# Aligning Application Architecture to the Business Context

R.J. Wieringa, H.M. Blanken, M.M. Fokkinga, and P.W.P.J Grefen

Center for Telematics and Information Technology, University of Twente, the Netherlands
(roelw|blanken|fokkinga|grefen)@cs.utwente.nl

**Abstract.** Alignment of application architecture to business architecture is a central problem in the design, acquisition and implementation of information systems in current large-scale information-processing organizations. Current research in architecture alignment is either too strategic or too software implementation-oriented to be of use to the practicing information systems architect. This paper presents a framework to analyze the alignment problem and operationalizes this as an approach to application architecture design given a business context. We summarize guidelines for application architecture design and illustrate our approach and guidelines with an example.

## 1 Introduction

Alignment of application architecture to business architecture is a central problem in the design, acquisition and implementation of information systems in current large-scale information-processing organizations. Current research in software architecture [1–3] focuses on the architecture problem for the software engineer rather than for the architect of large-scale information systems. These large-scale systems contain components that are bought rather than programmed. Their architecture problem exists on a higher aggregation level than that studied in software engineering. At this higher level architects are interested not only in the extent to which an architecture supports quality attributes, but also the extent to which the application architecture fits within the business context—the architecture alignment problem.

The business-IT alignment problem has been studied at a rather strategic level that is hard to operationalize for the practicing application architect [4]. There is a need for body of operational concepts and guidelines that encompasses business architecture as well as application architecture. In this paper we present such a framework. Its first version was the result of an extensive analysis of design methods and frameworks in software engineering, product engineering and systems engineering [5, 6]. We tested, simplified and elaborated the framework in a number of standard examples from the literature and then applied it to a large number of M.Sc. projects. Next, we applied it to real-life projects. Currently, we are using it in an empirical study to collect and analyze best practices in architecture alignment in banks, insurance companies and government organizations.

One contribution of this paper lies in the definition of an integrated and unified framework for business process design and application architecture design, in which both the business and its application software are viewed as reactive systems, i.e. as systems that respond to events in their environment. Our framework refines and operationalizes a number of other frameworks, such as that of Henderson and Venkatraman [4] and of Zachman [7]. The second contribution is that we collected and validated a comprehensive set of guidelines for defining an application architecture.

In section 2 we present a our view on architecture and in section 3 we present our framework. Section 4 describes our architecture design approach. Section 5 gives an example and section 6 summarizes the architectural alignment guidelines we have collected. Section 7 discusses the results achieved so far and presents some directions for further work.

## 2 A Systems View on Architectures

The most widely used definition of architecture is that it is the structure, or a set of structures, of a system, consisting of elements and relations between these [1]. We add to this the requirement that the collection of elements as a whole has an added value for its environment, that the elements do not have if taken separately. So our definition of architecture is as follows.

> The **architecture** of a system is the structure, or a set of structures, of a system, consisting of elements and relations between these, *such that the relations between the elements create an overall coherent system with an added value for its environment.*

This adds a systems view to the definition, where a "system" is any coherent collection of elements whose interaction produces an added value for its environment. This includes information systems and workflow management systems, but also businesses and other kinds of organizations.

A systems view of architectures allows us to apply the definition to business systems as well as software systems. It also makes clear that in order to design a system architecture, we must study the desired added value for its environment first.

## 3 The Framework

### 3.1 System Aspects

An important part of our framework consists of a classification of system properties or *aspects* as they are also called. Borrowing from a rich set of frameworks developed in systems engineering and industrial product design, we propose the classification of information processing system properties shown in figure 1 [8],

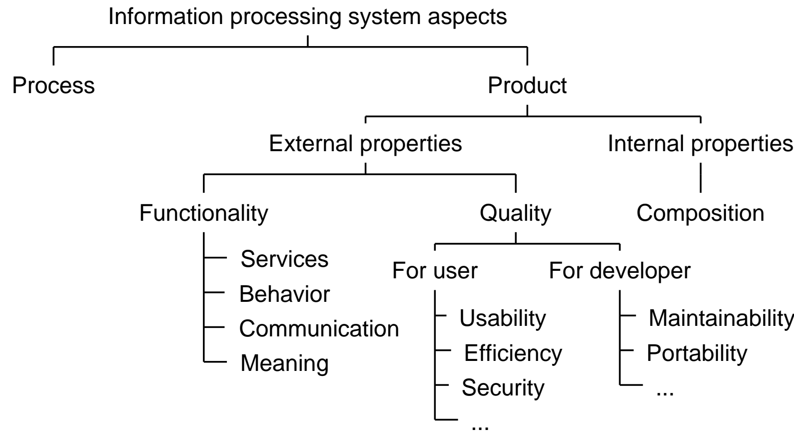



**Fig. 1.** Process aspects and product aspects.

[9], [5]. Each of the nodes in the tree represents an aspect of an information processing system, in other words a possible point of view from which to consider the system. The rationale behind this framework is as follows.

