such viewpoints.” He sketched a diagram on the chart with five arrows converging on the design problem. “These correspond roughly to organizational responsibilities in a situation such as yours, namely business processes, systems analysis, data integrity, infrastructure and intellectual or physical resources. If you each start with your own focus and the level of detail you need, we can then link the different views together to describe the complete problem.”

“What I would like each of you to do is prepare an outline of the requirements as you see them, and then we can see how they fit together and move on from there.”

“OK then,” Marcus said, “let’s do it, and meet again on the 28th to see how well it works. That’s all for now; I’ve got to talk to the finance people again.” He stood up and stalked out of the room, leaving the discussion going on; it was still going on hours later.

---

## 1.1 The ODP Reference Model

The aim of the Reference Model for Open Distributed Processing (the RM-ODP) is to provide a framework for specifying and building large or complex systems; we call the systems being produced *ODP systems*. These systems may be classical IT systems, information systems, embedded systems, business systems, or anything else in which we are interested.

If a system is at least moderately complicated, it is useful to extract the description of its structure and external properties from the details of its components or subsystems. If this abstract view concentrates on the distillation of general principals, it is called an *architecture*. When presented in a way that is useful for the derivation of a whole family of future systems, it is called a framework. Hence, when describing a business system supporting a broad range of applications, it is common to talk of an enterprise architecture or an

enterprise framework. However, of late, these terms have been over-used and now lack focus. In these terms, the RM-ODP is an architectural framework for the design of any distributed system, particularly those whose complexity poses a challenge.

The Reference Model was published in the mid 1990s, following almost 10 years of work in the International Standards Organization to harvest the best architectural work up to that time. The results were published as common text by both ISO and the ITU-T (the telecommunications standards forum). The RM-ODP was published in four parts [2-5]. These four parts provide an introduction, a set of rigorous basic concepts, the architectural framework, and a link to supporting formal techniques. The users of this framework are expected to be system designers, but it is also intended to help people who build tools to support such design activity, or who produce standards to capture best practice and reusable mechanisms in this area.

The RM-ODP defines a framework, but not a methodology. It gives the designer a way of thinking about the system, and structuring its specification, but does not constrain the order in which the design steps should be carried out. There are many popular design processes, and the framework can be used with practically any of them.

Since ODP system designs are typically large collaborative efforts, it is likely that the actual process will be iterative, filling in detail in different parts of the specification as ideas evolve and requirements are better understood. However, the sequence in which this is done will depend on circumstances. In a green field, design may follow a classical top-down, waterfall-style pattern. In a legacy migration exercise, it will start by capturing existing constraints. In an agile or rapid prototyping environment, design will stress modularization and fine-grained iteration. The ideas for structuring specifications presented here can be applied within any of these methodologies. They remain valid if the design approach changes, and provide a common framework and vocabulary for collaboration between designers using different processes.

Many competing architectural frameworks have recently been proposed, and some of the better-known ones are reviewed in section 16.5. However, ODP offers a set of distinguishing features that make it particularly relevant for the specification of open distributed systems for enterprise and information handling applications. First, it has the authority and stability that goes with its status as an international standard. You can use it with the confidence that it is controlled by proven international processes, and will not be unilaterally changed by some individual group or private organization. Second, it is based on a rigorously defined set of formal concepts, and so has a precision that positions it in close alignment with the current software engineering and model-driven trends within the industry. Third, it is based on well-developed enterprise modelling languages and a distributed system architecture, which jointly position ODP as a perfect framework for modelling large, cross-organizational and cross-jurisdictional systems that communicate over the Internet. Finally, it has a well-integrated and fully developed treatment of

conformance and compliance that makes it a practical tool, feeding naturally from design to development. All these factors contribute to ODP's position as the most effective architectural vehicle for understanding and achieving system interoperability.

---

## 1.2 Viewpoints

### 1.2.1 The Idea of Viewpoints

The RM-ODP is perhaps best known for its use of viewpoints. The idea behind them is to break down a complex specification into a set of coupled but separate pieces. This is a very old idea, used, for instance, to simplify engineering drawings (as shown in figure 1.1) and in the building plans produced by architects.

The writers of the reference model were keenly aware of the need to serve different stakeholders, and introduced the idea of there being a set of linked

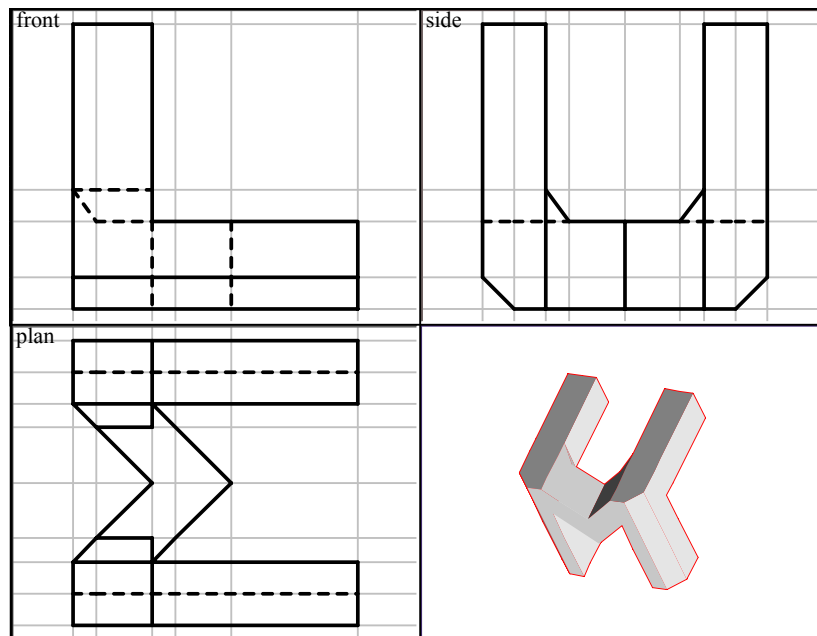


FIGURE 1.1: A traditional use of viewpoints in a mechanical drawing with first-angle orthographic projection.

viewpoints to maintain flexibility and avoid the difficulties associated with constructing and maintaining a single large system description.<sup>1</sup>

The idea is that, because the viewpoints in the set are interlinked (as explained in section 1.2.4) they are equivalent to a notional single large model. This equivalent model is not presented to any one user, since it would be too complex to be useful. However, tools may construct part or all of such a model where they need to manipulate information from more than one viewpoint.

