Applying Enterprise Application Architectures in Integrated Modelling

Knapen M.J.R. [1], Verweij P.J.F.M. [1] and Wien J.J.F. [1]

[1] Alterra, Environmental Sciences Group, Wageningen University and Research Centre, Wageningen
Email: rob.knapen@wur.nl

Keywords: *software architecture, information systems, information economics, model integration*

EXTENDED ABSTRACT

Software systems that support integrated policy assessment need to work with models from different domains and provide a framework to link these models together. They need to store model results, intermediate and raw data, and have one or more user interfaces for input and presentation. Such software can be built with an *inside-out* focus, emphasising the model linking, but also from an *outside-in* perspective. Here the features and technical development is primarily driven by usability and business requirements.

The inside-out view can very well satisfy the needs of the modellers and framework builders involved, but might not be enough for all other requirements of the many stakeholders in an integrated assessment, and in the project at a larger scale.

Thus the outside-in view must also be considered for the software to be successful. Looking from this perspective the software system is similar in basic functionality to other data intensive enterprise applications and common architectures and design patterns could be used for its construction.

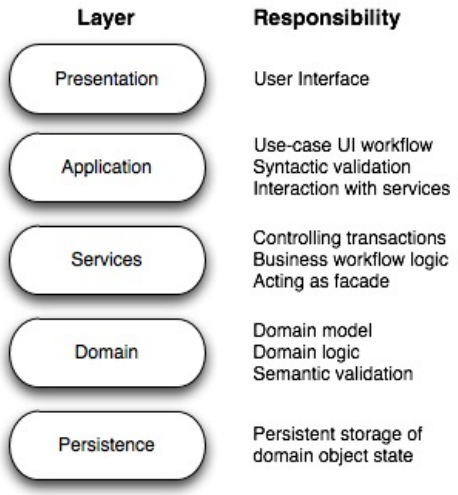

Figure 1: Layered Architecture

Enterprise applications (e.g. used for banking or insurance) typically have an architecture that separates functionality into 5 distinct layers (Figure 1). This includes a persistence layer for storage of domain object state, the domain layer for the domain model and domain logic, a services layer that controls transactions and contains the business logic, an application layer for use-case workflows, syntactic validation and interaction with the services layer and finally a presentation layer for the user interfaces. Following such separation of concerns, helped by some well known design patterns like data transfer objects, service layer, command pattern and CRUD (a pattern that organizes the persistence operations into Create, Read, Update and Retrieve operations that are implemented by a persistence layer) will improve the maintainability, possibilities for re-use and systems interoperability.

In this paper the software architecture of the SENSOR and SEAMLESS 6th framework EU projects will be used as case studies to illustrate the use of the layered architecture and the mentioned design patterns. Both these projects deal with environmental integrated assessments and include the construction of a decision support software system. They will illustrate the increased importance of the outside-in perspective and following standard software engineering practices to improve the interoperability and possibilities for re-use of components of the systems build.

INTRODUCTION

1.1. Integrated Assessments

Societies see the emergence of new governance concepts, based on the assumption that processes of planning and decision taking are no longer hierarchical but the product of complex interactions between governmental and non-governmental organizations, and the general public (the model of "co-production of knowledge" Callon, 1999). All involved are seeking to influence the collectively binding decisions that have consequences for their interests.

To account for this changing governance and the increased number of stakeholders involved, decisions need to be assessed in an integrated context. Such *Integrated Assessment* has been defined as "an interdisciplinary and participatory process combining, interpreting and communicating knowledge from diverse scientific disciplines to allow a better understanding of a complex phenomena (Rotmans and Van Asselt, 1996). It requires that information needs to be made accessible in a way such that all different types of stakeholders achieve a common understanding of the problems, objectives and possible solutions.

1.2. Resulting Requirements

Due to the nature of this integrated assessment, the multiple stakeholders involved (most likely from several domains), software systems that seek to support this process need to be able to work with different types of calculation and simulation models and provide a framework for linking them together. Integrated assessment systems therefore mostly are integrated modelling systems.