We start from the classic distinction between *process* on the one hand, and the *product* that results from this process on the other. The process of developing and exploiting an information processing system has an architecture, just as the result of this process, an information processing system, has an architecture. These architectures are related, because for example the composition of an information system determines the work breakdown structure of the development and exploitation process. However, the process architecture and product architecture are distinct because they are architectures of different things.

Our interest is in product architecture, where the product may be an information system, a business, or any other information processing system. (An information processing system is any system that manipulates symbols.)

The top-level distinction of product properties is between *external* and *internal* properties. External properties in turn are classified according another well-accepted partition, namely between *functional properties* and *quality properties*. Functional properties are services offered to the environment, and quality properties characterize the value that the system has for stakeholders. For example, usability, efficiency and security are aspects of the value that system services have for for users of the system, and maintainability and portability are aspects of the value of the system for developers.

The basic aspect of functionality is the *service* that the system has for its environment. (The word "function" in this paper is synonymous with "service". The word "functionality" in this paper means roughly "what a product does".) The system exists to deliver certain services to its environment. System services in turn are characterized by three functional properties. The *behavior* aspect con-

sists of the ordering of services over time. The *communication* aspect consists of the interactions with other entities (people, devices, businesses, software) during the delivery of the service, and the semantic aspect consists of the *meaning* of the symbols exchanged during the service.

The meaning aspect is the only aspect typical for information processing systems. Other kinds of systems deliver services by means of physical processes such as the exchange of heat or electricity, that do not have a meaning. Information processing systems deliver services by exchanging symbols with their environment, and these have a meaning (usually documented in a dictionary).

Turning now to the internal properties, the *composition* of the product, we observe that our classification of external properties repeats itself at lower levels in the aggregation hierarchy. Figure 2 illustrates this for the case where all external entities are information processing entities (whose interface is symbolic). Note that a designer does not have control over all aspects of a system. This

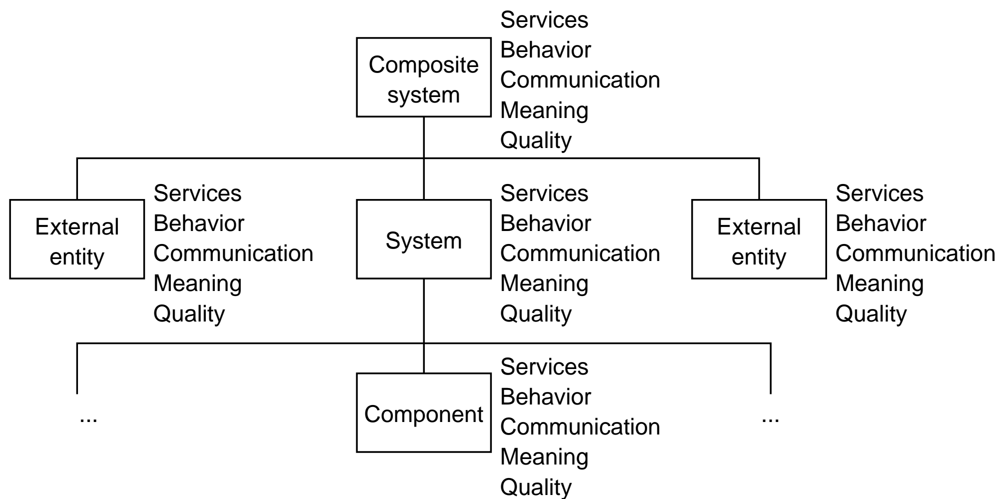



**Fig. 2.** Repetition of aspects at all levels of the aggregation hierarchy.

depends upon the design charter as discussed in section 4.

### 3.2 Service Levels and Refinement levels

Information system architects must deal with three worlds, namely the physical, social and linguistic worlds (figure 3). The physical world is the world of entities that have weight, consume energy and produce heat and noise. The social world is the world of business processes, needs, added value, needs, money, norms, values etc. Part of the social world is the linguistic world of symbol manipulation. We treat this separately because it is the world of software and documents. Note

that software exists only in the linguistic world; computers exist only in the physical world; and people exist in all three worlds.

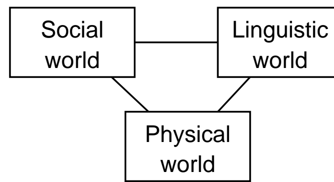



**Fig. 3.** Three alignment problems.

As suggested by figure 3, there are three alignment problems, not one. For example, to align software (in the linguistic world) to people (in the social world), we must ensure that the meaning attached by people to the symbols at the software interface agree with the manipulations of these symbols by the software, and that these manipulations have value for the people. To align software to the physical world, we must allocate it to processing devices with a location in the physical world, and to align this in turn to the social world we must align information processing devices to business processes. None of these alignment problems is trivial.