Indeed, the idea of having a well-integrated set of tools, or toolchain, is fundamental to the success of the viewpoint approach. Without such a toolchain, it would be necessary for programmers to consult every viewpoint to discover all the constraints on the code they are to write, largely nullifying the advantages of the approach.

The viewpoints are coupled to form a complete system specification, but it is not necessary to use the same techniques when expressing each member of the set. Each stakeholder will be familiar with languages and notations well tuned to handling their particular interests, and so the techniques used will often be different. Of course, the more similar the techniques, the simpler it is to correlate the various views, so there is a trade-off between ease of solution to the specific constituent problems and ease of integration.

One of the aims in selecting a set of viewpoints is for them to be as loosely coupled as possible. A benefit of using viewpoints is that they allow parallel activity in different teams, and so allow some parts of the specification to reach a level of stability and maturity before others. It takes skill to pick a good set of viewpoints; if two viewpoints are linked in too many ways, independent activity will be difficult, and if they do not reflect common groupings of activity in the industry, they will not belong to clearly identifiable stakeholders.

Thus we can see that, for example, separate descriptions of system state and behaviour would be poor candidates for viewpoints; they are linked by a fundamental duality and so closely intertwined. On the other hand, service use and service provision are good candidates because, for a well-chosen service, they each deal with independent detail; as long as the right service is provided, the details of how it is provided and how it is used do not interact and can be handled by different teams. The two sets of concerns are nearly orthogonal.

### 1.2.2 A Specific Set of Viewpoints

The idea of separating concerns by using a set of viewpoints can be applied to many design activities. However, components are more likely to be reused if the same set of viewpoints is accepted by many different teams. The largest possible degree of commonality is needed to support the creation of an

---

<sup>1</sup>The idea of providing a set of viewpoints has arisen in a number of different design and software engineering areas, and an attempt has recently been made to capture the general idea in IEEE 1471, *Recommended Practice for Architecture Description of Software-Intensive Systems* [75], which has recently been refined within ISO as the standard ISO 42010 [24].

international standard, where a single approach is needed to cover a large and long-lived community of users. The ODP reference model therefore defines five specific viewpoints (see figure 1.2), intended to appeal to five clear groups of users of a whole family of standards.

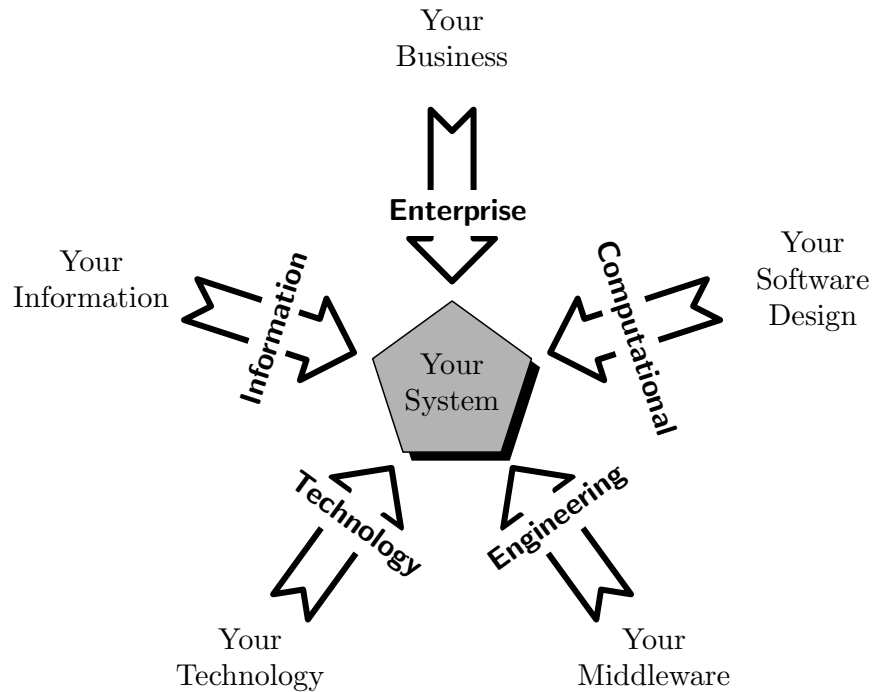


FIGURE 1.2: The five ODP viewpoints.

These five viewpoints are each the subject of a subsequent chapter, but they are introduced briefly here, concentrating on the broad objectives and areas of concern they cover.

The *enterprise viewpoint* focuses on the organizational situation in which the design activity is to take place. It concentrates on the objectives, business rules and policies that need to be supported by the system being designed. The stakeholders to be satisfied are therefore the owners of the business processes being supported and the managers responsible for the setting of operational policies. The emphasis is on business and social units and their interdependencies.

Note that the use of the word *enterprise* here is intended to cover any activity of interest; an enterprise can be whatever the specifiers have been tasked to describe. It can be a single product and its users, or a commercial organization, or a larger social structure involving many corporate or governmental entities. It captures whatever field of application we are currently focusing upon in this particular design activity.

The *information viewpoint* concentrates on the modelling of the shared information manipulated within the enterprise of interest. The creation of an information specification has broadly the same objectives that creation of a data dictionary had for previous generations. By providing a common model that can be referenced from throughout a complete piece of design, we can ensure that the same interpretation of information is applied at all points. As a result, we can avoid the divergence of use and incomplete collection of information that would result from separate members of the design team each making their own decisions about interpretation.

As an ideal, we would like there to be a single universal information model, but this is clearly not practical. The aim here, therefore, is to achieve a shared model for the particular design activity, but we shall see later that even this may not be achievable when we are considering federation of systems or legacy integration. The best we can achieve is a single model used within the scope of a particular design authority. However, this is already sufficiently challenging, while giving potential for huge quality improvements.

