The System Context Architectural Viewpoint

Eoin Woods
Barclays Global Investors
eoin.woods@barclaysglobal.com

Nick Rozanski
Barclays Global Investors
nicholas.rozanski@barclaysglobal.com

Abstract

A common requirement when describing the architecture of a software system is the ability to define the environment of a system, in terms of its external dependencies. In a view-based architectural description approach (such as "4+1" or "Rozanski and Woods") this need is met by adding a Context view containing this information to the architectural description and ideally defining a corresponding Context viewpoint to guide and standardise such views. This short paper explains the benefits of adding a Context view to architectural descriptions and provides an outline definition of the corresponding viewpoint to explain their content and how they are developed.

1. Introduction

The authors of this paper are also the authors of a set of viewpoints for software architecture (known in formally as the "Rozanski and Woods set" and defined in [1]). The original set of viewpoints defined in our book defines 6 viewpoints to guide the architectural design of information systems. However a "context" view that shows the overall context of the system being designed was not part of the set. This short paper explains why a context viewpoint was not defined in the original set and why, in the light of experience, the authors have found one to be necessary in practice. An outline definition of a context viewpoint for our viewpoint set is then provided, along with a brief discussion of how other viewpoint set authors have addressed this need.

2. Motivation for the Work

The process of software architecture involves both inward looking and outward looking concerns. The former are concerned with the details of the internals of the system and the latter are concerned with how the system interacts with its environment and serves the needs of its primary stakeholders (such as acquirers, end-users and support staff members).

Most software architecture definition and description techniques focus on internal concerns and in particular how the system's internal structures are designed and represented. This is perfectly natural, as the deliverables from software architecture activities are a set of descriptions of the system's internal structures, albeit with their design directly motivated and traceable to a set of stakeholder concerns.

When we defined our viewpoint set, we also focused on the internal structures of the system and our viewpoint set comprises 6 viewpoints, namely the Functional, Information, Concurrency, Development, Deployment and Operational viewpoints. All of these viewpoints guide the development of views that describe one or more aspects of the system's internal structure.

When defining the viewpoint set, we made the assumption that the system's requirements had been largely defined and that the work products of this activity would include clear statements of the system's requirements including the system's runtime context (i.e. the external actors that the system would be expected to interact with). While we recognise that there is extensive interplay between architecture, design and requirements definition, leading to an iterative approach resulting in interaction and change to the requirements, our assumption was that the system context would be defined by someone other than the architect. After all, if the system context isn't well defined, how can the requirements be written?

Experience has shown us to be wrong!

In reality, the software architect usually needs to include a definition of the system's context as part of their architectural description (and so create a view for it). The reasons for this include:

- the system context simply not being included in the requirements gathering exercise;
- a system context being loosely defined by requirements analysts, but at a level of detail which means that the architect needs to add significantly to it; and
- the software architect needing to reference elements of the system context elsewhere in the architectural description, so making it desirable for this information to be part of the architectural description and under the control of the architect.

This means that most of the architectural descriptions we have created since defining our viewpoints have included a "context view", created without a viewpoint definition.

In this paper, we aim to remedy this gap in our viewpoint set by providing an outline definition of a viewpoint to define the content of a context view. We

define the viewpoint using the same structure as we used for our original viewpoint set [1].

3. Defining the Context Viewpoint

3.1. Overview of the Viewpoint

All systems exist in some larger environment, be it a department, an organisation's IT environment, a mobile communications system or even a virtual world. The Context view aims to answer questions about this environment and specifically the technical relationships that the system being designed has with the various elements of this wider environment.

3.2. Concerns

The concerns that a context view addresses are:

- **Identity and Responsibilities of External Entities** - the key information that the Context view must define is the set of external entities that the current system interacts with in some way, the reason for the interaction and the responsibilities that the external entities are assumed to fulfil in the context of this relationship.
  It is important to make sure that external entities that the system has irregular or occasional interactions with (e.g. systems that are only polled for data at the end of each month) are defined just as carefully as those which the system interacts with continually.
  Similarly, it is important to consider and carefully define external entities that rely on this system as well as those that this system relies on (it is very easy to worry about what we need while rather neglecting what others are expecting from us!)
  Also make sure that different types of external entities are considered, including systems supplying or consuming data, systems called as services, systems that call us as services, physical entities such as reports and files and human actors who need to interact with this system.
- **External Interdependencies** - there are sometimes inter-dependencies between external entities that the system interacts with. An example could be where two systems have a data dependency between them that means that new data should always be sent to one of the systems, and acknowledged, before related data is sent to the other. These dependencies may be subtle and must be identified as part of this process.
- **Nature of the External Connections** - having defined the external entities the next concern is to decide or discover the nature of the connections with them. Connections can vary widely, from high volume messaging or RPC connections, through batch-oriented file or database interfaces, to totally manual connections involving human interaction or even document scanning. Some connections may need to be secured, some may need to implement very specific protocols. Collecting and agreeing the fundamental characteristics of each connection allows the architect to start thinking about the practical implications of them and helps to identify gaps in knowledge and potential problems..
- **Expected External Interactions** - having worked out the connections to be used, the architect needs to understand the interactions that are expected to occur over them and to agree these with the owners of the external entities. The frequency, schedule, volume and criticality of each interaction is needed to allow the design of appropriate solutions for them within the system.
- **External Interface Definitions** - at the most detailed level of concern, the architect needs to make sure that all of the system's external interfaces well defined, in the sense of making sure that data formats, interaction sequences and technical requirements such as message transports are clearly specified.