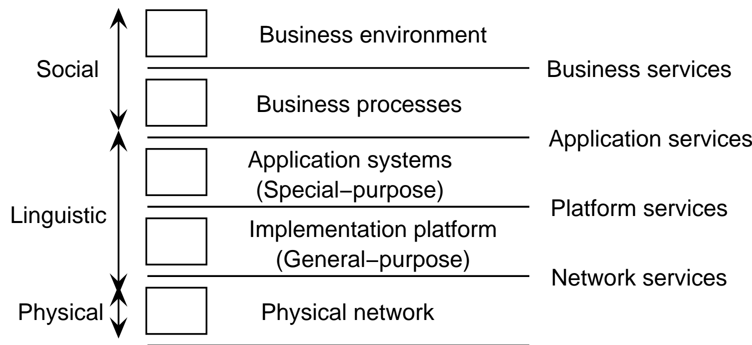



**Fig. 4.** Systems exist at all layers. At each layer, systems have the product aspects listed in figure 1.

A useful view on these alignment problems is to structure them in layers of service provision. Different situations will call for different layering structures, but we found that the one shown in figure 4 is a useful starting point in many

cases. Each particular project may feel a need to add or delete particular service provision layers. In figure 4, we partitioned the linguistic layer into two, one of general-purpose software entities, that are usually bought and not built by a business, and another of special purpose software entities, that typically are customized or built from components. The social layer is also partitioned into two, namely the business and its environment. Each layer consists of entities that provide services to entities at higher layers. In general, entities at one layer use services of entities at lower layers and provide services to entities at higher layers.

Each entity may itself be decomposed in components not accessible to other entities. We call this *encapsulation.* The aggregation hierarchy shown in figure 2 represents the decomposition of an entity into components encapsulated by it. Each of the entities at any layer, and each of its internal components, has the aspects described earlier.

Each horizontal line in figure 4 between two layers represents service provision. The diagram represents a service-provision hierarchy, but hides the fact that the social world also interacts with the physical world directly. At each layer, entities exist that have all the properties listed in figure 1, except at the physical layer, where interactions of entities do not have a meaning.

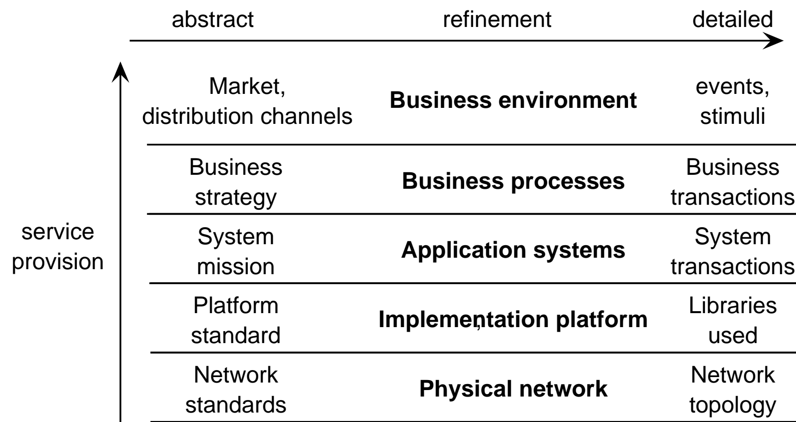



**Fig. 5.** At different levels, we can have different levels of detail in which we are interested.

Finally, we add the refinement dimension (figure 5). The refinement dimension does not classify systems, but it classifies *descriptions* of systems. Entities at each service level can be described at a high level of abstraction (few details) or at a high level of refinement (many details). Figure 5 shows some illustrative properties at different refinement levels.

To summarize, our architecture framework consists of

1. a structuring of systems into service provision layers (figure 5), where
2. at each layer, entities exist with properties that can be classified as shown in figure 1, and where
3. these properties can be described at many different levels of refinement (figure 5).

### 3.3 Comparison with other frameworks

Our framework refines the alignment framework of Henderson and Venkatraman [4]. They distinguish two dimensions, the service provision dimension (IT infrastructure level and business level) and a refinement dimension (strategic and operational levels).

Most frameworks for software system development distinguish three views, namely the function view, the behavioral view, and the data view of the system (e.g. [10]). These views correspond to our service, behavior and meaning aspects. Harel and Pnueli add to this the architecture dimension, which corresponds to our composition dimension [11].

Kruchten's 4+1 model [12] defines the logical and process views of a software system, that correspond roughly to our decomposition dimension and behavior view, respectively. His physical and development view correspond roughly to two implementation platform descriptions, namely allocation and module architecture. These are discussed in the next section.