The *computational viewpoint* is concerned with the development of the high-level design of the processes and applications supporting the enterprise activities. It uses the familiar tools for object-oriented software design, expressing its models in terms of objects with strong encapsulation boundaries, interacting at typed interfaces by performing a sequence of operations (or passing continuous streams of information). The computational specification makes reference to the information viewpoint for the definitions of data objects and their behavioural constraints.

The computational design is abstract in that its objects are not placed at specific locations and allocated particular resources to run on; this is done elsewhere. The same design can be implemented in many different ways by placing its objects on different platforms.

The *engineering viewpoint* tackles the problem of diversity in infrastructure provision; it gives the prescriptions for supporting the necessary abstract computational interactions in a range of different situations. It thereby offers a way to avoid lock-in to specific platforms or infrastructure mechanisms. A particular interaction may involve communication between subsystems, or between objects co-located in a single application server, and different engineering solutions will be used depending on which is currently the case. The engineering specification is akin to the specification of how middleware is provided; there are different solutions for use in different operating environments, but the aim is to provide a consistent set of communication services and other supporting services that the application designer can rely on in all cases.

The engineering viewpoint in ODP is also concerned with the provision to the computational designer of a set of guarantees, called *transparencies*. Providing a transparency involves taking responsibility for a distribution problem, so that the computational design does not need to worry about it.

Many of the mechanisms needed are nowadays available in the form of standard middleware or web services components, simplifying the engineering

specification, since it can reference the existing solutions and merely state how they are combined to meet the infrastructure needs of the system.

The *technology viewpoint* is concerned with managing real-world constraints, such as restrictions on the hardware available to implement the system within budget, or the existing application platforms on which the applications must run. The designer never really has the luxury of starting with a green field, and this viewpoint brings together information about the existing environment, current procurement policies and configuration issues. It is concerned with selection of ubiquitous standards to be used in the system, and the allocation and configuration of real resources. It represents the hardware and software components of the implemented system, and the communication technology that provides links between these components. Bringing all these factors together, it expresses how the specifications for an ODP system are to be implemented.

This viewpoint also has an important role in the management of testing conformance to the overall specification because it specifies the information required from implementers to support this testing.

### 1.2.3 Viewpoint Languages

We can think of any mechanism for conveying ideas as being a language, be it written, drawn or spoken. The communication can be between people, between machines, or understood by both.

Thus, we can speak of the set of concepts, conventions and constraints expressed in a particular viewpoint as forming a *viewpoint language*. The rules of interpretation for such a language can, in a particular instance, be seen as representing a viewpoint virtual machine. We can think of the supporting tools as parsing the language's grammar and checking its semantic rules, or as implementing the equivalent virtual machine; these are just two sides of the same coin.

From an architectural point of view, we need not be concerned with the physical representation of this language as marks on paper or encoded in messages. An abstract language can be represented by a number of different concrete notations, suited to use in different situations. Many tools, for example, can work with either a graphical or a textual notation, and store designs in a third, machine-oriented format, such as a dialect of XML. These are all different notations expressing the same abstract language.

Thus for each of the five viewpoints being considered here, we have a corresponding viewpoint language. We talk about the viewpoint when we wish to stress the perception of the stakeholder concerned, and about the language when we want to emphasize the way the ideas are communicated, but the two aspects are intimately coupled; one cannot express the ideas without using the language.

Because the different viewpoints stress different aspects of the design, and do so using different techniques, the stakeholders will each be most comfort-

able working with their own style of language and notation. For example, someone writing a business policy may be happier expressing goals in a declarative way, saying, perhaps, that a target level of production should always be achieved, while someone documenting a process may naturally think in imperative terms, expressing what has to be done as a sequence of instructions.

As another example, a business process may involve extended activities with real-time deadlines and have measures of the fraction of work completed, while a computational task may concentrate on sequences of events that are considered indivisible (or *atomic*), with deadlines expressed by equivalent events, with no model of continuous time at all. Working with sequence, but without continuous time like this simplifies analysis, but makes the expression of continuous properties of the system, such as quality of service, more difficult.

#### 1.2.4 Viewpoint Correspondences

Although we achieve a powerful simplification by dividing a system specification into the views seen by different stakeholders, the specification must continue to be a coherent description of a single target system. If we had no links between the viewpoints, this would not happen; what was intended to be a single design would just fall apart into five bits. It is therefore vital that the viewpoints be linked, and this is done by establishing a set of correspondences between them, as visualized in figure 1.3.

In current software tools that present different user views, this linkage is often derived from the names of objects. If the same name appears in two diagrams, they are assumed to represent two aspects of the same thing.

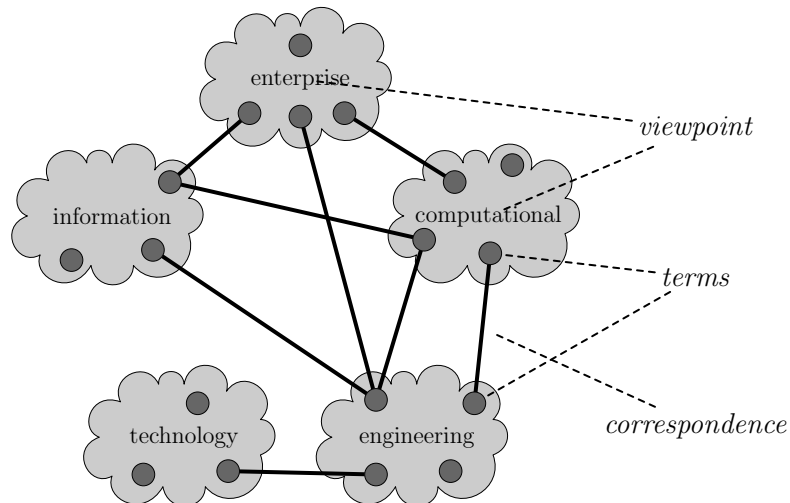


FIGURE 1.3: How correspondences link terms in different viewpoints.

However, if the viewpoints are to be developed by loosely coordinated teams, it is not safe to assume they share a single namespace; it is just too expensive to ensure that name assignments are unique. It is also often the case that the correspondences are not simply one-to-one; the relationships will generally be more complex.