3.3. Models

Context Diagram

The context diagram is the key model within a context view, placing the system in its environment by relating it to the external actors that it interacts with via explicit relationships that represent the connections to and from it. A context diagram is usually quite simple and contains elements of the following types:

- **System** - the system being designed, treated as a "black box", with its internal structure hidden;
- **External Entities** - systems, people, groups and other entities that the system interacts with; and
- **Connections** - the links between the external entities and the system being designed.

The two notations that we commonly see used for context diagrams are UML and "boxes and lines".

The UML standard [4] doesn't define a context diagram, the assumption being that the context of the system will be captured using a "use case" diagram, with the boundary of the system being represented by a classifier (class, component or package) that contains the use cases or simply by a diagrammatic annotation such as a rectangle drawn around the use cases. However there are a number of practical difficulties with this approach including the complexity of the resulting diagram, the fact that the use case list may not be available when the context diagram is created, and the convention that the external connections will be made to specific use cases, whereas in reality we want to abstract this detail away and treat the system as a "black box" in the context view.

The solution to these difficulties is to create a UML diagram of the form shown in Figure 1.

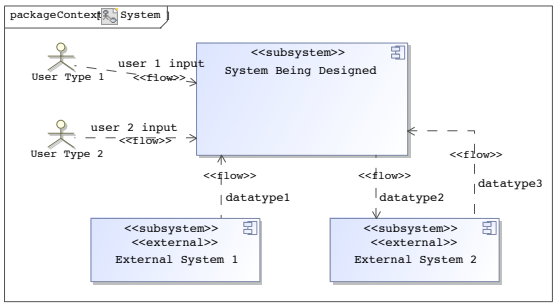

Figure 1. UML context diagram

This sort of UML diagram can be created using the "use case" diagram editor of many mainstream UML modelling tools, but in fact doesn't share a lot of similarity with the standard use-case diagram. The key points about it are:

- The system is represented as a UML component, stereotyped as a «subsystem», a stereotype found in the UML standard profile.
- External entities that cause human interactions with the system are represented as UML actors.
- External entities that are systems are represented as either further subsystem components or actors, possibly with their icons changed via stereotyping to be more representative of them (as suggested by the UML standard).
- Connections between the external entities and the system being designed can be represented as UML information flows, UML dependencies or UML associations. Space prevents us discussing these options in any detail, but briefly, the advantage of the information flow is that it allows the type of information passing over it to be made clear within the UML meta-model. The dependency and association don't allow this as easily and so this has to be stored separately or informally. The possible disadvantage of the information flow (and the dependency) is that they are uni-directional and so to represent bi-directional communication, two connections are needed (as the example in Figure 1 shows).

So, while UML can be used to create a context diagram, it would be fair to say that it doesn't offer particularly good support for it. For this reason, we often use informal "boxes and lines" notation, drawing something more akin to a "rich picture" of the system's context using a simple, ad-hoc notation (and it's obviously important to define the notation clearly). Figure 2 contains an example, which is probably familiar!

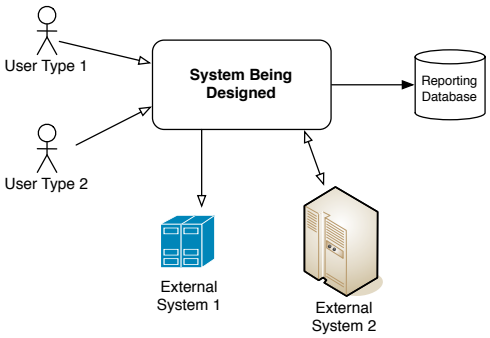

Figure 2. Informal context diagram

The advantage of the informal boxes and lines diagram is that is can be made to be much more expressive than plain UML, and it's probably easier to create in most cases. One of the major disadvantages, apart from having to design and explain the notation, is that this model (or picture) is separate to the rest of your architectural models, assuming they're in UML. However, a number of UML modelling tools can nowdraw this sort of informal picture, which largely addresses this concern.