Zachman distinguishes three kinds of descriptions, the data, process, and network description [13, 7], which correspond, roughly, to our meaning, behavior and communication aspects. These descriptions can be used according to Zachman to describe the system from a great number of perspectives, namely the scope of the system, the business view, the system model, the technology model, the component model, people, external business events, and business goals. This seemingly unrelated and arbirary list of perspectives can be systematized by placing them at various levels of service provision and refinement on our framework figure 5. Details of this and other comparisons can be found elsewhere ([5, pages 329–330], [6]).

## 4 Our Design Approach

Figure 6 illustrates our design approach. Compared to the layers in figure 4, the business service interface and the application service interface have both been expanded to a separate layer in figure 6. The figure lists a number of architecture descriptions typically found in business modeling and information systems design. We explain these descriptions in section 5. We should point out immediately that we do not claim that all these descriptions must be produced for all software systems. Rather, each particular project will produce a few of these descriptions. The problem which descriptions to choose in which circumstances is a topic of current research. The diagram is intended to illustrate an apparent contradictory dependency among design decisions. We give two examples, that support seemingly contradictory conclusions.

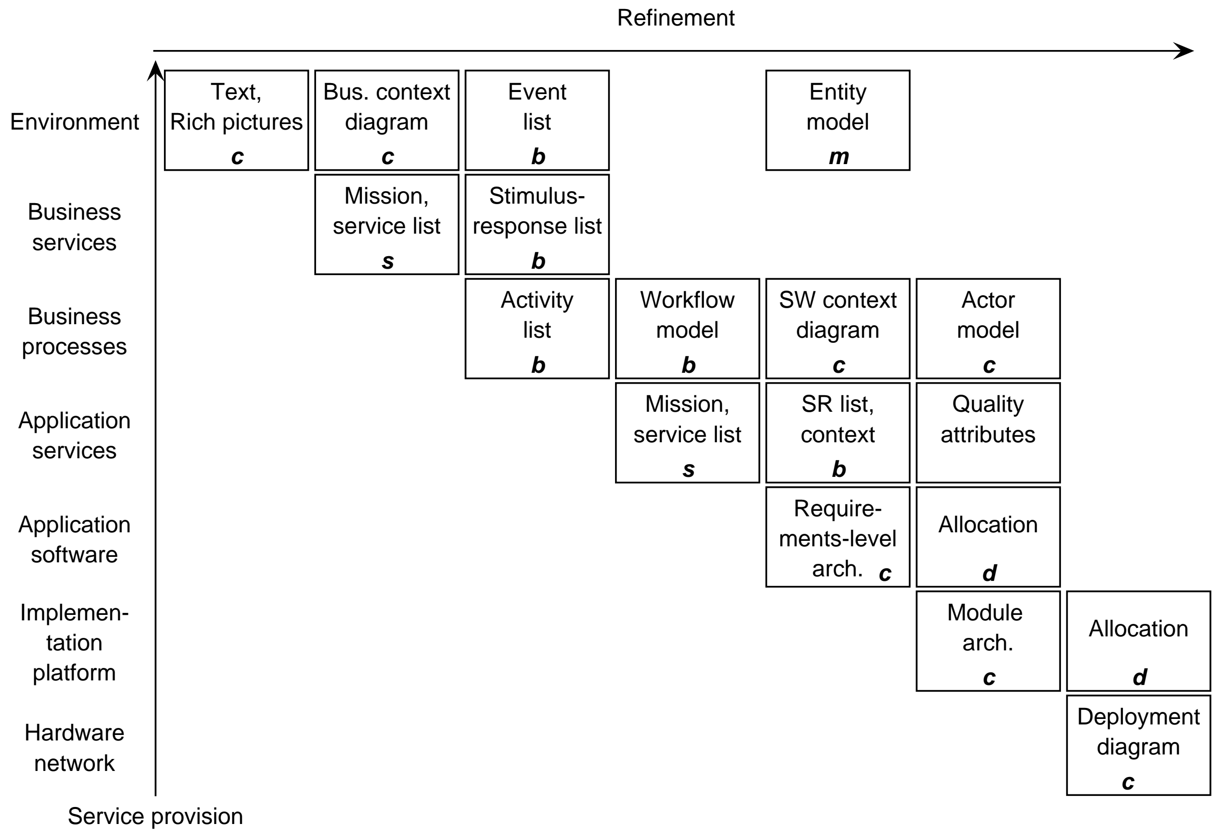



**Fig. 6.** Overview of some descriptions. **s** = service view, **b** = behavior view, **c** = communication view, **m** = semantic view (meaning), **d** = decomposition view.

- To make a list of business activities (third description along the diagonal), we need a stimulus-response list of the business first; and to make that list, we need a list of relevant events in the environment of the business first. So in order to write a description of a lower-level entity, we need descriptions of higher-level entities first.
- Once we have described the market and distribution channels of the business at a high level of abstraction by a piece of narrative text supplemented by a rich picture (top-left description), then it is not useful to add detail to this environment description by making a business context diagram and a business event list, because at this point, we do not know which part of the context is relevant and which events are relevant. We know which part of the context is relevant, after we have a list of business services (second description along the diagonal) and we know which events are relevant once we know which stimuli can arrive at the business boundary. So to refine a description of a higher-level entity, we need descriptions of lower-level entities first.