This can be seen by considering the correspondences between the computational and engineering viewpoints (a situation similar to that between the user and provider of middleware). The computational viewpoint takes a very simple view of interactions, abstracting away from the platform-specific details of particular interactions; the engineering viewpoint, on the other hand, abstracts away from most of the detail of the computational design, distinguishing just a few categories of interacting objects, not worrying about why the interactions take place. Thus, the correspondences between computational object types and basic engineering object types representing them are typically not one-to-one. The correspondences often involve some level of abstraction or filtering in one direction or the other.

---

### 1.3 Fundamental Concepts

Part 2 of the ODP Reference Model sets out a family of basic concepts, explaining precisely what is meant when we say that ODP is based on an object model. This is particularly important when dealing with the coordination of a number of tools that all share the general idea of an object and an interface at a ball and stick level, but often diverge in subtle ways when the fine detail is examined.

Precision is essential when we are trying to bring together multiple viewpoint languages, together with the languages implicit in the available tools, to create a consistent framework. This integration depends on knowing exactly what the basic concepts used are, and where they vary between languages.

The fundamental concepts are defined in a general way, and they can then be used in any of the ODP viewpoints. Often, a concept is further specialized in a particular viewpoint, but always in a way that is entirely consistent with the basic definition. For example, the concept of an interface is defined in full generality as one of the fundamental concepts, but viewpoints specialize it, so that we have engineering interfaces, computational interfaces and so on.

This book does not go into all the detail; that can be found in the standards. However, there are some fundamental concepts that are used in all the viewpoint languages; these deserve to be discussed here. They are presented in outline in this section, and more detail can often be found when they are used later in the book.

### 1.3.1 Object Model

ODP system specifications are expressed in terms of *objects*. Objects are representations of the entities we want to model, including physical elements (mobile phones), human beings (John, the repair centre clerk) or more abstract entities (a pending repair order). An object contains information and offers services. In other words, an object is characterized by its *state* or, dually, by its *behaviour*. Depending on the viewpoint and the style of the notation used, the primary emphasis may be placed either on behaviour or on state.

The use of the object paradigm provides *abstraction* and *encapsulation*, two important properties for the specification and design of complex systems. Abstraction allows highlighting of those aspects of the system relevant from a given perspective, while hiding those of no relevance. Encapsulation is the property by which the information contained in an object is accessible only through interactions at the interfaces supported by the object. Because objects are encapsulated, there are no hidden side effects outside the object arising from the interactions. It also implies that the internal details of an object are hidden from other objects, which is crucial in ensuring the interchangeability of alternative implementations, and in providing the basis for dealing with heterogeneity, interoperability and portability.

Behaviour is expressed as a collection of *actions*. Actions can be anything that may happen, and can be classified into *interactions* (which involve participation from the object's environment) and *internal actions* (which take place without such participation). An example of an internal action is the one that models the sudden breakdown of a mobile phone. Interactions are used to model, for instance, a customer's request to the PhoneMob clerk to have his handset repaired, or the act of transferring the data from one phone memory card to another. Each of the objects involved in an interaction plays a particular *action role* characterized by the information it contributes or accepts and by whether or not it originated the action.

A *system* is composed of a configuration of interacting objects. Objects interact at *interfaces*, which are subsets of their possible interactions with other objects, together with the set of constraints on when they may occur. An *event* is the fact that an action has taken place. When an event occurs, the information about the action that has happened becomes part of the state of the system and may thus subsequently be communicated in other interactions. Such a communication is called an event notification.

The specification of an interaction concentrates on the objects participating in it. However, in some circumstances, we may want to focus on exactly where the interaction takes place and how it might be observed. This is done by introducing the concept of an *interaction point*, which is concerned with where, in time and space, the interaction happens. We shall see later, in chapter 8, how this concept leads to the more specific ideas of *reference point* and *conformance point* when we are concerned with ensuring that some interactions are observable during testing.

Finally, a *service* is a behaviour, triggered by an interaction between provider and consumer objects, that adds value for its users by creating, modifying or consuming information.

### 1.3.2 Types, Classes and Templates

As in most object-oriented modelling and programming languages, ODP objects, actions and interfaces are generally specified in terms of their *types*. In ODP, a type is a predicate that characterizes a set of elements, and serves to identify and describe them. For instance, type `Customer` describes the common characteristics of the company's customers that are relevant to the system. Similarly, type `RepairOrder` captures the information that any repair order in the system should have. Predicates can be expressed in many different notations, from textual languages to graphical modelling notations (a UML class is a typical notation for representing types, describing both the state and the behaviour of the corresponding objects in terms of attributes and operations).

Types help to classify entities into categories, which form collections of objects that satisfy a given type. Such collections are called *classes*. Note, however, that the UML concept of class differs from the ODP concept of class. A UML class is a *description* of a set of objects, while an ODP class is the set of objects itself. Therefore, the UML concept of class is closer to the ODP concept of type. There is no UML concept that is similar to the ODP concept of class.

In addition to types and classes, ODP also uses the concept of *template*, which refers to the specification of an element, including sufficient detail to allow an instantiation to take place. For example, object-oriented programming language concrete classes are templates. Types can only be involved in testing whether instances satisfy them or not; templates can be used to create instances. Templates may also include parameters, which will need to be bound to specific values at instantiation time.

### 1.3.3 Composition

*Composition* and its inverse, *decomposition*, are key concepts in ODP. We have already mentioned that systems are composed of interacting objects. The composition of two or more objects is itself another object (called the composite object). The characteristics of the new object are determined by the objects being combined and by the way they are combined (the composition operator used).

Behaviours can also be composed, yielding a new behaviour. This can apply either to behaviour fragments or to the behaviour of complete objects. Thus, the behaviour of a composite object is the corresponding composition of the behaviour of the component objects, possibly hiding some interactions to make them internal actions.

Action compositions can be specified to form processes. A *process* is a collection of steps taking place in a prescribed manner and leading to an *objective*. A *step* is an abstraction of an action, used in a process, that leaves unspecified the objects that participate in that action. Steps are introduced in the definition because not all the action details need be specified in the composition. The objective is present because the goal of a process should always be made explicit in its specification. In ODP, the objective of an element expresses its practical advantage or intended effect. It is expressed as preferences about future states.