Interaction Scenarios

As well as creating a model to clearly define the external entities and connections, it is often important to create a secondary interaction model that illustrates the expected interactions between your system and the external entities. This sort of model often helps to uncover implicit requirements and constraints (such as ordering, volume or timing constraints) and helps to provide a level of validation that is rarely achieved with a context diagram alone.

Space prevents us from explaining these models in any detail, but they are usually captured using simple textual interaction lists (rather like those used for use-case definitions) or UML sequence diagrams that illustrate the interactions via a graphical notation.

Interface Definitions

It's also worth noting that interface definitions or specifications are an important part of defining the system's context. They're usually too large and detailed to form part of the architectural description, but it would typically reference them to make interaction details clear and demonstrate that such definitions do exist for use later in the system lifecycle.

3.4. Problems and Pitfalls

Some of the common problems and pitfalls that we have observed when developing Context views are listed below, along with suggestions for resolution.

- **Uneven Focus Across Externals** - it's easy to focus on the key (or powerful) user groups or external systems that you interact with. However getting almost any of these external connections

wrong can sink a system, so they all need to be taken into account.

- **Implicit Dependencies** - there are often subtle dependencies between external entities that cause complications when interacting with them (e.g. assuming that a particular business entity is instantaneously available from two systems). It's important to check these external inter-dependencies early so that they don't cause design problems late in the day.
- **Loose or Inaccurate Connection Descriptions** - it's always tempting to get the basic idea of an external connection and leave it at that, hoping that the design process will drive out the details. In fact, you always have to do this to some extent as you can't understand every detail of every connection. However, it is important to understand enough detail so that the architectural implications can be understood.
- **Complicated Interactions** - interactions with some external entities (e.g. humans or old systems) can be a lot more complicated than expected, so it's easy to end up with unexpected problems when you come to build the interfaces.
- **Missing Data** - you can't check every field of every external connection, but you do need to understand what types of data each needs to check that you've actually got it in your system.

3.5. Checklist

When developing a context view, you can use the following list of questions as a checklist in order to check the completeness and consistency of the view.

- Do you confident that you have identified all of the external entities that the system needs to interact with and their responsibilities?
- Have you got a good understanding of the nature of the connection with each external entity?
- Is a clear interface definition available for all of the technical interfaces? (i.e. to/from other systems)
- Have you considered possible dependencies between the external entities that you have to interact with?
- Do you have a context diagram illustrating the connections from the system to its environment, with sufficient definition underpinning the diagram?
- Have you explored a set of realistic scenarios for external interactions between your system and external actors?

4. Related Work

Although quite a number of software architecture approaches don't include a context view we aren't the only authors to note the need to define the system environment and context clearly.

Garland and Anthony [2] specifically define a "Context Viewpoint" in their large set of viewpoints for information systems development, and explain how to create this view and relate it to the later parts of the software architecture process. This Context Viewpoint is quite similar to the one we define here, although it is defined using Garland and Anthony's conventions and so does not contain all of the information we present here and does differ in some of the details of the advice proffered.

The other related description we are aware of is a discussion of context diagrams in the SEI "Views and Beyond" approach and its view-types [3]. In this approach, the context diagram is used as a supplementary description, alongside the architectural views, used to provide context for a "view packet", to allow the reader to understand the scope of the architectural documentation being read. Again, there are many similar ideas in this approach to the advice we offer here, although we see the Context view as more central to the architectural description and worthy of a place as a first class architectural view in its own right. We also provide a little more information than this reference, due to our viewpoint description format.

5. Conclusions and Further Work

Experience since we developed our viewpoint set has led us to believe that that a Context view is a valuable part of almost any architectural description. Therefore we have started the process of defining a Context viewpoint intended to extend our viewpoint set and guide the creation of context views.

This short paper is only a first step in the process though, with space limits meaning that many of the descriptions above are brief outlines that need to be expanded to fully explain our ideas in this area. Future work intends to address this by building on this initial work to create a full viewpoint definition for the Context viewpoint.

6. References

[1] Rozanski, N., and E. Woods, *Software Systems Architecture*, Addison-Wesley, Upper Saddle River, 2005.

[2] Garland, J., and R. Anthony, *Large-Scale Software Architecture*, Wiley, Chichester, 2003.

[3] Clements, P., F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord and J. Stafford, *Documenting Software Architectures*, Addison-Wesley, Boston, 2002.

[4] Object Management Group (OMG), *Unified Modelling Language: Superstructure*, version 2.0, www.omg.org, 2005.