The paradox is resolved if (1) we develop descriptions in one column of the table together and (2) develop a refinement of a description only after we know which lower-layer service descriptions we need this refinement for.

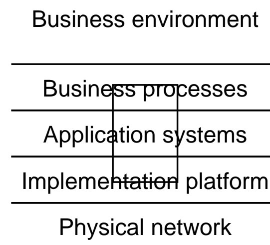



**Fig. 7.** The design charter. The area in the box is what the designer can change. The box circumscribes design freedom.

The second element of our design approach is that we start from an identification of the design charter, which is the circumscription of the area that the designer is allowed to change. Figure 7 gives an example. Outside the box, entities are given to the designer as they are, whether she likes it or not. Inside the box, entities may be changed withing the limits set by the design charter. Figure 7 illustrates that the design charter may span several service layers but need not include all entities at any layer. In practice, the situation is not so neat, because the charter may be spread out over many layers and skip other layers; indeed, the designer may have difficulty finding a crisp boundary of the design charter at all, and in addition, during the project, a boundary thought to be clear may change after all. This makes it all the more important to make

negotiations about the boundaries of the design charter an integrated part of the system design project.

In passing, we note that the overview of descriptions in figure 6 indicates for each description which viewpoint is described. For each viewpoint, there are well-known description techniques available. For example, behavior can be described by event lists, state transition tables and state transition diagrams; communication can be described by various box-and-arrow diagrams; and decomposition can be represented by traceability tables. Notations are not the subject of this paper. A thorough survey of available techniques and guidelines for their use is given elsewhere [14].

## 5 Example

We illustrate our alignment approach with a case study we did for a small company, that we will call Global Travel International (GTI). GTI provides database and web-page hosting services to the travel business in a small country in North-Western Europe. Our design charter says that we may design an application to support the primary processes of GTI, but we must not change business processes nor can we change the implementation platform or underlying hardware network. To make application architecture design decisions, we start with a number of descriptions of the business environment and business processes.

The market of GTI is described in figure 8. This is an example of a top-level business environment description (top-left box in figure 6).

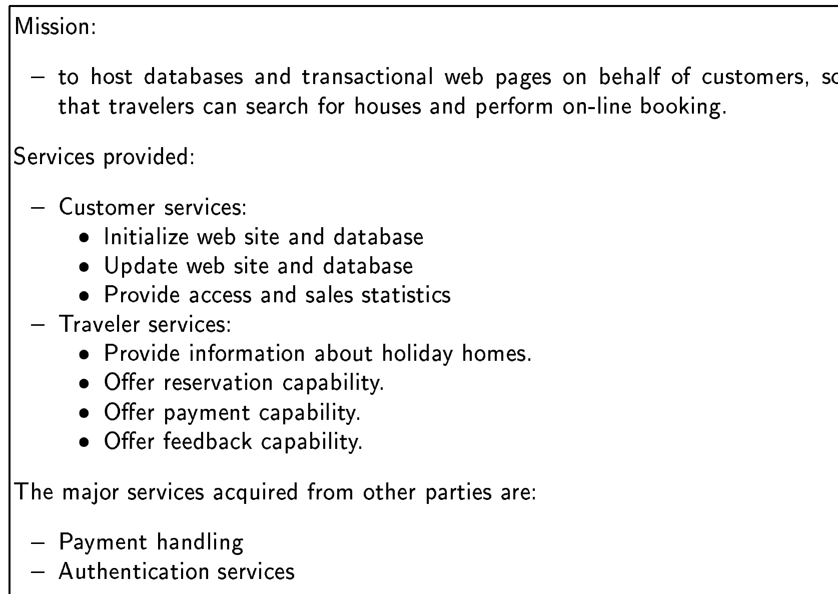
The market of GTI is database and web page hosting to facilitate on-line renting of holiday homes.

- **Customers** are sellers or suppliers.
- **Sellers** of GTI are private or corporate individuals seeking to rent out holiday homes. Examples are travel agencies, call centers, travel portals. One seller can act on behalf of many home owners.
- **Suppliers** are private or corporate entities that own homes that they offer for rent by travelers, either directly or through sellers. Examples are private home owners, tour operators and national tourist bureaus.
- **Travelers** are private individuals seeking to rent holiday homes.

Current threats are the dip in the travel industry and in Internet business. The largest opportunity is that GTI has the software to help other companies take the next large step in e-business.

**Fig. 8.** Market of GTI.

Descending to the business service layer in figure 6, we give the business mission of GTI in figure 9. This allows us to focus on the relevant part of the


Mission:

- to host databases and transactional web pages on behalf of customers, so that travelers can search for houses and perform on-line booking.