### 1.3.4 Grouping Objects Together

ODP distinguishes different ways of organizing sets of objects. In the simplest case, objects can be organized into *groups*, which are sets of objects with a particular *relationship* that characterizes either some structural interdependence between them, or an expected common behaviour. Examples are the group of all the information objects, or the group of engineering objects that are related to a given computational object.

*Domains* are often used in describing these groupings; a domain is a set of objects related by a characterizing relationship to a controlling object. Every domain has its associated controlling object, which is not generally itself a member of the domain. One example of this is a naming domain, in which a set of names are associated with objects by the controlling object. Another is a management domain in which a set of printer objects is managed by a controller.

Objects can also be organized into *configurations*, which are collections of objects linked by their interactions at particular interfaces. Examples are the configuration of objects that together provide a given service, or the configuration of engineering objects that implements a channel.

These concepts can be specialized in specific viewpoints. For example, in the enterprise language, a *community* is a configuration of enterprise objects formed to meet a particular *objective*, as specified in a given *contract*. In our PhoneMob example, one community is the configuration of objects that together provide the basic repair services to customers. Another community is the logistics organization formed by a set of objects with the objective of providing delivery services to users in a secure and timely manner. A third example is the banking community, which is a configuration of enterprise objects that together provide a set of banking services (payments, money transfer and so on) to its customers.

Finally, a *federation* is a community of domains formed to meet a shared objective. It models many commercial situations, such as the setting up of partnerships and joint ventures; examples are the federation of a set of airlines that agree to work together to provide transportation services to their customers by means of code-shared flights, or a federation of banks that share their ATMs so that customers can use any of them interchangeably. In our

example, the PhoneMob company and a large insurance firm can federate to provide a wider range of services to their individual customers. Each member of a federation agrees by participating in the federation to be bound by the contract and policies of the community (which may include obligations to contribute resources or to constrain behaviour) so as to pursue the shared objective. At the same time, a federation preserves the autonomy and independence of the original participants.

### 1.3.5 Contracts

As a general concept, a *contract* defines the rules governing the collective behaviour of a set of objects. It specifies obligations, permissions and prohibitions that apply to these objects when they act as a group. These could express, for example, quality of service constraints, indications of duration or periods of validity, behaviour that invalidates the contract, or liveness and safety conditions.<sup>2</sup>

The contract concept can be used in any viewpoint. In the enterprise viewpoint, we have community contracts (see chapter 2) reflecting a business context for interactions. For example, the community contract representing a repair organization expresses obligations on its service centres, their staff and customers, as well as conditions about efficiency, security, response times and confidentiality to be met when delivering the repair services.

There are several uses of the concept of a contract in the computational viewpoint. One example is its use to describe a service contract, which defines the obligations that an object makes when providing an interface with which arbitrary other objects will interact. Another is the binding contract which captures the properties agreed upon when a particular binding is established (this may be either a primitive or a compound binding; see chapter 4).

Finally, any computational object interacts within an environment representing its place in a configuration, and an *environment contract* states non-functional properties of the interactions in which an object participates, such as response time, throughput or resource consumption. These computational environment contracts reflect constraints on the corresponding objects and interactions in the engineering viewpoint.

### 1.3.6 Policy Concepts

*Policies* provide a powerful mechanism for declaring business rules, and also for specifying and implementing the structural and behavioural variability required in any open distributed system. Policies serve to identify the pieces of behaviour that can be changed during the lifetime of the system, as well as

---

<sup>2</sup>Liveness is the property of a system that says it will eventually do what it is supposed to do, and safety is the property that says it will never do something it is supposed not to do.

the constraints on these behaviours. In other words, a policy can be seen as a constraint on a system specification foreseen at design time, whose details can be modified to manage the system to meet particular (and changing) circumstances. More details are given in chapter 10.

Policies are defined in terms of *rules*. A rule is a constraint on a system specification. Rules are normally expressed as obligations, permissions, authorizations or prohibitions. For instance, one rule may say that any phone user associated with a customer can place an order to have a mobile phone repaired (*permission*), while another rule may dictate that the repair centre must repair the handset or provide a substitute phone within 48 hours (*obligation*); a further rule may state that VIP customers are entitled to get substitute phones immediately (*authorization*); a fourth rule may say that, for security reasons, a SIM memory must not be returned to any customer other than the one who placed the original repair order (*prohibition*).

---

## 1.4 Useful Building Blocks

One of the aims of the reference model is to promote the use of common terminology for describing distributed architectures, particularly with regard to the functional elements needed to support and manage distributed applications. Many vendors have their own names for these functions, making it more difficult to draw parallels between different solutions, and to find vendor-neutral terms to describe interworking mechanisms.

The reference model therefore provides a catalogue of architectural functions needed to support distribution. This vocabulary can be used in any of the viewpoints, but it concentrates on giving full coverage of the engineering viewpoint and many of the functions identified are normally contained within the system's infrastructure. However, some may be available as services for use in the computational specification, and others may be exploited in a more abstract form, for example when modelling repositories in the enterprise specification.

The catalogue covers four main areas. The first is the *management* of different groupings of engineering objects, providing control of resourcing, protection and activation. Then it categorizes *synchronization* mechanisms, followed by various *repository* functions, together with different specialized features to aid resource discovery and interworking. Finally, it identifies important building blocks for the provision of a range of distributed *security* functions.

Use of this taxonomy helps different teams to understand the functions offered by existing components and so aids reuse. The functions can be used as keywords to index catalogues of components or collections of higher-level technical design patterns, aiding design. This provides a common vocabulary

for explaining a system's architecture to potential users or external assessors or conformance testers. It also allows the expression of platform-independent designs in a commonly understood way without reference to specific technological choices, making it easier for teams using a model-driven engineering approach to exchange ideas on requirements and solutions.

---

## 1.5 Service Orientation

In recent years, much play has been made of the use of service orientation as a design principle. From an architectural point of view, however, there is no significant difference between service-oriented architectures (SOA) and the architectural framework defined in ODP; current service-oriented schemes can be seen as a subset of the more general ODP approach.