What is signifying is that these models and the data involved come from different domains and use their own language and concepts. This is not a small problem that needs to be overcome and is the focus of several research efforts. Using ontology's to resolve the issues and harmonize the knowledge concepts used in data and models is one promising solution (Wien *et al.* 2007).

However besides being able to link the models, raw data must be made accessible for these models and intermediate and final calculation results stored. And of course all this must be supported by intuitive user interfaces. Often it can not be one and more of them are needed due to the different roles each of the stakeholders can fulfil (Van der Wal *et al.* 2005).

Additionally there is a trend for follow ups once the integrated assessment system is build. It needs to be maintained, maybe re-used or connected to another *integrated* assessment system.

1.3. Multiple Views

Any software system (and its architecture) can be viewed from multiple perspectives. In fact this is needed for its design and implementation, where owner, designer and builder all look at the same system from a different point of view (Zachman, 1987).

Even within one of those points of view, e.g. that of the designer (or architect), there is more then a single perspective to consider. According to Perry and Wolf (1992) the distinction between a processing, data and connections view is rather important.

Naturally all these views are intertwined. However looking at the design and construction of an integrated assessment system we can distinguish two broad overall approaches, the *inside-out* and the *outside-in* way of thinking.

Integrated assessment systems tend to be studied and build as part of (large) research projects where the approach is mostly inside-out. This emphasises the linking of the models and perhaps the development (or use) of a framework to support it. Possibilities of the models and the created links between then, and the framework's flexibility are then put behind a (most of the time rather technical) user interface (if any).

The same system could also be thought of from an outside-in perspective. This is more of a User Centred Design (UCD) (Raskin, 2000) approach where usability and business requirements drive the features and technical development. Tell-tailing would be that the user interface would be build first, and its usefulness studied with the intended users (or user types).

The inside-out view serves mostly the modellers and framework builders, and the outside-in all the other (end) users that most likely pay (in some way or another) for the development of the system and only care to a limited level about the intricate internal machinery of connected interdisciplinary models. To them, in essence it is a data intensive software system that has to provide usable information at the right moment (timely for the decision taking process).

Data intensive software systems are well known to software engineering, for example consider the

large banking and insurance systems. These are commonly called (business) Information Systems, or Enterprise Applications.

If we regard integrated modelling systems from this perspective, would it make sens to apply to them some of the same software design rules – or software architecture, as used for other enterprise applications?

2. ENTERPRISE APPLICATION ARCHITECTURE

2.1. Enterprise Architecture

Enterprise application architecture tells us something about the design of an information system. A common mistake is to confuse it with enterprise architecture, a methodology for structuring an organization itself. However enterprise application architecture usually is part of an enterprise architecture, since information systems play a large role within organizations. The enterprise architecture at its strategic level sets boundaries for any enterprise application architecture and also provides necessary input when Information Economics (Van der Wal *et al.* 2003) are used to decide whether an (enterprise) application should be build or not.

Due to their size and importance to organizations careful planning and construction of enterprise applications is required. Zachman (1987) already noted that the increased scope of design and levels of complexity of information systems implementations is forcing the use of some logical construct (or architecture) for defining and controlling the interfaces and the integration of all the components of the system.

Over time plenty of enterprise systems have been build, traditional administrative applications come to mind first. For software design the shared aspects are studied and formalized as *software architectures* so that they can be used as a framework for satisfying requirements; as basis for cost estimations and process management; to help reuse and for (system) analysis purposes.

2.2. Software Architectures

There is not one software architecture, there are many. Even worse, architecture is relative. What you think architecture is depends on what you are doing (Zachman, 1987).