Services provided:

- Customer services:
  - Initialize web site and database
  - Update web site and database
  - Provide access and sales statistics
- Traveler services:
  - Provide information about holiday homes.
  - Offer reservation capability.
  - Offer payment capability.
  - Offer feedback capability.

The major services acquired from other parties are:

- Payment handling
- Authentication services

**Fig. 9.** Mission and external functions (services) of GTI.

context. The context diagram in figure 10 adds detail to the market description of figure 8 and at the same time provides the context in which to interpret the mission statement. Conversely, the mission statement explains why this part of the context is relevant.

Having identified the business context, we can go on to identify business services. Since we are not designing the business but describing it, this is a knowledge problem. Let us assume that the list of business services is known. From the list of business services we go on to make a coarse activity model of the business (third box along the diagonal). We identify a business activity by (1) the business event that triggers it and (2) the service delivered by it. This requires us to make a list of relevant business events and, if necessary, the list of stimuli at the business interface that each event may cause. For example, a customer's desire to rent a holiday home may be communicated to the business by telephone, fax, email or web page, so that one event (the creation of the wish to rent a holiday home) may be communicated to the business through four possible stimuli. Due to lack of space we abstract from multichanneling and restrict ourselves to business events only. Figure 11 summarizes the service list, event list and activity list of GTI.

So far, we have collected information about the business and summarized this in a number of descriptions. Now we approach the area within our design charter and we perform our first design activity, which is to gather activities into groups that contribute to related business services. We call these groups *busi-*

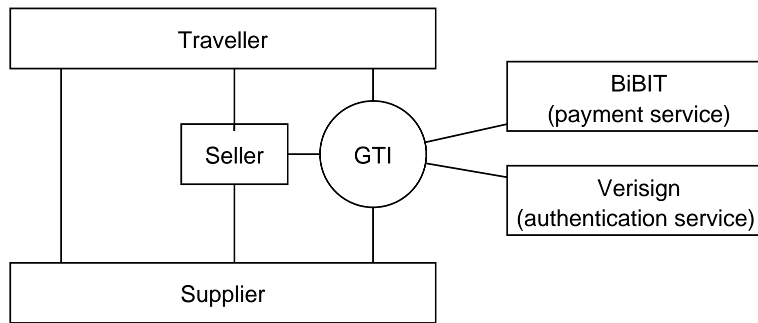



**Fig. 10.** The business context diagram of GTI.

| Service \ Event | Traveller requests booking | Traveller pays | Final payment too late | Traveller cancels booking | Traveller complains | Traveller provides experiences | New customer acquired | Customer changes products | Time to provide supplier with occupation data |
|---|---|---|---|---|---|---|---|---|---|
| Booking | Compose, Check, Reserve booking. | | Release booking. | Release booking. | | | | | |
| Payment | Accept payment | Accept payment. Clear payment | Send reminder. Reimburse. | Reimburse | | | | | |
| Feedback | | | | | Treat complaint | Collect feedback. | | | |
| Initialize customer web site | | | | | | | Create database and web site | | |
| Maintain web site | | | | | | | | Update | |
| Provide statistics | | | | | | | | | Derive data |

**Fig. 11.** GTI processes. The upper row lists events in the business environment. The leftmost column lists GTI services. The entries lists the activities in the business process triggered by events and delivering services.

*ness responsibilities.* Each business responsibility is an aggregate of a number of business services offered to the business environment. The business responsibilities we identified for GTI are indicated by the fat rectangles in figure 11. From left to right along the diagonal, these are Booking, Traveler feedback, Web site maintenance and Information provision to customers. The business services within one business responsibility are related because they serve one process in the business environment. As we will see in the next section, this satisfies several design guidelines and we therefore have good reason to believe this is a stable structure.

Grouping business activities into business responsibilities does not change the business processes, but it defines a structure that we can use to define our application architecture. The general design principle is that we must structure the system according to those elements in the business that are expected to remain stable during the life of the system.

Our grouping of business activities into responsibilities leads us to our first design decision about the application architecture: We decide to define one application for each business responsibility. In addition, we decide to introduce a component that deals with the web interface and some databases to contain data about the subject domain. The guidelines behind these decompositions are discussed in section 6. Applying these guidelines, we get the requirements-level architecture of figure 12. This is called a requirements-level architecture because