The main tenets of SOA are that functions should be packaged into loosely coupled units that provide clearly defined services, and that applications should be constructed by composition of services that can be discovered dynamically based on some form of publication or brokerage mechanism. More recent SOA activities add to this a distinction between services offered from different design perspectives, yielding different flavours of service, such as business services, technical services and so on.

ODP defines service as a fundamental concept, representing the added value offered as a result of interaction at some interface. Since the ODP object model is based on strong encapsulation, this means that there is a close alignment with the SOA view of service. The discovery and dynamic use aspects are covered by the ODP binding model and the definition of common functions such as the trader. The different types of services are captured by the ODP viewpoints, with business services being expressed in the enterprise viewpoint and the technical services being expressed primarily in the computational viewpoint. However, in the ODP enterprise viewpoint, as we shall see in the next chapter, there is a greater emphasis on declarative expression and flexible structures, so the definition of a business service is only one of the available design tools. Correspondences defined between the different aspects of a service in different viewpoints are also needed to provide a consistent specification of the service as a whole.

So what differentiates a service-oriented architecture? The main differences are not architectural, but are more concerned with raising the engineering expectations about openness and resilience, largely as a result of years of implementation experience with web-based systems and of striving for the widespread adoption of open resource identifiers like Uniform Resource Identifiers (URIs).

---

## 1.6 Human Computer Interaction

An essential aspect of the design of any information system is the specification of the interfaces the system offers to the people that will interact with it, something that is normally called human-computer interaction. In this design field, the user interface (or UI for short) is where interactions between humans and machines occur.

A typical enterprise design will be concerned with many kinds of activity. Some of these will involve people, some will involve automated solutions, and some will involve human-computer interaction. However, these categories are not fixed; there will be a progressive migration towards automated solutions, without there necessarily being major structural changes in the design.

While, in a sufficiently abstract description, the functions performed can remain the same when such changes are made, the disciplines involved in expressing their detail are quite different; middleware providers and user interface designers use different tools and techniques. However, both disciplines make heavy use of tools. In the middleware world, tools automate generation of stubs from interface definitions and handle the incorporation of mechanisms to ensure reliability and fault tolerance. Tools applied to the user interface design, on the other hand, can produce software components to act as user proxies within the system and manage user dialogues in terms of interaction with, for example, a sequence of web forms and other pages. These tools can also be used to apply a uniform look and feel across the organization.

In the descriptions that follow, we will focus primarily on system structure, but we will indicate how the approach also enables the management of human-computer interaction, using either the same information that captures the system structure or specific information added to support this aspect of the design.

Thus, the enterprise description can be used to identify interactions between human and system-supported activities, while the information viewpoint can define data types that can be used either in system-to-system interactions or in user interfaces and the computational viewpoint can contain details of the dialogue needed to support human-computer interaction.

Introducing the support of user interfaces into the design will, in general, result in the inclusion of distinct objects into the computational design to represent each user and the proxies for these users within the supporting system. Doing so adds precision to the specification of the user interactions expected and, as we shall see in chapter 8, provides a basis for testing the correctness of conformance to the user interactions specified.

The support for user interaction is thus incorporated within the viewpoint framework, but we shall return to this topic to bring out the consequences for the individual viewpoint languages as they are discussed in following chapters.

## 1.7 The Right Tools for the Job

### 1.7.1 Reusing UML via UML4ODP

Although, as we indicated earlier, the ODP viewpoint languages are defined in an abstract way without commitment to a particular concrete notation, we do need to select such a notation before we can write a real, useful model. It does not matter particularly what notation we choose, as long as our toolchain can handle it and integrate it with others already in use, but it will help the designers to get started if the notation is already familiar.

UML [30] is currently the most popular modelling language in the industry, and most designers have some familiarity with it. Its expressive power can be increased by using the associated object constraint language (OCL) [36]. So why not use it here? The problem with UML as it stands is that it does not support the separation of concerns we want, or the structuring concepts ODP introduces to support it. UML has functional views, expressed in its diagram types, but they are closely coupled, and there is a single hierarchical namespace underpinning them. This makes the separation we want difficult to achieve.

What about model-driven engineering, then? Many MDE prototypes use UML for expressing their source and target models, don't they? Yes, they do, but that's the point. They work by translating between quite separate models, each with its own internal consistency and its own namespaces. The transformations provide the level of decoupling that is needed (see chapter 15).

With the development of these technologies, and of an increasing number of domain-specific dialects of UML, the originators of ODP returned to UML and decided that, because it was now widely accepted, it could be used in many cases to provide a familiar notation for the individual ODP viewpoint specifications. In consequence, they produced a new standard, *ISO 19793: Use of UML for ODP system specifications*, or UML4ODP for short [22]. This standard provides a profile that maps the ODP concepts to the UML notation,<sup>3</sup> so that they can be manipulated with conventional UML tools. A suitable plug-in to a UML tool allows consistency checking across multiple viewpoints.

Of course, ODP can be supported by other notations, tailored to other user communities, such as business process designers, as long as a similar level of tool integration can be achieved. However, in this book we will keep things simple and concentrate largely on the UML4ODP notation.

---

<sup>3</sup>The UML4ODP standard is based on UML 2.1.1 [30].

### 1.7.2 UML4ODP in a Nutshell

The UML4ODP standard defines both a UML-based notation for the expression of the ODP system specifications and an approach for structuring them using that notation, thus providing the basis for model development methods. This makes the UML4ODP notation useful not only to ODP modellers who want to use UML to describe their ODP systems, but also to UML modellers who have to deal with the specification of nontrivial systems and need some approach to structure their large UML system specifications.

UML4ODP uses standard UML concepts and relies on the standard extension mechanisms provided by UML for defining new languages and, in particular, on UML profiles. More precisely, UML4ODP defines seven related UML profiles: one for each ODP viewpoint, one for describing correspondences and one for modelling conformance in ODP system specifications (see chapter 8). All the model diagrams shown in this book are drawn using these profiles.

Thus, an ODP system specification expressed in UML4ODP consists of a single UML model stereotyped «ODP\_SystemSpec» that contains a set of models, one for each viewpoint specification, each stereotyped «<X>\_Spec», where <X> is the viewpoint concerned (see figure 1.4). Each viewpoint specification uses the appropriate UML profile for that language.