According to Perry and Wolf (1992) we can see a software architecture as defined by elements, form and rationale. It is a set of design elements (processing elements, data elements and connecting elements) that have a particular form. This form consists of weighted properties and relationships. The properties are used to constrain the choice of architectural elements, and the relationships constrain their "placement". The rationale for various choices made in defining an architecture is an integral part of it.

Where architecture is a formal arrangement of architectural elements, *architectural style* is that which abstracts elements and formal aspects from various specific architectures. For example a distributed style or a multi-processor style. There is no hard dividing line between architecture and architectural styles.

Architecture helps in *comparing software systems* and *reusing components*. Possibilities for re-use are the greatest where specifications for the components are constrained the least – at the architectural level. Component re-use at the implementation level is usually too late because the implementation elements often embody too many constraints. Moreover, consideration of re-use at the architectural level may lead development down a different (but still valid) solution path.

From the many existing software architectures there is one that is very often used for enterprise applications, the *layered architecture*. "Layered" applies mostly to the form aspect of the architecture. It will be the focus of the remainder of this paper to see how well it can be applied to systems for integrated assessment.

2.3. Layered Architecture

The principle of dividing a software system into layers reaches back to the early days of computer science (Dijkstra, 1968). Originally the idea was used for the design of operating systems but it applies equally well to other types of software.

In the layered software architecture the system's functionality is usually separated in up to 5 layers (see Figure 1). Included are a persistence layer for storage of domain object state, a domain layer for the domain model and domain logic, a services layer that controls transactions and contains the business logic, an application layer for use-case workflows, syntactic validation and interaction with the services layer and finally a presentation layer for the user interfaces.

As a general rule each layer has only dependencies on those below it, not above, limiting the effect of changes and increasing maintainability. In a way

each layer acts as a client to the tier below and as a server to the tier above.

All layers can be located on a single computer, or they can be divided amongst a number of systems. For example the top one or two layers (presentation and application) can be filled in with web-based clients (see 2.4), or the persistence layer can be implemented by a relational database running on a different server. This is referred to as a multi-tier architectural style.

A well known two-tier architecture style is that of the client-server model used for dumb terminals connected to large time-sharing mainframe systems.

2.4. Clients

The client-server model is still in use today, not only for mainframe systems but also for remote systems that provide services over the Internet. Think of e-mail clients using mail storage servers or applications running inside a web browser.

Confusingly “client” is used for software as well as hardware. In general clients are classified as "fat clients", "thin clients", or "hybrid clients".

	Local storage	Local processing
Fat Client	Yes	Yes
Hybrid Client	No	Yes
Thin Client	No	No

Table 1: Types of Clients

A *fat client* (also known as a thick client or rich client) is a client that performs the bulk of any data processing operations itself, and does not necessarily rely on the server.

A *thin client* is a minimal sort of client, its functionality limited to the presentation layer. Thin clients use the resources of the host computer. A thin client's job is generally just to graphically display pictures provided by an application server, which performs the bulk of any required data processing.

A *hybrid client* is a mixture of the above two client models. Similar to a fat client, it processes locally, but relies on the server for storage data. These are also known as *rich clients* and implement both presentation and application layers.

In designing a multi-tier architecture, there is a decision to be made as to which parts of the task should be done on the client, and which on the server. This decision can crucially affect the cost of clients and servers, the robustness and security of the application as a whole, and the flexibility of the design for later modification, porting and re-use.

2.5. Design Patterns

The layered software architecture and its multi-tier style is a common solution to a recurring problem. It is known as a (architectural) design pattern (Fowler, 2007). Other design patterns exist, not only for the architectural level. These patterns provide a mechanism for providing design advice in a reference format. A classical well-known book about design patterns for software systems is that of the Gang of Four (Gamma *et al*. 1994).

In the next paragraphs a few design patterns will be further described because of their significant relevance to the integrated assessment systems discussed. In particular these are:

- *Data Transfer Objects*: A way to optimise remote method calls.
- *Service Layer*: To centralize access to business logic.
- *Messaging*: For exchange of information between applications.
- *CRUD*: A method for data persistence.