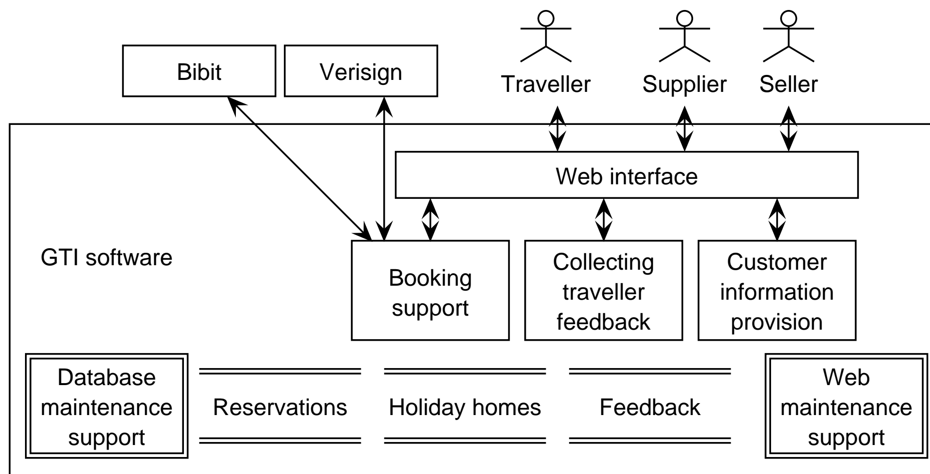



**Fig. 12.** Requirements-level application-architecture. Components with double borders have interfaces to all other components.

it is defined in terms of the functional requirements and business context, but not in terms of the available implementation platform. This architecture is also

called conceptual architecture [1], [2] and an old term for it is essential systems model [15].

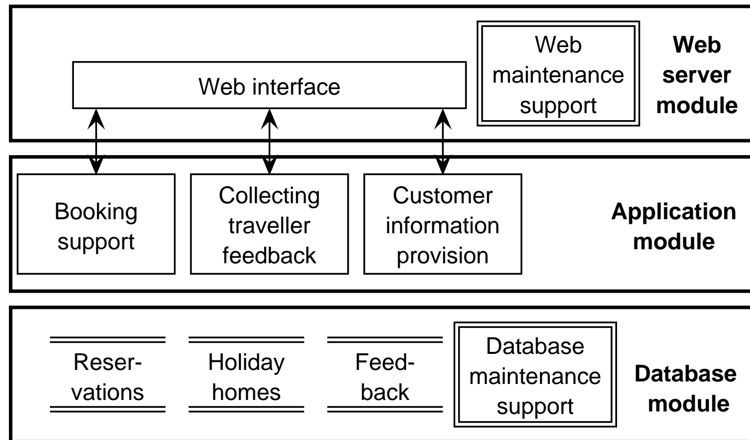



**Fig. 13.** Implementation-level modules.

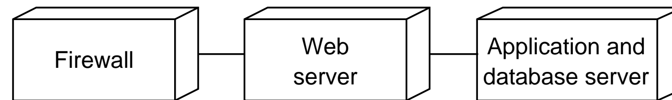



**Fig. 14.** Allocation of modules to nodes in the deplyment network.

To transform this into an implementation architecture, we must define modules and map these to nodes in the hardware network. Both the hardware network and the general-purpose implementation platform are given to us, not designed by us. They fall outside our design charter. The network consists of several PCs connected to the Internet behind a firewall that runs on a separate PC, and the implementation platform consists of a DBMS with a 4GL and a web server. This allows us to group the requirements-level components as shown in figure 13 into modules that run on this platform. Balancing PC load against data traffic, we map these to the network as shown in figure 14. Note that this decision requires an analysis of desired quality attributes. As indicated in figure 6, the allocation of requirements-level components to modules, and of modules to nodes in the network, requires a description of desired quality attributes.

## 6 Alignment Guidelines

So far, we have presented an architecture framework that includes both the business and the supporting information technology, we defined a design appropach in terms of this framework, and we illustrated this by means of an example. How does this help us to improve alignment of application architecture to business architecture?

Our framework helps solving the alignment problem because it yields us a number of simple architecture design guidelines. We discuss these guidelines here, explaining how we used them in our example.

- **Functional decomposition.** For each service to be delivered, a component is defined. For GTI, this would lead to one software component for each of the business services listed in figure 11. If services can be changed independently from each other, this leads to a modular structure. Since the services in one business responsibility of GTI are not independent from each other, functional decomposition would not lead to a modular architecture.
- **Communication-oriented decomposition.** For each communication with external entities, define one component. There are three variants of this guideline.
  - ***Device-oriented.*** For each device to be communicated with, define a component that handles this communication. One uses this guideline to hide the peculiarities of a device from the rest of the software and to restrict changes in devices to one component. Since devices often can be changed independently from each other, this leads to a modular structure. The identification of a web interface component in our example is device orientation.
  - ***Actor-oriented.*** For each actor to be communicated with, define a component that handles this communication. For example, one may encapsulate the dialog with this actor in a separate component. If dialogs are independent from each other, this leads to a modular structure. The web interface in our example may be internally structured by actor.
  - ***Event-oriented.*** For each event to be responded to, define a separate component. This is called event-partitioning by McMenamin and Palmer [15]. If events can be recognized independently from each other, this leads to a modular architecture. In the case of GTI, event-orientation would lead to the definition of one component for each column in figure 11. Since the events handled in one business responsibility are not independent from each other, this would not lead to a modular architecture. In a multi-channeling architecture, where one event can be communicated to the system in a variety of ways, event-partitioning leads the idea of a front-office in which stimuli are analyzed to identify the underlying event.
- **Behavior-oriented decomposition.** For each process in the environment to be monitored or controlled, define one system component. If different processes are independent from each other, this leads to a modular architecture.