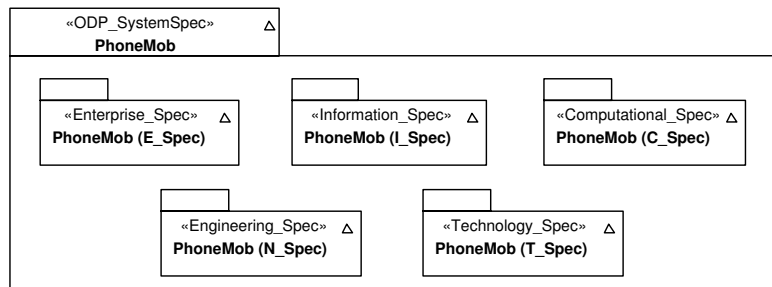


FIGURE 1.4: The viewpoints contributing to an ODP system specification, expressed using UML4ODP.

In the profiles, stereotypes are used to represent the ODP concepts as specializations of the appropriate UML metaclasses. For example, figure 1.5 shows the UML profile for the information viewpoint language, as specified in the UML4ODP standard. It defines eight stereotypes and the UML metaclasses they extend. Some of the stereotypes have associated icons (shown in the right upper corner of the stereotype box). Icons are very useful because they provide an intuitive notation to the users of the ODP specifications. This becomes particularly important in some viewpoints, such as the enterprise viewpoint. The designer must decide how much of this information to show in any particular diagram. In this book, for example, we generally show the icons, and include stereotype names where doing so helps understanding,

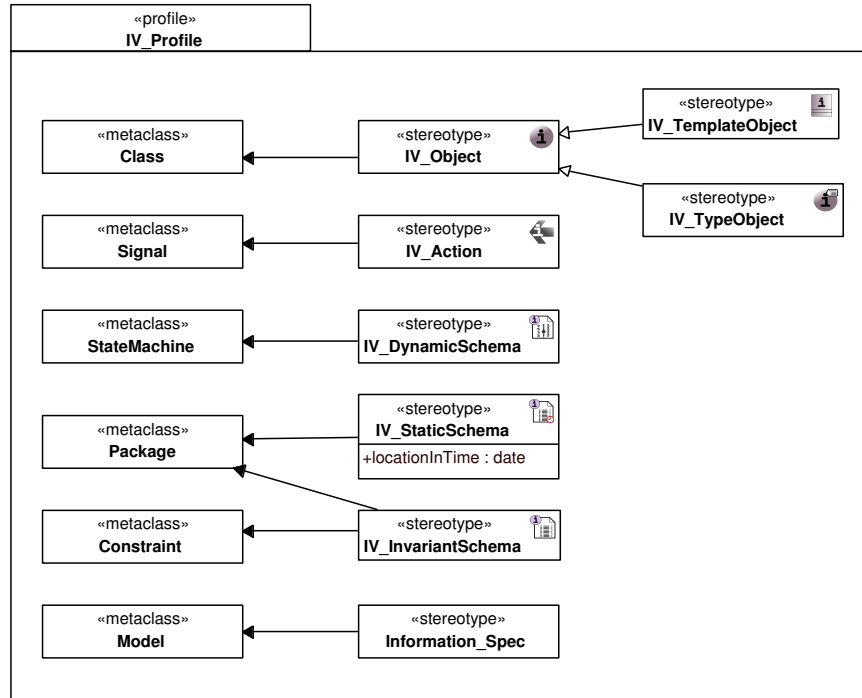


FIGURE 1.5: The UML profile for the information viewpoint.

Taken from the UML4ODP standard; for copyright, see Preface.

but omit them where the meaning is already clear, so as to avoid cluttering up some of the more complicated diagrams.

Tag definitions are also used to specify stereotype properties. For example, the tag definition `locationInTime` of stereotype `IV_StaticSchema` allows the specification of the exact location in time of the static schema being modelled. Figure 3.4 in chapter 3 shows an example of how this tag definition is used in the specification of a concrete static schema.

### 1.7.3 The Integrated Toolkit

Most software development now uses some form of integrated development environment (IDE), which takes care of much of the routine housekeeping concerned with building and archiving complex project structures. Eclipse [65], for example, is a particularly popular environment because of its extensibility. The same principle applies to other areas, such as business planning, and the trend is towards greater integration of the tools used, forming them into a single integrated toolchain.

One area where there has recently been a step forward in this respect is the development of *model-driven engineering* solutions, in which transformational techniques are applied to a quite abstract design, filling in detail to generate code suited to a particular environment. This process is generally not completely automatable, but reports based on test cases show that up to 85% automatic generation can be achieved in a typical database application [56]. OMG maintains an interesting library of such case studies [38].

This kind of integration is essential for the effective exploitation of a multi-viewpoint framework. Information needs to be taken from each viewpoint and combined to create a running system. As the design evolves, there need to be simple ways of checking that its parts have at least a basic level of consistency (just as compilation of a multi-package program gives some check on its structural coherence). All this requires the tools used in the different viewpoints to interwork so that checks can be made to see that the rules in the different viewpoints do not lead to contradictions.

Model-driven tools offer not only a much smoother pathway from design to implementation, but also a much more efficient basis for the management and evolution of large systems.

---

Marcus was sitting in on the design review to support Alex. Although the consultant was now known to the team and largely accepted, there had been teething troubles and a few bruised egos, so that support was still needed. Marcus had to admit he didn't follow a lot of the detail, but he could see that the main features of the new PhoneMob system were coming together.

It always intrigued him to see how the different groups imposed their own style on their models. He could recognize the organic style of the data modellers, with the branches and leaves growing and unfolding across the page, and could distinguish it from the Norman crypt style of the platform people, with whole solid blocks of function stacked up into stocky columns leading up to a broad vault of interconnections. The applications designers favoured tight modular balloons, kissing to exchange their messages. And the business analysts, with their narrow striped shirts, favoured road maps with the main routes and the special byways picked out with graphical symbols for all the world like service stations and tourist attractions.

But, putting aside these speculations, he knew it was not all going smoothly. There were rough edges and there were disputes. Even Marcus could see from the presentations that some things were being repeated by the different groups, but each in their own way. This needed to be sorted out.