These design patterns apply mostly to the design elements of the architecture.

Data Transfer Objects

A data transfer object (DTO) is an object that holds all the data required for a call to a remote interface. Typically this is a call from one layer to a lower layer, which might not be on the same system.

Such a call will be expensive (considering time and other resources required) so the number of calls should be minimized. Instead of using a large number of parameters all the data is passed in a DTO. DTO’s must be serialized to go across a connection (to another system).

A factory class (or encoder / decoder) can be used to create DTO’s from domain objects and vice versa, illustrated by Figure 2. This separates the required logic from the rest of the system.

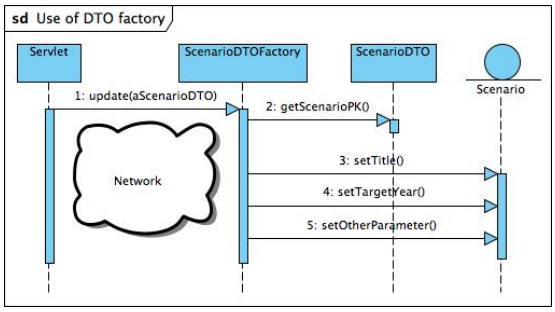

Figure 2: DTO Factory

Service Layer / Command pattern

Enterprise applications typically require different kinds of interfaces to the data they store and the logic they implement: data loaders, user interfaces, integration gateways, and others. Despite their different purposes, these interfaces often need common interactions with the application to access and manipulate its data and invoke its business logic. The interactions may be complex, involving transactions across multiple resources and the coordination of several responses to an action. Encoding the logic of the interactions separately in each interface causes a lot of duplication.

A Service Layer defines an application's boundary and its set of available operations from the perspective of interfacing client layers. It encapsulates the business logic, controlling transactions and coordinating responses in the implementation of its operations.

Several patterns exists to implement a service layer, e.g. the Session Façade pattern or the Command pattern illustrated in the following diagram:

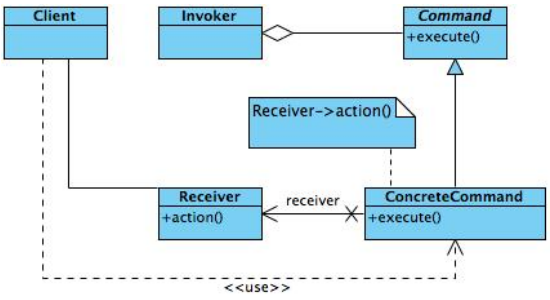

Figure 3: Command Pattern Classes

Messaging

Messaging is an important aspect of distributed (multi-tier) software architectures. And not only is it relevant within a single application, it also plays a role in integrating several applications. Messaging can be synchronous or asynchronous, and usually it uses some kind of middleware that provides the "plumbing" such as data transport, data transformation and routing. An example of such middleware is the *Message Bus*: a combination of a common data model, command set, and a infrastructure to allow different systems to communicate.

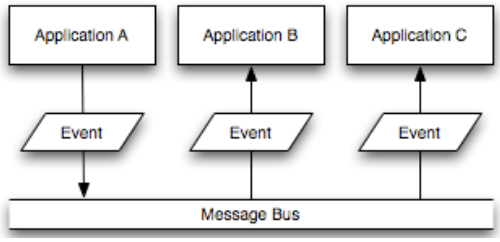

Figure 4: Message Bus

Asynchronous messaging is fire-and-forget information exchange. The sender of the message does not have to wait for a response from the recipient because they can rely on the middleware to ensure delivery (of the request and eventually the response). Unlike synchronous messaging it does not rely on direct connections.

Representational State Transfer (REST; the architectural style of e.g. HTTP; Fielding, 2000) and web services are examples of asynchronous messaging and a good strategy for integration of enterprise applications. They promote a loosely coupled solution and force the developers to recognize that working with remote applications is slower. This also encourages design of components with high cohesion (local processing) and low adhesion (remote work).