We used this guideline for GTI because each business responsibility contains one business process and it is mapped to one application architecture component.
- **Subject-oriented decomposition.** For each part of the world about which data must be maintained, define one system component. This is the standard guideline in databases and also an important guideline in object-oriented architecture design. In GTI, we used this to partition the databases into three parts, that deal with reservations, holiday homes and feedback, respectively.

As illustrated by the GTI example, we can use several guidelines to design an architecture. Different parts of an architecture may be identified by different guidelines.

These guidelines have been known in the literature for some time but they have not yet been systematized. Our contribution is to expose their underlying system: Each of these guidelines corresponds to one of the functional system aspects of our architecture framework (figure 1). Each of these guidelines map some aspect of the environment to some part of the system architecture. This transfers the modularity, or lack of it, of the environment to the system architecture. For example, if functions are independent from each other, then functional decomposition leads to a modular architecture; but if functions are dependent on each other, so that a change in one leads to a change in another, then functional decomposition does not lead to a modular architecture. And so on for the list of architecture design guidelines. The role of these guidelines with respect to other architectural concerns, such as maintanability and flexibility, remains a topic for further research.

## 7 Discussion and Conclusions

Our design approach picks up some ideas of Information Engineering [16] but replaces data-orientation by event-orientation. We view the business and its supporting software system as reactive systems that respond to signals and changes in conditions in their environment and to significant moments in time. Where IE maps business processes to information systems, we map business events to business services, some of which will be realized as application services.

Our approach refines that of Henderson and Venkatraman [4] by providing operationalized guidelines for aligning a requirements-level application architecture to the business architecture. We believe that it is the first comprehensive framework and design approach that links business requirements to application architecture.

Current work includes the use of the framework to collect and analyze architecture design practices in large information-processing organizations such as banks, insurance companies and government organizations. Our aim is to make a catalog of guidelines that have proven to be useful in large-scale information systems. An important class of guidelines concerns the choice of descriptions to produce for a particular architecture. Another class concerns the impact that a change in description will have on other descriptions. Another topic for further

research is the impact of architectural guidelines on architectural concerns such as maintainability and flexibility.

## Acknowledgements

Thanks are due to the company where we performed the GTI case study for their cooperation. This paper benefitted from comments by Pascal van Eck and by the anonymous reviewers on an earlier version.

## References

1. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice. Addison-Wesley (1998)
2. Hofmeister, C., Nord, R., Soni, D.: Applied Software Architecture. Addison-Wesley (2000)
3. Shaw, M., Garlan, D.: Software Architecture: Perspective on an Emerging Discipline. Prentice Hall (1996)
4. Henderson, J., Venkatraman, N.: Strategic alignment: leveraging information technology for transforming operations. IBM Systems Journal **32** (1993) 4–16
5. Wieringa, R.: Requirements Engineering: Frameworks for Understanding. Wiley (1996)
6. Wieringa, R.: A survey of structured and object-oriented software specification methods and techniques. ACM Computing Surveys **30** (1998) 459–527
7. Zachman, J.: A framework for information systems architecture. IBM Systems Journal (1987) 276–292
8. Hall, A.: Three-dimensional morphology of systems engineering. IEEE Transactions on System Science and Cybernetics **SSC-5** (1969) 156–160
9. Roozenburg, N., Eekels, J.: Product design: Fundamentals and Methods. Wiley (1995)
10. Olle, T., Hagelstein, J., Macdonald, I., Rolland, C., Sol, H., van Assche, F., Verrijn-Stuart, A.: Information Systems Methodologies: A Framework for Understanding. Addison-Wesley (1988)
11. Harel, D., Pnueli, A.: On the development of reactive systems. In Apt, K., ed.: Logics and Models of Concurrent Systems. Springer (1985) 477–498 NATO ASI Series.
12. Kruchten, P.: The 4+1 view model of architecture. IEEE Software **12** (1995) 42–50
13. Sowa, J., Zachman, J.: Extending and formalizing the framework for information systems architecture. IBM Systems Journal **31** (1992) 590–616
14. Wieringa, R.: Design Methods for Reactive Systems: Yourdon, Statemate and the UML. Morgan Kaufmann (2003)
15. McMenamin, S.M., Palmer, J.F.: Essential Systems Analysis. Yourdon Press/Prentice Hall (1984)
16. Martin, J.: Information Engineering. Prentice-Hall (1989) Three volumes.