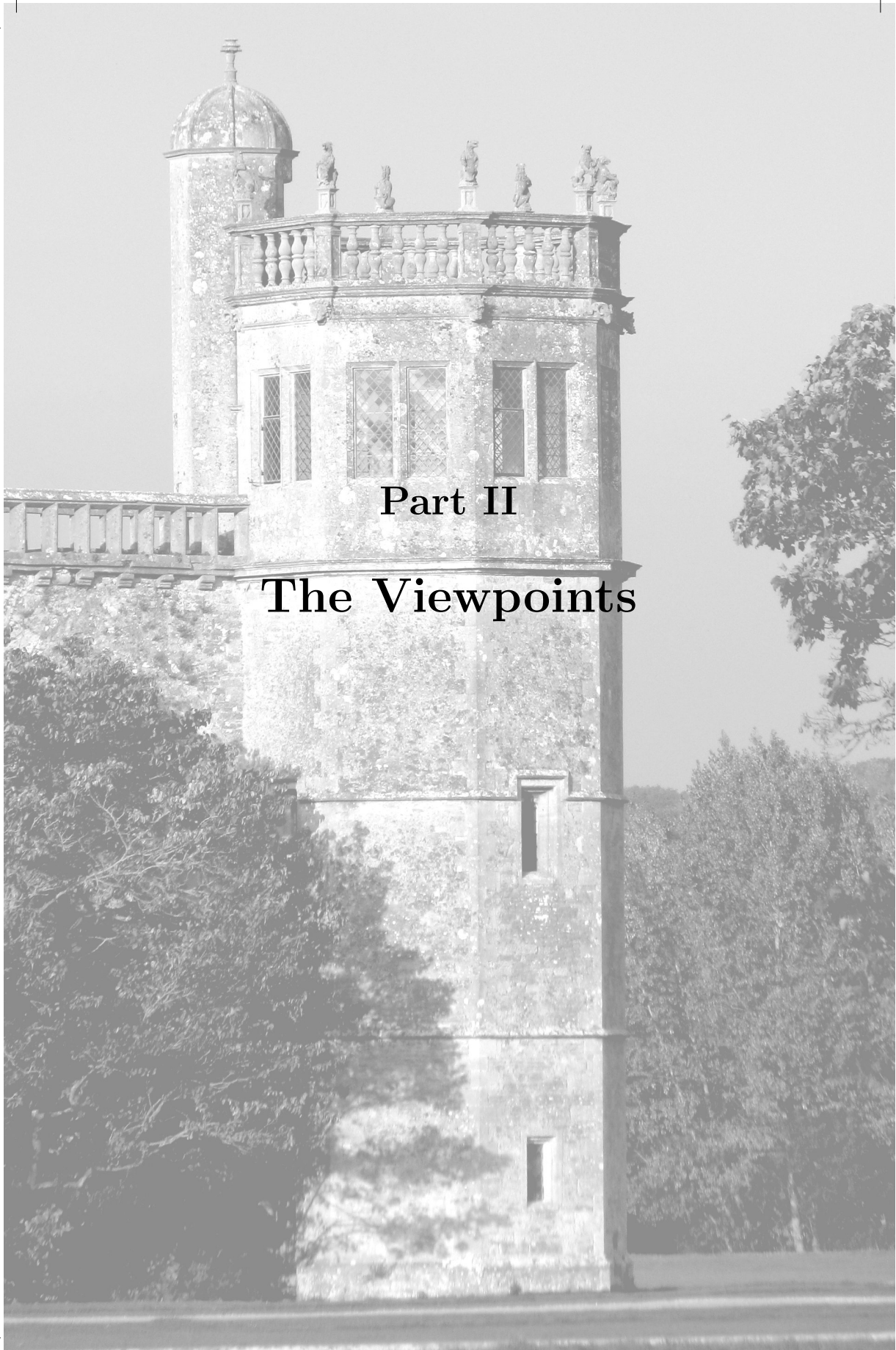
"Look," said Alex, "there is excellent work here, but you need to move towards seeing what you are doing as part of a bigger whole, and exploit it to simplify your lives." He turned to Claire. "Here, in the initial client registration, you have a big set of classes dealing with identity, account and contact details. And here, in the billing step, you have a different presentation of

roughly the same stuff.” She looked hurt. “Yes, but this is a first cut; I would expect to refactor those into a common set of classes when the structure is clear.” “But remember Ivor’s presentation. He has exactly the same information in his customer object already. You can just reference it.” “But he has too much information, and his names don’t correspond to the way we look at the process.” Alex smiled. “Remember what I said earlier about correspondences. You can refer to the classes in your design as being in an abstract package. Then, if you have a correspondence with the information view that renames and selects items as appropriate, the tools can construct the correct concrete classes in your view for you, based on that linkage. You don’t need to do it all from scratch again.”

Ivor scowled. “But then, if I change that part of my design, I might break the computational view.” “Certainly, it’s possible,” said Alex, “but when you ask for a global check before committing, you would get a warning. However, that should be rare, and if it happens, it probably means there is a shared issue you should be discussing anyway. Without that linkage, you would still be thinking about that common part of the design in different ways, but no one would know.”

Nigel waved his pencil in the air. “But that means that when the business analysts have a bright idea and decide to add a video feature to customer notifications, it ripples all the way down to the infrastructure, and the nightly builds will break! That would be ridiculous.” “Of course,” said Alex. “That would be silly. But what happens at the moment?” “Eleanor raises it in the weekly meeting, Claire looks at the application implications and sends me a memo, and I cost the deployment change with Trevor and Trudy. That’s generally the end of the matter.” Everybody laughed, except Marcus. “And we don’t improve our market position,” he growled.

“But hang on,” said Alex, “you still have the version management to protect you from any real damage. The business guys try the change, do a check, and get a red flag. They can then start the consultation process; maybe Eleanor starts a thread to discuss it, and everyone can contribute. You can all see the branch with the new change in it and trace the flags it has generated in your own view to see the consequences, so the resolution should be much quicker and the results more reliable. In the end, you all win. And we can do the same sort of thing with most of the other overlaps we have found this afternoon.”



Part II

The Viewpoints