CRUD pattern

Almost all applications include some form of persistence storage and have to perform Create, Retrieve, Update and Delete (CRUD) operations on it (Kilov, 1990). This is the task of the persistence layer. However for most software it is also important for the user interface that should at least allow these operations on some of the domain objects.

3. CASE STUDIES

The discussed layered architecture with the multi-tier style, and the design patterns for DTO, service layer, messaging and CRUD appear to be useful for building integrated assessment applications. In this chapter two of such projects / applications will be investigated to see how the architectural elements presented here are applied. The

SEAMLESS and *SENSOR* projects are both long term (4 years) EU projects, approaching their final research and development phase.

3.1. SEAMLESS

Project

SEAMLESS (www.seamless-ip.org), a EU-FP6 Integrated Project, aims at generating an integrated framework of computer models. This framework can be used for assessment of how future alternative agricultural and environmental polices affect sustainable development in Europe.

Software

Within the project the second major prototype of the software (SEAMLESS-IF) as been delivered (2007). It completes the transition from a desktop application (the first prototype) to a multi-tier (web based) application. This also will be the architecture for the final version. The following diagram illustrates this architecture:

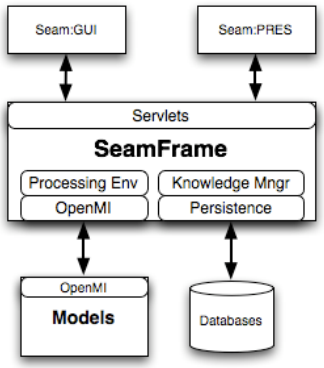

Figure 5: SEAMLESS Architecture

In the top line some possible applications are shown that implement user interfaces, and fill in the presentation and application layers of the system. As part of SEAMLESS-IF the Seam:GUI and Seam:PRES will be developed. These hybrid clients use REST (at the moment implemented through stateless servlets) to communicate with SeamFrame, the server component. SeamFrame implements the Service Layer. DTO's and the command pattern are used for this. An OpenMI (www.openmi.org) based processing environment is part of SeamFrame and handles the linked models.

SeamFrame uses the domain model and classes generated for it from the SEAMLESS ontology by the knowledge manager. Through a Hibernate based object relational mapping the domain model is stored in databases.

3.2. SENSOR

Project

The EU-FP6 Integrated Project SENSOR (www.sensor-ip.org) will develop science based ex-ante Sustainability Impact Assessment Tools (SIAT) to support decision making on policies related to multifunctional land use in European regions.

Sustainability of land use in European regions is a central point of policy and management decisions at different levels of governance. Implementation of European policies designed to promote and protect multifunctional land use requires the urgent development of robust tools for the assessment of different scenarios impacts on the environmental, social and economic sustainability in European regions. SENSOR will built, validate and implement *Sustainability Impact Assessment Tools (SIAT)*, including databases and spatial reference frameworks for the analysis of land and human resources in the context of land use policies for Europe.

Software

The first major prototype of the SENSOR software was, like SEAMLESS first prototype and NitroEurope a single system desktop application with limited layered architecture.

Architectural design for the second prototype is at the time of writing still under consideration. First drafts indicate also a transition to a multi-tier layered architecture style based on web services. Main reasons for this are integration with applications from other SENSOR project partners (WMS, 3D landscape web services), integration with SEAMLESS, re-use of SEAMLESS components and simplified software distribution.

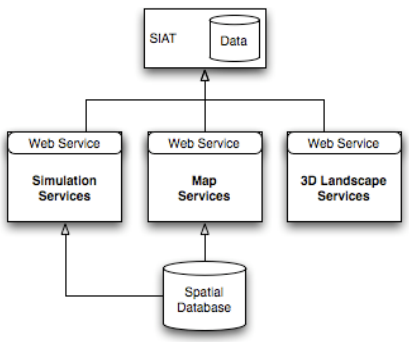

Figure 6: SENSOR Architecture

4. CONCLUSION

The layered software architecture found in many enterprise applications and the design patterns described here proof to be useful for integrated assessment systems. They help structuring the software when looking at it from the outside-in.

From the network perspective both projects show an increase use of the simpler RESTful web services over full SOAP implementations. The reduced complexity improves the maintainability of the systems and re-usability of its components.

It is also clear that the rationale is an integral part of the initial architecture. Both the projects (SEAMLESS and SENSOR) move from a desktop application to a multi-tier implementation, because they see added value in that. For possible re-use, future follow-up projects or integration. Organizations can use information economics in taking these decisions, but this leads back to a need for enterprise (application) architecture.

5. ACKNOWLEDGEMENTS

The sixth framework programme for Research and Technological Development (FP6) is a collection of the actions at EU level to fund and promote research. With a budget of 17.5 billion euros for the years 2002 - 2006 it represents about 4 to 5 percent of the overall expenditure on RTD in EU Member States. The main objective of FP6 is to contribute to the creation of the European Research Area (ERA) by improving integration and co-ordination of research in Europe.

6. REFERENCES

Callon, M. (1999), The Role of Lay People in the Production and Dissemination of Scientific Knowledge. *Science, Technology, and Society*, 4(1), 81-94.

Dijkstra, E.W. (1968), The structure of the 'THE'-multiprogramming system, *Communi-cations of the ACM* 11(5): 341 – 346.

Fielding, R. (2000), Architectural Styles and the Design of Network-based Software Architectures. *Dissertation. University of California. http://www.ics.uci.edu/~fielding*

Fowler M. (2007), Patterns of Enterprise Application Architecture. Addision Wesley, ISBN: 0321513754.

Gamma E., Helm, R., Johnson R. and Vlissides J. (1994), Design Patterns: elements of reusable object-oriented software. Addison Wesley, ISBN: 0201633612.

Kilov, H. (1990), From semantic to object-oriented data modeling. *Proceedings of the First International Conference on Systems Integration*. 23-26 Apr 1990, Page(s): 385 – 393.

Perry, D.E. and Wolf, A.L. (1992), Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17:40--52, October 1992.

Pulkkinen, M. (2006), Systematic Management of Architectural Decisions in Enterprise Architecture Planning. Four Dimensions and Three Abstraction Levels.

Raskin, J. (2000), The Humane Interface: New directions for designing interactive systems. ACM Press. ISBN: 021379376.

Rotmans, J. and M. B. A. van Asselt (1996), Integrated assessment: a growing child on its way to maturity. *Climatic Change.* 34: 327-36.

Van der Wal, T., Wien, J.J.F., Otjens, A.J. (2003), The Application of Frameworks increases the Efficiency of Knowledge Systems. *Proceedings of the 4th EFITA conference on ICT in agriculture.*

Van der Wal, T., Knapen, R., Svensson, M., Athanasiadis, I. and Rizzoli, A.E. (2005), *Trade-offs in the design of cross-disciplinary software* systems. In Zerger, A. and Argent, R.M. (eds) MODSIM 2005 International Congress on Modelling and Simulation. Modelling and Simulation Society of Australia and New Zealand, December 2005 ISBN: 0-9758400-2-9. Pages 732-737.

Wien, J.J.F., Knapen, M.J.R., Janssen, S., Verweij, P.J.F.M., Athanasiadis, I.N., Li, H., Villa, F. and De Zeeuw, C.J. (2007), Using ontology to harmonize knowledge concepts in data and models. *MODSIM 2007 International Congress on Modelling and Simulation.* Modelling and Simulation Society of Australia and New Zealand, December 2007. Submitted.

Zachman, (1987), A Framework for Information Systems Architecture. *IBM Systems Journal, vol. 26, no. 3, 1987.* IBM Publication G321-5298.